Washington University in St. Louis
Washington University Open Scholarship

Engineering and Applied Science Theses & Dissertations

McKelvey School of Engineering

Summer 8-15-2017

Parallel Real-Time Scheduling for Latency-Critical Applications

Jing Li
Washington University in St. Louis

Follow this and additional works at: https://openscholarship.wustl.edu/eng_etds

Part of the Computer Engineering Commons

Recommended Citation

Li, Jing, "Parallel Real-Time Scheduling for Latency-Critical Applications" (2017). *Engineering and Applied Science Theses & Dissertations*. 302.
https://openscholarship.wustl.edu/eng_etds/302

This Dissertation is brought to you for free and open access by the McKelvey School of Engineering at Washington University Open Scholarship. It has been accepted for inclusion in Engineering and Applied Science Theses & Dissertations by an authorized administrator of Washington University Open Scholarship. For more information, please contact digital@wumail.wustl.edu.

WASHINGTON UNIVERSITY IN ST. LOUIS

School of Engineering and Applied Science

Department of Computer Science and Engineering

Dissertation Examination Committee:
Chenyang Lu, Chair
Kunal Agrawal, Co-Chair
Sameh Elnikety
Christopher Gill
Roch Guérin
I-Ting Angelina Lee

Parallel Real-Time Scheduling for Latency-Critical Applications

by
Jing Li

A dissertation presented to
The Graduate School
of Washington University in
partial fulfillment of the
requirements for the degree
of Doctor of Philosophy

August 2017

St. Louis, Missouri

Table of Contents

List of Tables

List of Figures

Acknowledgments

During my graduate study, I have received enormous help and support from many people, and this thesis would not have been possible without all of them.

First and foremost, I would like to thank my advisors, Dr. Chenyang Lu and Dr. Kunal Agrawal, who are my dearest academic father and mother. They taught me every aspect of research, from technical knowledge, basic research skills, to big picture and research taste. They always believe in me, help me to get through difficulties and encourage me to reach for the stars. I have learned from them more than just research. All of these would benefit me for my academic career and the rest of life.

I was fortunate to have worked closely with many brilliant collaborators. I would like to express my thanks and appreciation to Dr. Christopher Gill, Dr. Benjamin Moseley, Dr. I-Ting Angelina Lee and Dr. Jian-Jia Chen for always sparking inspiring ideas about research and providing generous help to my work. The collaborative work with all of them has become important parts of my dissertation. I am also grateful to Dr. Sameh Elnikety, Dr. Yuxiong He and Dr. Kathryn S. McKinley for supervising my internship at Microsoft Research in 2014, stimulating interesting research directions and giving valuable comments on my work. I especially want to thank Sameh for his support during my job search.

I am thankful for my fellow collaborators who made this thesis possible: Abusayeed Saifullah, David Ferry, Kefu Lu, Son Dinh, Shaurya Ahuja, Kevin Kieselbach and Zheng Luo, with whom I have enjoyed debugging codes, solving math and writing papers together. I extend my thanks to the students in the Parallel Computing Technology Group that I sadly did not have the chance to write papers with: Robert Utterback, Jordyn Maglalang, James Orr, and Ramsay Shuck. Many thanks also to the great collaborators from Purdue University: Dr. Shirley Dyke and Dr. Arun Prakash, Gregory Bunting and Amin Maghareh.

My heartiest gratitude also goes to all the past and current members of the Cyber-Physical Systems Laboratory (CPSL): Sisu Xi, Mo Sha, Chengjie Wu, Bo Li, Lanshun Nie, Rahav Dor, Chong Li, Dolvara Gunatilaka, Chao Wang, Yao Yuan, Yehan Ma, Haoran Li and Xin Hu. I have spent countless joyful hours with them, discussing research and enjoying wonderful times together. Special thanks to Bo Li, who helps me restore confidence and gives me critical suggestions whenever I struggle in life. Also thanks to my friends in WashU and many other friends outside school, including Ming Yin, Hao Yan, Xin Chen, Meng Xu, Hongxing Liu, Han Lu, Shenmeng Xu, Hanyang Chen and Zihang Xiao.

I would like to take this opportunity to express my gratitude to the Department of Computer Science and Engineering at Washington University in St. Louis. I have greatly enjoyed my graduate study here and received all kinds of help from the friendly department staff and faculty members. I especially want to thank our Department Chair and my committee member, Dr. Roch Guérin, for passionately making our department a wonderful place for study and research, and for being extremely supportive to my graduate study over the years.

Last but not the least, I am grateful to my parents for giving me endless love and for always being there whenever I need them. Special thank to my husband Mo Yu, for accompanying me across the continent during my entire graduate study and for making my life full of pleasure. I could not have become who I am now without all of you.

Jing Li

Washington University in St. Louis

August 2017

Dedicated to my family.

ABSTRACT OF THE DISSERTATION

Parallel Real-Time Scheduling for Latency-Critical Applications

by

Jing Li

Doctor of Philosophy in Computer Science

Washington University in St. Louis, 2017

Professor Chenyang Lu, Chair

Professor Kunal Agrawal, Co-Chair

In order to provide safety guarantees or quality of service guarantees, many of today's systems consist of latency-critical applications, e.g. applications with timing constraints. The problem of scheduling multiple latency-critical jobs on a multiprocessor or multicore machine has been extensively studied for sequential (non-parallizable) jobs and different system models and different objectives have been considered. However, the computational requirement of a single job is still limited by the capacity of a single core. To provide increasingly complex functionalities of applications and to complete their higher computational demands within the same or even more stringent timing constraints, we must exploit the internal parallelism of jobs, where individual jobs are parallel programs and can potentially utilize more than one core in parallel. However, there is little work considering scheduling multiple parallel jobs that are latency-critical.

This dissertation focuses on developing new scheduling strategies, analysis tools, and practical platform design techniques to enable efficient and scalable parallel real-time scheduling for latency-critical applications on multicore systems. In particular, the research is focused on two types of systems: (1) static real-time systems for tasks with deadlines where the temporal properties of the tasks that need to execute is known a priori and the goal is to guarantee the temporal correctness of the tasks prior to their executions; and (2) online systems for latency-critical jobs where multiple jobs arrive over time and the goal to optimize for a performance objective of jobs during the execution.

For static real-time systems for parallel tasks, several scheduling strategies, including global earliest deadline first, global rate monotonic and a novel federated scheduling, are proposed, analyzed and implemented. These scheduling strategies have the best known theoretical performance for parallel real-time tasks under any global strategy, any fixed priority scheduling and any scheduling strategy, respectively. In addition, federated scheduling is generalized to systems with multiple criticality levels and systems with stochastic tasks. Both numerical and empirical experiments show that federated scheduling and its variations have good schedulability performance and are efficient in practice.

For online systems with multiple latency-critical jobs, different online scheduling strategies are proposed and analyzed for different objectives, including maximizing the number of jobs meeting a target latency, maximizing the profit of jobs, minimizing the maximum latency and minimizing the average latency. For example, a simple First-In-First-Out scheduler is proven to be scalable for minimizing the maximum latency. Based on this theoretical intuition, a more practical work-stealing scheduler is developed, analyzed and implemented. Empirical evaluations indicate that, on both real world and synthetic workloads, this work-stealing implementation performs almost as well as an optimal scheduler.

Chapter 1

Introduction

In order to provide safety guarantees or quality of service guarantees, many of today's systems consist of latency-critical applications. For example, in cyber-physical systems, such as autonomous vehicles [96], avionic systems [127] and hybrid structural testing [120], computers interact with humans or the physical environment. To guarantee safety and stability of such systems, the computation of an application must complete by a specific deadline. In interactive cloud services, such as web search, stock trading, ads, and online gaming, the services with the most fluid and seamless responsiveness incur a substantial competitive advantage in attracting and captivating users over less responsive systems [56, 57, 83, 155]. Therefore, these systems strive to minimize latency. With the growing number of application domains where these kinds of guarantees are needed, there is an increasing need for systems that can run complex applications with timing constraints.

In addition, today all computing platforms, from cellphones to desktops to clouds, are parallel machines. The imperatives to reduce power consumptions as well as assembly and production costs are pushing system deployment toward combining multiple applications onto a common platform. Examples span from clouds where multiple clients submit their applications to consolidated Electronic Control Units on modern cars that host multiple applications with diverse functionalities from brake control to infotainment. As this integration continues, we face challenges in designing platforms which can provide appropriate timing guarantees to each application while preserving scalability.

The problem of scheduling multiple latency-critical applications (jobs) on a multiprocessor or multicore machine has been extensively studied for *sequential* (non-parallizable) jobs and different system models and different objectives have been considered [3, 5, 7, 10, 14, 17, 19, 27–30, 40, 49, 50, 54, 74, 76, 105, 109, 117, 144, 149]. In this case, since jobs can only be executed sequentially and use one core at a time, the system can only exploit parallelism by running multiple jobs simultaneously on a multicore system. Therefore, increasing the number of cores allows us to increase the number of jobs the system can schedule, but the computational requirement of a single job is still limited by the capacity of a single core.

Since the processor speed in today's hardware platforms has stagnated and systems are only becoming more parallel, there has been a continually increasing interest in how to harness multicore more effectively. Moreover, to provide increasingly complex functionalities of applications, the computational demand of each individual job increases significantly. To complete higher computational demands within the same or even more stringent timing constraints, we must exploit the *internal parallelism* of jobs, where individual jobs are parallel programs and can potentially utilize more than one core in parallel. Many languages and libraries, such as such as Cilk [32], Intel Cilk Plus [89], Threading Building Blocks [134], OpenMP [125], Microsoft's Task Parallel Library [107], IBM X10 [147], have been designed to allow programmers to write parallel programs. In these languages, the programmer expresses algorithmic parallelism, through linguistic constructs such as "spawn" and "sync," or parallel-for loops.

While researchers have looked at how to schedule multiple sequential jobs with various timing constraints and latency-related objectives, there is little work considering how to schedule multiple parallel jobs. In order to exploit the untapped efficiencies in the multicore platforms and improve the quality of service guarantees to applications, *this thesis focuses on developing theoretical foundations and practical implementations of parallel real-time systems for latency-critical applications.* For different types of real-time systems with different timing requirements, this thesis develops new scheduling strategies, analysis tools, and practical platform design techniques in particular to applications that are parallel programs.

In the remainder of this chapter, Section 1.1 first classifies real-time systems with latency-critical applications into two types and discusses the challenges encountered when exploiting internal parallelism of applications. Section 1.2 introduces how to model and schedule parallel programs. The thesis statement is presented in Section 1.3, followed by the contribution and organization of the thesis in Section 1.4.

1.1 Systems with Latency-Critical Applications

This thesis designs and implements provably good and practically efficient scheduling strategies for executing multiple latency-critical applications. Latency-critical applications arise in many real-world systems, where computation processes interact with humans and/or the physical environment. In this thesis, based on the types of timing constraints and models of the systems, we classify these systems into two types.

Static real-time systems for tasks with deadlines: These systems are time-critical and have a high cost of failure. For instance, in earthquake engineering, a hybrid-testing framework studies the dynamic response of the physical structure. This framework connects the physical specimen within a closed loop through sensors and actuators to an intensive numerical simulation of the large remaining structure. In order to guarantee system safety and stability, the computation must be completed by a specific deadline. Therefore, the goal of these systems is to *guarantee the temporal correctness* of the applications *prior to* their executions, which is crucial for keeping the systems stable and safe.

To provide such timing guarantees, these systems are usually modeled statically, where the temporal properties of the tasks that are known *a priori*. Specifically, the arrival patterns and computational demands of jobs are known in advance and modeled as reoccurring *real-time tasks*. In these systems, a real-time scheduler must determine the schedulability (in terms of meeting deadlines) for a given task set at design time. While guaranteeing schedulability, the scheduler can optimize for higher utilization of the system. The static real-time system model characterizes many embedded systems and cyber-physical systems, such as those in autonomous vehicles, avionics and robotic.

For static real-time systems, Part I in this thesis for the first time considers scheduling general parallel tasks that are programs written in parallel languages. This creates new challenges to the design, analysis and implementation of platforms. Techniques for designing and analyzing schedulers for sequential jobs do not generalize to parallel tasks due to three reasons. (1) Unlike a sequential job, a parallel job has internal structure that dictates when its sub-computations can be executed. (2) The internal scheduling of each job interacts with the scheduling between jobs. (3) This internal structure may be unknown beforehand.

Part I develops theoretical techniques for analyzing parallel tasks and proved bounds for different real-time schedulers. To compare the performance between different schedulers for parallel real-time tasks, this thesis proposes a *capacity augmentation bound*, which directly tells us how many resources a scheduler needs. Roughly speaking, a scheduler with a capacity bound of b can handle a load about $1/b$ of the capacity (with an additional constraint on the parallel tasks critical-path lengths).

Online real-time systems for latency-critical jobs: In many application environments such as clouds, grids and shared servers, clients send jobs to be processed on a server and the server scheduler decides when to process them. The goal of the scheduler is both to use the resources efficiently and to provide a good quality of service to jobs. In these systems, jobs arrive *over time* and the systems do not know the existence of jobs until they arrive. Instead of guaranteeing schedulability prior to execution (which would be impossible), online schedulers try to *optimize some application-specific performance objectives.*

In interactive services on cloud, the *response time (latency, flow time)* determines the service experience of a single user. For example, in an online search service users send search requests to servers in the cloud and the service needs to respond within 100ms for users to find the service responsive. With multiple users to serve, system administers optimize an objective of the accumulated performance over all users. The specific objective depends on the scenario. For instance, to provide good average quality of service among all users, the system can optimize over its average latency. In contrast, to make sure that even the worst-case latency any user experienced is still acceptable, the scheduler can minimize the maximum latency.

While online scheduling has been well-studied for sequential jobs, there is little work on parallel jobs. Scheduling parallel jobs online presents unique challenges, because parallel jobs can behave in counter-intuitive ways. For instance, intuitively, if we run the same set of jobs using the same scheduler on a faster and a slower machine, the faster machine should perform better. This intuition is correct for sequential jobs and standard techniques for analyzing online algorithms rely on this fact. Surprisingly, this intuition does not hold for parallel jobs. In particular, a greedy scheduler may actually fall behind in the aggregate amount of work processed, when compared to the same scheduler with less speed. Therefore, to analyze parallel jobs, we had to develop new techniques since standard techniques do not directly apply.

Part II presents the first nontrivial results considering scheduling parallel jobs online for different objectives. To compare the theoretical performance of different schedulers, we define that a scheduler A is *s-speed c-competitive*, if A is c-competitive when executed on a s-speed machine while the optimal offline algorithm is on a unit speed machine. We say that a scheduler is *scalable*, if it only needs the minimum possible extra speed augmentation to have constant competitiveness compared to the optimal scheduler.

1.2 Parallel Scheduling for Applications

During the last decade, the performance increase of processor chips has come primarily from increasing numbers of cores. This has led to extensive work on developing parallel languages and runtime systems for applications to utilize multiple cores at the same time. Examples include Intel Cilk Plus [89], Threading Building Blocks [134] and OpenMP [125], etc. Using these languages, the programmers only need to express algorithmic parallelism, but do not need to provide any mapping from sub-computations to cores — it is the job of the parallel runtime systems to execute the work of each job onto multiple cores efficiently. Parallel programs under this approach are also known as dynamic multithreaded programs. A single parallel job (an execution of a parallel program) can be represented as a *directed acyclic graph (DAG)* where each node represents a sequence of instructions (thread) and each edge represents a dependency between nodes.

Scheduling a single parallel job has been studied extensively in the parallel computing literature. Parallel runtime systems generally use work-stealing as a scheduler since it is known to be an efficient scheduler for such programs both theoretically and in practice [33,75]. A single parallel job having W *work* — the running time on 1 processor, and P *critical-path length* — the length of the critical path (the longest path in the program), can be executed in $O(W/m + P)$ (expected) time on m processors (or workers) using a work-stealing scheduler. This running time is asymptotically optimal and guarantees linear speedup for programs with sufficient parallelism.

However, the problem of how to schedule these parallel programs in multiprogrammed environments where a quality of service guarantee must be provided is not well studied. For static real-time systems, there have been some prior work that are based on task decomposition [103, 123, 137, 138], which first decomposes each parallel task into a set of sequential subtasks with assigned intermediate release times and deadlines, and then schedules these sequential subtasks using a known multiprocessor scheduling algorithm. Decomposition techniques require a thorough knowledge of the structure of tasks as well as the individual worst case execution time of each subtask prior to execution. Such knowledge is expensive to acquire and may be inaccurate, pessimistic or even unavailable when tasks from different vendors are integrated on a common computing platform. Moreover, decomposition introduces implementation complexities in real systems.

For online systems with multiple parallel DAG jobs, there has been some prior work on how to allocate processors to programs in a fair and efficient manner [2, 84], but none of the work considers how to optimize for latency-related objectives. For online system optimizing for latency-related objectives, a parallelism model, called arbitrary speed-up curve model, has been considered. In this model, each job i is associated with a sequence of phases, where each phase is associated with a function that specifies the processing rate of the phase on a given number of cores. The speed-up curve model was first introduced by [63] and used later in [12, 46–48, 65, 73, 80, 81]. While the speed-up curve model is a theoretically elegant model, most languages and libraries generate parallel programs that are more accurately modeled using DAGs. The speed-up curve model also cannot be simulated using the DAG model. For example, in the speed-up curve model one could have a speed-up curve with a processing rate of $\sqrt{m}$, when given m cores. In the DAG setting, a job's parallelizability is linear up to the number of nodes ready to be scheduled and thus it is unclear how to simulate this speed-up curve.

1.3 Thesis Statement

This thesis designs and implements *provably good and practically efficient scheduling strategies* for executing multiple parallel latency-critical applications. For both types of systems described in Section 1.1, different scheduling strategies are designed and proved to have the best known theoretical performance for the respective scheduling problems. Significant attention is paid to both the theoretical performance and the practicality of the proposed scheduling strategies. Therefore, many of the strategies are improved to be more efficient and scalable, and are implemented in middleware platforms to provide scheduling service to applications written in widely used parallel languages. The new scheduling strategies, analysis tools, and practical platform design techniques presented in this thesis supports the following statement.

Thesis Statement: By appropriately leveraging the internal parallelism of latency-critical applications, static and online real-time systems are able to exploit the untapped efficiencies in the multicore platforms to drastically improve the quality of service guarantees of applications with increasing computational demands.

1.4 Thesis Contributions

In this section, we briefly summarize the organization and contributions of this thesis. The results of this thesis is from the collaboration with many co-authors: Kunal Agrawal, Shaurya Ahuja, Jian-Jia Chen, Sameh Elnikety, David Ferry, Christopher Gill, Yuxiong He, Kevin Kieselbach, I-Ting Angelina Lee, Chenyang Lu, Kefu Lu, Mahesh Mahadevan, Benjamin Moseley, Kathryn S. McKinley.

Part I of the thesis focuses on the real-time scheduling problems for tasks in static real-time systems requiring different guarantees. The contributions of this part include:

Scheduling parallel tasks with hard real-time constraints:

- The well known global earliest-deadline-first scheduling (GEDF) is applied to schedule parallel tasks with hard real-time constraints and is proved to have a capacity augmentation bound of 2.618, which is the best known bound for parallel DAG tasks under any global strategy (Chapter 3).
- The widely used global rate-monotonic scheduling is applied to parallel real-time tasks and is proved to have a capacity augmentation bound of 3.732, which is the best known bound for parallel DAG tasks under any fixed priority scheduling (Chapter 3).
- A lower bound on the capacity augmentation of GEDF is presented showing that the bound of 2.618 is tight when the number of cores is sufficiently large (Chapter 3).

- A novel scheduling strategy, namely federated scheduling, is proposed for parallel real-time tasks and is proved to have the best capacity augmentation bound of 2 under any strategy, which is shown via a lower bound example (Chapter 4).
- A PGEDF platform is developed, which provide GEDF scheduling service to parallel programs written in the widely used OpenMP language (Chapter 3).
- A RTCG platform is developed, which provide federated scheduling service to parallel programs written in OpenMP (Chapter 3).
- Empirical experiments on randomly generated task sets show that RTCG generally has better schedulability in practice (Chapter 3).

Scheduling parallel tasks in mixed-criticality real-time systems:

- Federated scheduling strategy is generalized to real-time systems with two criticality levels, named MCFS, and is proved to have a capacity augmentation bound of 3.67 for large number of cores, which to our knowledge is the first such performance bound for parallel mixed-criticality tasks (Chapter 5).
- For tasks with utilization at least one, MCFS is applied to both dual-criticality and multi-criticality systems and is proved to have capacity augmentation bounds of 3.41 and 3.62 , respectively (Chapter 5).
- An implementation of an MCFS runtime system in Linux is presented, which supports parallel programs written in OpenMP, and is evaluated through empirical experiments to demonstrate the practicality of MCFS (Chapter 5).

Scheduling parallel tasks with soft real-time constraints:

- A federated scheduling strategy that addresses average-case workloads is proposed and is proved to have a stochastic capacity augmentation bound of 2, which is the first known result for stochastic parallel tasks with soft real-time constraints (Chapter 6).

- To support scalable soft real-time computing, a Real-Time Work-Stealing platform (RTWS) is developed, which is a real-time extension to the widely used Cilk Plus concurrency platform (Chapter 7).

- For large scale soft real-time systems, experimental evaluations show that RTWS outperforms RTCG in term of deadline miss ratio, relative response time and resource efficiency (Chapter 7).

Part II of the thesis focuses on the online scheduling problems for parallel latency-critical jobs that optimize different latency-related objectives. The contributions of this part include:

Online Scheduling of parallel jobs to maximize profit:

- For maximizing the number of jobs that meet a single target latency during an execution, a new adaptive work stealing policy, called tail-control, is designed, which utilizes instantaneous job progress and system load to choose when to parallelize job by stealing, when to admit new jobs, and when to limit parallelism of large jobs (Chapter 8).

- Tail-control is implemented in the Intel Threading Building Blocks (TBB) library and is shown to significantly outperforms three baseline work-stealing schedulers (steal-first, admit-first, and default TBB) on real-world workloads, achieving up to a 58% reduction in the number of jobs that exceed the target latency (Chapter 8).

- For maximizing the profit of finishing jobs by their individual deadlines during an execution, the first non-trivial results is presented, showing an $O(1)$-competitive algorithm using resource augmentation (Chapter 9).

- For maximizing the profit of jobs with general profit functions, an algorithm that is $O(1)$-competitive using resource augmentation is presented (Chapter 9).

Online Scheduling of parallel tasks to minimize the maximum flow time:

- The simple First-In-First-Out scheduler is analyzed and proved to be $(1+\epsilon)$-speed $O(\frac{1}{\epsilon})$-competitive for any $\epsilon > 0$, which the first known non-trivial results for maximum flow time in the DAG model (Chapter 10).

- A more practical work-stealing scheduler is proposed and shown to have a maximum flow time of $O(\frac{1}{\epsilon^2}\max\{\mathrm{OPT}, \ln(n)\})$ for n jobs, with $(1+\epsilon)$-speed, which is essentially tight as a lower bound of $\Omega(\log(n))$ for work-stealing is presented (Chapter 10).

- The proposed work-stealing scheduler for minimizing maximum flow time is implemented in Thread Building Block and is shown to have comparable performance compared to a simulated optimal scheduler on realistic and synthetic workloads (Chapter 10).

- For the case where jobs have weights (typically representing priorities) and the objective is minimizing the maximum weighted flow time, a non-clairvoyant algorithm Biggest-Weight-First (BWF) is proposed and shown to be $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^2})$-competitive for any $\epsilon > 0$, which is essentially the best positive result that can be shown in the online setting for the weighted case due to strong lower bounds without resource augmentation (Chapter 10).

Online Scheduling of parallel tasks to minimize average flow time:

- The Latest-Arrival-Processor-Sharing (LAPS) algorithm is proved to be scalable which is $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^3})$-competitive for any $\epsilon > 0$ (Chapter 11).

- The first greedy algorithm is presented, which is a generalization of the shortest jobs first algorithm, and is proved to be $(2+\epsilon)$-speed $O(\frac{1}{\epsilon^4})$-competitive for any $\epsilon > 0$ (Chapter 11).

- A non-clairvoyant distributed scheduling algorithm Distributed Random Equi-Partition (DREP), which requires minimal synchronization and a small bounded number of preemptions, is proposed and proved to be $(2+\epsilon)$-speed $O(\frac{1}{\epsilon^3})$-competitive (Chapter 12).

Chapter 2

Preliminaries and Notation

This chapter provides an introduction to parallel job model, static real-time system model and online real-time system model that are used throughout this thesis.

2.1 Parallel Job Model

We first describe the types of parallel jobs considered in this thesis. Specifically, we are interested in parallel programs that can be generated using parallel languages and libraries, such as Cilk [32], Intel Cilk Plus [89], Threading Building Blocks [134], OpenMP [125], Microsoft's Task Parallel Library [107], IBM X10 [147], etc. In these languages, the programmer expresses algorithmic parallelism, through linguistic constructs such as "spawn" and "sync," "fork" and "join," or parallel-for loops.

These programs can be modeled using **directed acyclic graphs (DAGs)**. Each node (subtask) in the DAG represents a sequence of instructions (a strand) and each edge represents a dependency between nodes. A node (subtask) is *ready* to be executed when all its predecessors have been executed. Multiple ready nodes for the same job can be scheduled simultaneously, but each core can only execute one node at a time. A job is completed only once all of the nodes in its DAG have been completely processed. We assume the scheduler knows the ready nodes for a job at a point in time, but does not know the DAG structure a priori; the DAG unfolds dynamically as the job executes. Figure 2.1 shows an example of DAG job with 6 nodes.

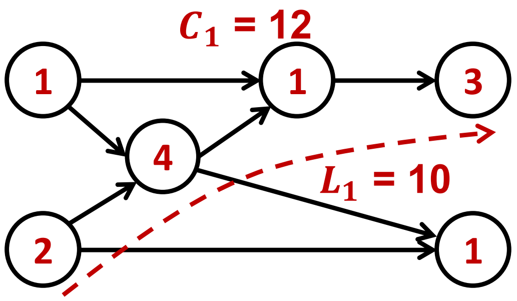


Figure 2.1: A directed acyclic graph (DAG) job J_1 with six nodes. The execution time of each node is annotated in the center of the node. The total work C_1 is the sum of the execution times of all nodes, which is 12. The critical-path, i.e., the longest path in the DAG, is annotated using the dashed line. The critical-path length L_1 is 10.

Throughout this thesis, it is not necessary to build the analysis based on specific structure of the DAG. Instead, only two parameters related to the execution pattern of job J_i are defined:

- **total execution time (or work)** C_i of job J_i: This is the summation of the execution times of all the subtasks of task τ_i.
- **critical-path length (or span)** L_i of job J_i: This is the length of the critical-path in the given DAG. Critical-path length is the execution time of the job on an infinite number of cores.

Note that by this definition, critical-path length of a sequential job is equal to its work. Figure 2.1 shows an example of DAG task with the critical-path annotated following the dashed line.

Both the work and critical-path length of a task can be measured by profiling tools. For example, both parameters of a Cilk Plus program can be measured using Cilkview [85] or Cilkprof [139].

In general, parallel programs can have arbitrary DAG structures. In real-time scheduling, researchers have given special consideration to a subset of DAG tasks, where the programs *only* use the parallel-for construct and do not nest these parallel-for loops. This restriction generates a special type of DAG, which we call **synchronous DAG**. Each parallel for-loop is represented by a **segment** — a segment contains a set of nodes (iterations) that can be executed in parallel with each other. The end of each segment is a synchronization point and the next segment can begin only after all iterations of the current segment complete. A sequential region of code is simply a segment with 1 iteration. Each synchronous task is a sequence of such segments. Synchronous tasks are also called as Fork/Join tasks in some publications. Figure 2.2 shows an example of a synchronous task with five segments; two of them are parallel segments, and the remaining three are sequential segments. This synchronous structure can be generated from a simple program shown in Figure 2.3, where parallel_for constructs can be Cilk Plus' cilk_for constructs or OpenMP's omp for directives.

Theoretically, there is no difference between scheduling the general DAG and synchronous task. However, in practice, they are usually implemented using different parallel programming constructs. For example, in Cilk Plus a node in a DAG is generated by cilk_spawn and the synchronization point is realized by cilk_sync. Similarly, in OpenMP they are generated by omp task and omp taskwait. As comparison, segments of synchronous tasks are supported by many different parallel languages like OpenMP and Cilk Plus in the form of the high-level parallel for programming construct.

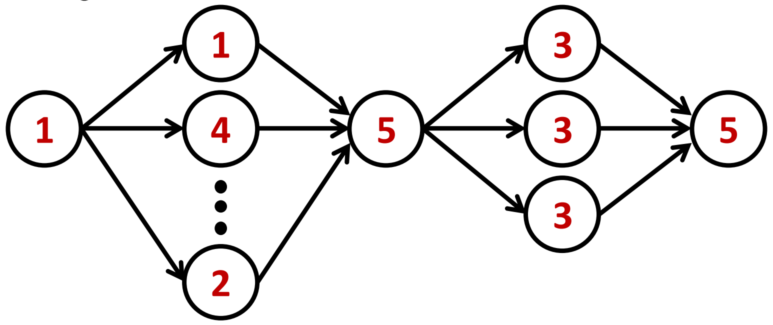


Figure 2.2: A synchronous task with two parallel-for loops. The execution time of each node is annotated in the center of the node. The second segment contains 20 nodes.

2.2 Scheduling Parallel Jobs

Most parallel languages and libraries, including those mentioned above, provide a runtime system that is responsible for scheduling the DAG on the available cores, i.e., dynamically

```
main()
{
    // Do some sequential work
    foo();
    // Do the first parallel segment
    parallel_for (i = 1; i <= 20; i++) {
        first_func();
    }
    // Other sequential work
    bar();
    // Do the second parallel segment
    parallel_for (i = 1; i <= 3; i++) {
        second_func();
    }
    // The last sequential work
    baz();
}
```

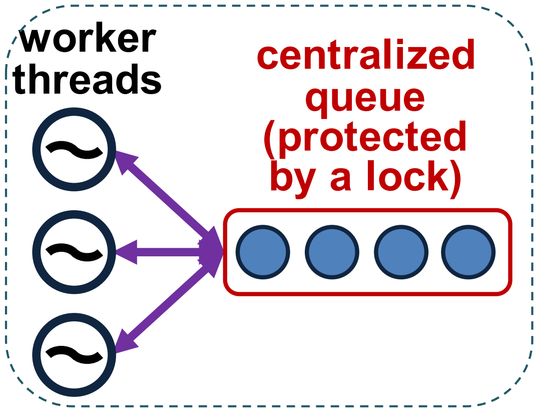
Figure 2.3: Example of a synchronous program.

dispatch the nodes of the DAG to these cores as the nodes become ready to execute. At a high-level, two types of scheduling strategies are often used: centralized scheduling and randomized work-stealing.

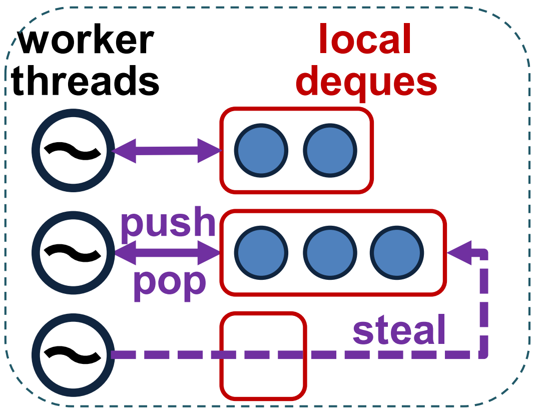
2.2.1 Centralized Greedy Schedulers

The system maintains a centralized data structure (such as a queue) of ready nodes that is shared by all the cores in a work sharing manner. There are a couple of possible instantiations of this strategy. In *push* schedulers, there is a master thread that dispatches work to other threads as they need this work. In *pull* schedulers, worker threads access this data structure themselves to grab work (ready nodes) as they need them. For example, the scheduler in the runtime system of GNU OpenMP is a pull scheduler, as in Figure 2.4(a).

Work-sharing schedulers have the nice property that they are **greedy** or **work-conserving** — as long as there are available ready nodes, no worker idles. However, these schedulers often have high overheads due to constant synchronizations. In particular, in a push scheduler, the master thread can only send work to cores one at a time. In a pull scheduler, the centralized queue must be protected by a lock and often incurs high overheads due to this.



(a) A pull scheduler



(b) A work-stealing scheduler

Figure 2.4: Examples of centralized scheduling and work stealing

2.2.2 Randomized Work-Stealing Schedulers

In a randomized work-stealing scheduler, there is no centralized queue and the work dispatching is done in a distributed manner [32]. If a job is assigned n_i cores, the runtime system creates n_i worker threads for it. Each worker thread maintains a local double-ended queue (a *deque*), as shown in Figure 2.4(b). When a worker generates new work (enables a ready node from the job's DAG), it pushes the node onto the bottom of its deque. When a worker finishes its current node, it pops a ready node from the bottom of its deque. If the local deque is empty, the worker thread becomes a *thief* and randomly picks a *victim* thread among the other workers working on the same task and tries to steal work from the top of the victim's deque. For example, the third worker thread's deque is empty in Figure 2.4(b), so it randomly picks the second worker thread and steals work.

Given a single job, randomized work-stealing attains asymptotically optimal and scalable performance with high probability [8,33]. It is also very efficient in practice and the amount of scheduling and synchronization overhead is small. In contrast to centralized schedulers where the threads synchronize frequently, very little synchronization is needed in work-stealing schedulers since (1) workers work off their own deques most of the time and don't need to communicate with each other at all and (2) even when a worker runs out of work and steals occur, the thief and the victim generally look at the opposite ends of the deque and don't conflict unless the deque has only 1 node on it. In addition, work-stealing often has good cache performance, again since workers work on their own deques.

However, because of this randomized and distributed characteristic, work-stealing is not strictly greedy (work conserving). In principle, workers can spend a large amount of time stealing, even if some other worker has a lot of ready nodes available on its deque. On the other hand, work-stealing provides strong probabilistic guarantee of linear speedup ("near-greediness") [148]. Moreover, it is much more efficient than centralized schedulers in practice. Therefore, variants of work stealing are the default strategies in many parallel runtime systems such as Cilk, Cilk Plus, TBB, X10, and TPL [32, 89, 107, 134, 147]. Thus, for soft real-time systems where occasional deadline misses are allowed, work stealing can be more resource efficient than a strictly greedy scheduler.

2.3 Parallel Languages and Runtime Systems

OpenMP is a programming interface standard [125] for C, C++, and FORTRAN that allows a programmer to specify where parallelism can be exploited, and the GNU OpenMP runtime library in GCC is one of implementations of the OpenMP standard. OpenMP allows programmers to express parallelism using compiler directives. In particular, parallel for loops are expressed by `#pragma omp parallel for`, a parallel node in a DAG is expressed by `#pragma omp task` and synchronization between omp tasks is expressed by `#pragma omp taskwait`. While the details of scheduling are somewhat complex, and vary between omp parallel for loops and omp tasks, at a high level, GNU OpenMP provides an instantiation of a centralized pull scheduler. Available parallel work of a program is kept in a centralized queue protected by a global lock. Whenever a worker thread generates nodes of omp tasks or iterations in a parallel for loop, it has to get the global lock and places these nodes in the queue. When it finishes its current work, it again has to grab the lock to get more work from the queue.

Cilk Plus is a language extension to C++ for parallel programs and its runtime system schedules parallel programs using randomized work stealing. All Cilk Plus features are supported by GCC. Potential parallelism can be expressed using three keywords in the Cilk Plus language: a parallel node in a DAG is generated by cilk_spawn and the synchronization point is realized by cilk_sync; additionally, parallel for-loops are supported using a cilk_for programming construct. Note that in the underlying Cilk Plus runtime system, cilk_for is expanded into cilk_spawn and cilk_sync in a divide and conquer manner. Therefore, there is no fundamental difference between executing parallel DAGs or synchronous tasks in Cilk Plus. The Cilk Plus runtime system implements a version of randomized work stealing. When a function spawns another function, the child function is executed and the parent is placed on the bottom of the worker's deque. A worker always works off the bottom of its own deque. When its deque becomes empty, it picks a random victim and steals from the top of that victim's deque.

2.4 Classic Static Real-Time System Model

In a classic static real-time system, the temporal correctness (i.e. schedulability) of applications must be guaranteed prior to their executions. Therefore, a priori (static) knowledge of jobs in the system described using a real-time task model is necessary. In this section, we describe the classic real-time task model and the theoretical performance bounds considered in this thesis.

Real-time task model: We consider a set τ of n independent sporadic real-time tasks $\{\tau_1, \tau_2, \ldots, \tau_n\}$. A **task** τ_i represents an infinite sequence of arrivals and executions of task instances (also called jobs). We consider the **sporadic task model** [26] where, for a task τ_i, the *minimum inter-arrival time* (or **period**) T_i represents the time between consecutive arrivals of task instances, and the **relative deadline** D_i represents the temporal constraint for executing the job. If a task instance of τ_i arrives at time t, the execution of this instance must be finished no later than the *absolute deadline* $t+D_i$ and the release of the next instance of task τ_i must be no earlier than t plus the minimum inter-arrival time, i.e. $t + T_i$. We consider **implicit deadline tasks** where each task τ_i's relative deadline D_i is equal to its minimum inter-arrival time T_i; that is, $T_i = D_i$. We consider the schedulability of this task set on a uniform multicore system consisting of m identical cores.

Each task $\tau_i \in \tau$ is a DAG program or a synchronous program — in principle, each job may have different internal structure. A task τ_i is characterised by its work C_i and L_i critical-path length. For systems with different temporal constraints, the specific definition for work and critical-path length varies. In particular, for systems with hard real-time constraints, i.e., all deadlines of all tasks must be met throughout the system execution, both parameters are the *worst-case* values. In other words, the work (critical-path length) of a task τ_i is the maximum work (critical-path length) of all its job instances. In contrast, for systems with soft real-time constraints, we can model the work (critical-path length) as a random variable. For systems with multiple criticality levels, we use two estimates of work (critical-path length): nominal work (critical-path length) representing the average computation of the task and the overload work (critical-path length) bounding the worst-case computation of the task.

Using the work and period of a task τ_i, we can calculate its *utilization* $u_i = \frac{C_i}{T_i} = \frac{C_i}{D_i}$ for implicit deadlines. The total utilization of the task set is $U_{\sum} = \sum_{\tau_i \in \tau} u_i$, which indicates the load of the task set. According to whether a task must run in parallel to meet its deadline, we classify tasks using their utilizations: task τ_i is a **low-utilization** task if $u_i = C_i/D_i \leq 1$ (and hence $C_i \leq D_i$); or it is a **high-utilization** task, if τ_i's utilization $u_i > 1$. By this classification, a low-utilization task can meet its deadline even if it is forced to run sequentially, while a high-utilization task needs at least two cores in order to complete its computation by this deadline.

Theoretical bounds for real-time schedulers: One can generally derive two types of performance bounds for real-time schedulers. The traditional bound is called a **resource augmentation bound** (also called a *processor speed-up factor*). A scheduler $\mathcal{S}$ provides a resource augmentation bound of $b \geq 1$ if it can successfully schedule any task set τ on m cores of speed b as long as the ideal scheduler can schedule τ on m cores of speed 1. A resource augmentation bound provides a good notion of how close a scheduler is to the optimal schedule, but it has a drawback. Note that the *ideal scheduler* is only a hypothetical scheduler, meaning that it always finds a feasible schedule if one exists. Unfortunately, since we often cannot tell whether the ideal scheduler can schedule a given task set on unit-speed cores, a resource augmentation bound may not provide a schedulability test.

Another bound that is commonly used for sequential tasks is a **utilization bound**. A scheduler $\mathcal{S}$ provides a utilization bound of b if it can successfully schedule any task set which has total utilization at most m/b on m cores.[1] A utilization bound provides more information than a resource augmentation bound; any scheduler that guarantees a utilization bound of b automatically guarantees a resource augmentation bound of b as well. In addition, it acts as a very simple schedulability test in itself, since the total utilization of the task set can be calculated in linear time and compared to m/b. Finally, a utilization bound gives an indication of how much load a system can handle; allowing us to estimate how much over-provisioning may be necessary when designing a platform. Unfortunately, it is often impossible to prove a utilization bound for parallel systems due to Dhall's effect; often, we can construct pathological task sets with utilization arbitrarily close to 1, but which cannot be scheduled on m cores.

[1] A utilization bound is often stated in terms of $1/b$; we adopt this notation in order to be consistent with the other bounds stated here.

Therefore, for parallel systems this thesis defined a concept of **capacity augmentation bound** which is similar to the utilization bound, but adds a new condition.

Definition 1 *Given a task set τ with total utilization of $U_{\sum}$, a scheduling algorithm $\mathcal{S}$ with **capacity augmentation bound** b can always schedule this task set on m cores of speed b as long as τ satisfies the following conditions on unit speed cores.*

$$\textit{Utilization does not exceed total cores, } \sum_{\tau_i \in \tau} u_i \leq m \tag{2.1}$$

$$\textit{For each task } \tau_i \in \tau, \textit{the critical path } L_i \leq D_i \tag{2.2}$$

Note that no scheduler can schedule a task set τ on m unit speed cores unless Conditions (2.1) and (2.2) are met. This definition can be equivalently stated (without reference to the speedup factor) as follows: Condition (2.1) says that the total utilization $U_{\sum}$ is at most m/b and Condition (2.2) says that the critical-path length of each task is at most $1/b$ of its relative deadline, that is, $L_i \leq D_i/b$. Note that a scheduler with a smaller b is better than another with a larger b.

A capacity augmentation bound is quite similar to a utilization bound: it also provides more information than a resource augmentation bound does; any scheduler that guarantees a capacity augmentation bound of b automatically guarantees a resource augmentation bound of b as well. It also acts as a very simple schedulability test. Finally, it can also provide an estimation of the load a system is expected to handle.

2.5 Classic Online Scheduling Model

In the online scheduling problem of multiple jobs, n jobs arrive over time and are scheduled on m identical processors (cores). In the *online* setting, the scheduler is only aware of the job at the time it arrives in the system. Hence, each job J_i has an **arrival time (release time)** r_i, which is the first time an online scheduler is aware of the job. Once a request arrives, it is **active** until its completion. A request is **admitted** once the system starts working on it. An active request is thus either executing or waiting. The time when job J_i completes under an online scheduler A is denoted as **completion time** c_i. Throughout this thesis, we consider the setting where each *individual* job is a parallel job represented by a Directed-Acyclic-Graph (DAG).

In today's systems, latency (response time) is often a very important measure of performance. The **latency (response time or flow time)** F_i of an job is the time elapsed between the time when the job *arrives* in the system and the time when the request completes, i.e., $F_i = c_i - r_i$. In an online scheduling problem, there are many different optimization objectives, most of which relate to the latency of jobs. For instance, to optimize for the responsiveness of an interactive service, the scheduler can be designed to minimize the maximum latency experienced by a job. If the completion of a job is only useful when its flow time is below a (single) target latency, then the scheduler must optimize the number of jobs that can meet the target latency. The target latency can be generalized to jobs with different deadlines and the scheduler tries to maximize the number of jobs that can meet their individual deadlines. This can be further generalized to system where jobs have different importances characterized as weight w_i — the weight is usually known to the scheduler when the job arrives and may not be correlated to the work of the job. For the unweighted setting, $w_i = 1$ for all jobs.

The online scheduling problem is well-studied theoretically for the case where each job is sequential, i.e. can only use one processor (core) at a time. For most online scheduling problems where the scheduler is only given partial knowledge of the arriving jobs during an execution, no online algorithm can be optimal for all the execution instances. Therefore, to evaluate the theoretical performance of an online scheduler, we compare the performance of the scheduler A with the performance $f(O, I)$ of an optimal offline algorithm O that knows the entire knowledge of the jobs in an execution instance I in advance. We say that an online scheduler A is ***c*-competitive**, if $f(A, I) \leq cf(O, I) + b$, where b is a fixed constant.

For different objectives, such as maximum weighted flow time and average flow time, there exist strong lower bounds showing that no schedule can achieve constant competitiveness on multiprocessor systems. Therefore, previous work has considered a **resource augmentation** analysis where the algorithm is given extra speed over the optimal scheduler [93]. We say that a scheduler A is ***s*-speed *c*-competitive**, if A is c-competitive when executed on a s-speed machine while the optimal schedule is on a unit speed machine. In particular, if a scheduler is $(1+\epsilon)$-speed $O(f(\epsilon))$-competitive for any $\epsilon > 0$ where $f(\epsilon)$ is some function that only depends on ϵ, we say that it is **scalable** and this is the best positive result one can show for problems that have strong lower bounds on the competitive ratio.

Part I

Static Real-Time Systems for Parallel Tasks with Deadlines

Introduction

This part of the thesis focuses on the parallel real-time scheduling problems for tasks in static real-time systems. In these systems, the arrival patterns and computational demands of jobs are known in advance and modeled as reoccurring real-time tasks; and the temporal correctness (i.e., schedulability) of tasks must be guaranteed prior to the execution.

Chapter 3 and Chapter 4 consider the problem of scheduling parallel real-time tasks with implicit deadlines on a uniform multicore machine. Tasks have hard real-time constraints and are characterized using their worst-case work and critical-path length parameters. Chapter 3 analyzes two well known algorithms, namely global earliest-deadline-first and global rate-monotonic. A PGEDF platform is implemented for supporting global earliest-deadline-first scheduling service for parallel programs written in the widely used OpenMP language. Chapter 4 proposes a new algorithm, namely federated scheduling, which is a generalization of partitioned scheduling to parallel tasks. Federated scheduling is proven to have the best theoretical bounds and is implemented in RTCG platform. Empirical results also show that federated scheduling generally has better schedulability in practice.

Chapter 5 focuses mixed-criticality system that comprises safety-critical and non-safety-critical tasks sharing a computational platform. Thus, different levels of assurance are required by different tasks in terms of real-time performance. In addition, as the computational demands of real-time tasks increases, tasks may require internal parallelism in order to complete within stringent deadlines. In this chapter, we consider the problem of mixed-criticality scheduling of parallel real-time tasks and propose a novel mixed-criticality federated scheduling (MCFS) algorithm, which is based on federated intuition for scheduling parallel real-time tasks. It strategically assigns cores and virtual deadlines to tasks, in order to achieve good schedulability and performance bounds. An implementation of an MCFS runtime system in

Linux that supports parallel programs written in OpenMP is presented, which demonstrates the practicality of the MCFS approach.

Chapter 6 considers the soft real-time performance of federated scheduling and address average-case workloads instead of worst-case ones. In particular, this chapter considers stochastic tasks — tasks for which execution time and critical-path length are random variables. In this context, bounded expected tardiness is used as the schedulability criterion and stochastic capacity augmentation bound is defined as performance bound. We present three federated mapping algorithms with different complexities for core allocation and different schedulability performance. All of them guarantee bounded expected tardiness and provide the same bound of 2.

Chapter 7 addresses the challenge of scaling up parallel real-time computations on a large number of cores. Although randomized work stealing has been adopted as a highly scalable scheduling approach for general-purpose computing, it may seem unsuitable for providing real-time guarantees due to the non-predictable nature of random stealing. Surprisingly, this chapter via experiments with benchmark programs shows that random work stealing delivers tighter distributions in task execution times than a centralized greedy scheduler. To support scalable soft real-time computing, a Real-Time Work-Stealing platform (RTWS) is presented. RTWS employs the federated scheduling algorithm in Chapter 6 to allocate cores to multiple parallel tasks offline, while leveraging the work stealing scheduler to schedule each task on its dedicated cores online. Experimental results show that RTWS outperforms RTCG on a 32-core system for soft real-time tasks.

Contents of this part of the thesis have appeared in the following publications:

- J. Li, K. Agrawal, C. Lu and C. Gill, Analysis of Global EDF for Parallel Tasks, Euromicro Conference on Real-Time Systems (ECRTS'13), July 2013. Outstanding Paper Award.

- D. Ferry, J. Li, M. Mahadevan, K. Agrawal, C.D. Gill and C. Lu, A Real-Time Scheduling Service for Parallel Tasks, IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'13), April 2013.
- A. Saifullah, J. Li, K. Agrawal, C. Lu and C.D. Gill, Multi-core Real-Time Scheduling for Generalized Parallel Task Models, Real-Time Systems (RTS), Issue 4, pages 404-435, July 2013.
- J. Li, J.-J. Chen, K. Agrawal, C. Lu, C. Gill and A. Saifullah, Analysis of Federated and Global Scheduling for Parallel Real-Time Tasks, Euromicro Conference on Real-Time Systems (ECRTS'14), July 2014.
- J. Li, K. Agrawal, C. Gill and C. Lu, Federated Scheduling for Stochastic Parallel Real-time Tasks, IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'14), August 2014.
- J. Li, Z. Luo, D. Ferry, K. Agrawal, C. Gill and C. Lu, Global EDF Scheduling for Parallel Real-Time Tasks, Real-Time Systems (RTS), 51(4): 395-439, July 2015.
- J. Li, D. Ferry, S. Ahuja, K. Agrawal, C. Gill and C. Lu, Mixed-Criticality Federated Scheduling for Parallel Real-Time Tasks, IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'16), April 2016. Outstanding Paper Award.
- J. Li, S. Dinh, K. Kieselbach, K. Agrawal, C. Gill and C. Lu, Randomized Work Stealing for Large Scale Soft Real-time Systems, IEEE Real-Time Systems Symposium (RTSS'16), December 2016.
- J. Li, D. Ferry, S. Ahuja, K. Agrawal, C. Gill and C. Lu, Mixed-Criticality Federated Scheduling for Parallel Real-Time Tasks, Real-Time Systems (RTS), 1-52, June 2017.

Chapter 3

Global Scheduling for Parallel Real-Time Tasks

In this chapter, we consider the problem of scheduling parallel real-time tasks with implicit deadlines on a uniform multicore machine. Tasks have hard real-time constraints and are characterized using their worst-case work and critical-path length parameters, which is formally defined in Section 2.4.

In this setting, we consider *global policies* in which a parallel task is allowed to use all the available cores in the system and can be migrated among these cores. Specifically, we analyze two well known global scheduling strategies: namely global earliest-deadline-first (global EDF or GEDF) and global rate-monotonic (global RM or GRM). In GEDF, a job has higher priority than another if its absolute deadline is earlier. In GRM, a task has higher priority than another task if its periods is shorter.

We show that GEDF and GRM provide strong performance guarantees, in the form of *capacity augmentation bounds*, for scheduling parallel DAG tasks with implicit deadlines. In particular, we show that if on unit-speed cores, a task set has total utilization of at most m and the critical-path length of each task is smaller than its deadline, then GEDF can schedule it with speed $\frac{3+\sqrt{5}}{2} \approx 2.618$; and GRM can schedule it with speed $2 + \sqrt{3} \approx 3.732$. We also provide lower bounds showing that the bounds are tight for GEDF when m is sufficiently large. These are the best known capacity augmentation bound for parallel DAG tasks under any global strategy and any fixed priority scheduling, respectively.

The rest of this chapter is organized as follows. Section 3.1 presents a survey on hard real-time scheduling for sequential and parallel tasks. Section 3.2 introduces canonical DAG, which is utilized to prove the capacity augmentation bounds for GEDF and GRM in Section 3.3 and 3.4, respectively.

3.1 Related Work on Hard Real-Time Systems

Real-time scheduling for sequential tasks: Most prior work on real-time scheduling atop multiprocessors has concentrated on sequential tasks [54]. In this context, many sufficient schedulability tests for GEDF and other global fixed priority scheduling algorithms have been proposed [5, 10, 14, 17, 29, 30, 76, 105, 144]. In particular, for implicit deadline hard real-time tasks, the best known utilization bound is $\approx 50\%$ using partitioned fixed priority scheduling [7] or partitioned EDF [19, 117]; this trivially implies a capacity bound of 2. [19] proved that global EDF has a capacity augmentation bound of $2 - 1/m$ for sequential tasks on multiprocessors.

Earlier work considering intra-task parallelism makes strong assumptions on task models [52, 106, 121]. For more realistic parallel tasks, e.g. synchronous tasks, Kato et al. [95] proposed a gang scheduling approach.

Decomposition-based scheduling for parallel real-time tasks: Most earlier approaches for scheduling synchronous tasks involve task decomposition. Specifically, a parallel task is decomposed into a set of independent sequential tasks with sub-release time and sub-deadline. Therefore, the dependency relationships are broken considering that subtasks in same task are separated by different release offset. In the meanwhile, dependencies are sustained because sub-release time and sub-deadline from subtasks with dependency relationships do not overlap. After the task decomposition, subtasks are scheduled using existing strategies for scheduling sequential tasks on multiprocessors, such as deadline monotonic [72] or GEDF [17]. A maximum density of a task is bounded after the decomposition or transformation and hence guarantees a capacity augmentation bound.

For a restricted set of synchronous tasks, Lakshmanan et al. [103] proved a capacity augmentation bound of 3.42 using deadline monotonic scheduling for decomposed tasks. For more general synchronous tasks, Saifullah et al. [138] provided a different decomposition strategy for general parallel synchronous tasks and proved a capacity augmentation bound of 4 for GEDF and 5 for partitioned deadline monotonic scheduling. The decomposition strategy was improved in [123] for using less cores. For the same general synchronous model, the best known augmentation bound is 3.73 [96] also using decomposition. In the respective papers, these results are stated as resource augmentation bounds, but they are in fact the stronger capacity augmentation bounds.

The decomposition approach in [138] was later extended to general DAGs [137] to achieve a capacity augmentation bound of 4 under GEDF on decomposed tasks (note that in that work GEDF is used to schedule sequential decomposed tasks, not parallel tasks directly). This is the best augmentation bound known for task sets with multiple DAGs.

In general, decomposition-based strategies require explicit knowledge of the structure of the DAG off-line in order to apply decomposition. In contrast, non-decomposition based strategies, the program can unfold dynamically since no off-line knowledge is required.

Non-decomposition scheduling for parallel real-time tasks: For non-decomposition strategies, researchers have studied primarily global earliest deadline first (GEDF) and global rate-monotonic (GRM). Andersson and Niz [6] show that GEDF provides resource augmentation bound of 2 for synchronous tasks with constrained deadlines. [51] and [9] presented schedulability tests for GEDF and partitioned fixed priority scheduling respectively. Baruah et al. [23] proved that when the task set is a *single DAG task* with arbitrary deadlines, GEDF provides a resource augmentation bound of 2.

For multiple DAGs under GEDF, [34] and [112] independently proved the same resource augmentation bound $2 - \frac{1}{m}$ using different proving techniques, which extended the resource augmentation bound of $2 - \frac{1}{m}$ for sequential multiprocessor scheduling result from [130]. In

[34], they also proved that global deadline monotonic scheduling has a resource augmentation bound of $3 - \frac{1}{m}$, and also present polynomial time and pseudo-polynomial time schedulability tests for DAGs with arbitrary-deadlines.

There has been some result for other scheduling strategies and different real-time constraints. Nogueira et al. [124] explored the use of work-stealing for real-time scheduling. The paper is mostly experimental and focused on soft real-time performance. The bounds for hard real-time scheduling only guarantee that tasks meet deadlines if their utilization is smaller than 1. Liu and Anderson [114] analyzed the response time of GEDF without decomposition for soft real-time tasks.

Real-time platforms: Various platforms support sequential real-time tasks on multi-core machines [36, 43, 108]. LITMUS$^{\text{RT}}$ [36] is a straightforward implementation of GEDF scheduling with usability, stability and predictability. The SCHED_DEADLINE [108] is another comparable GEDF patch to Linux and has been submitted to mainline Linux. A more recent work, G-EDF-MP [43] uses massage passing instead of locking and has better scalability than the previous implementations. Our platform prototype, PGEDF, is implemented using LITMUS$^{\text{RT}}$ as the underlying GEDF scheduler. Our goal is to simply to illustrate the feasibility of GEDF for parallel tasks. We speculate that if the underlying GEDF scheduler implementation is replaced with one that has lower overhead, the overall performance of PGEDF will also improve.

Recently, scalability of global scheduling was addressed with the use of message passing to communicate global scheduling decision [44]. Before that Brandenburg et al., [37] empirically studied the scalability of several scheduling algorithms for multiprocessors.

As for parallel tasks, there are two systems [71, 96] that support parallel real-time tasks based on different decomposition strategies. Kim et al. [96] used a reservation-based OS to implement a system that can run parallel real-time programs for an autonomous vehicle application, demonstrating that parallelism can enhance performance for complex tasks. Ferry et al. [71] developed a parallel real-time scheduling service on standard Linux. However, since both systems adopted task decomposition approaches, they require users to provide exact task structures and subtask execution time details in order to decompose tasks correctly. The system presented [71] also requires modifications to the compiler and runtime system to decompose, dispatch and execute parallel applications. The platform prototype presented here does not require decomposition or such detailed information.

For platforms without task decomposition, [151] presents a platform supporting partitioned fixed-priority scheduling for parallel synchronous tasks on a special COMPOSITE operating system with significantly lower parallel overhead.

Ferry et al. [70] studied how to exploit parallelism in real-time hybrid structural simulations (RTHS) to improve real-time performance. The resulted parallelized RTHS program was executed and scheduled by our RTCG prototype in Chapter 4. Experiments on RTHS in [70] thus illustrates how parallel real-time scheduling can practically help to improve performance in cyber-physical systems.

3.2 Canonical Form of a DAG Task

In this section, we introduce the concept of a DAG's canonical form. Note each task can have an arbitrarily complex DAG structure which may be difficult to analyze and may not even be known before runtime. However, given the known task set parameters (work, critical path length, utilization, etc.) we represent each task using a canonical DAG that allows us to upper bound the demand of the task in any given interval length t. These results will play an important role when we analyze the capacity augmentation bounds for GEDF in Section 3.3 and GRM in Section 3.4.

For analytical purposes, instead of considering the complex DAG structure of individual tasks τ_i, we consider a **canonical form** τ_i^* of task τ_i. The canonical form of a task is represented by a simpler DAG. In particular, each subtask (node) of task τ_i^* has execution time ϵ, which is positive and arbitrarily small. Note that ϵ is a hypothetical unit-node execution time. Therefore, it is safe to assume that $\frac{D_i}{\epsilon}$ and $\frac{C_i}{\epsilon}$ are both integers.

Recall that we classify each task τ_i as a *low-utilization* if $u_i = C_i/D_i < 1$ (and hence $C_i < D_i$); or *high-utilization* task, if τ_i's utilization $u_i \geq 1$. Low and high-utilization tasks have different canonical forms described below.

- The canonical form τ_i^* of a low-utilization task τ_i is simply a chain of C_i/ϵ nodes, each with execution time ϵ. Note that task τ_i^* is a sequential task.

- The canonical form τ_i^* of a high-utilization task τ_i starts with a chain of $D_i/\epsilon - 1$ nodes each with execution time ϵ. The total work of this chain is $D_i - \epsilon$. The last node of the chain *forks* all the remaining nodes. Hence, all the remaining $(C_i - D_i + \epsilon)/\epsilon$ nodes have an edge from the last node of this chain. Therefore, all these nodes can execute entirely in parallel.

Figure 3.1 provides an example for such a transformation for a high-utilization task. It is important to note that the canonical form τ_i^* does not depend on the DAG structure of τ_i at all. It depends only on the task parameters of τ_i.

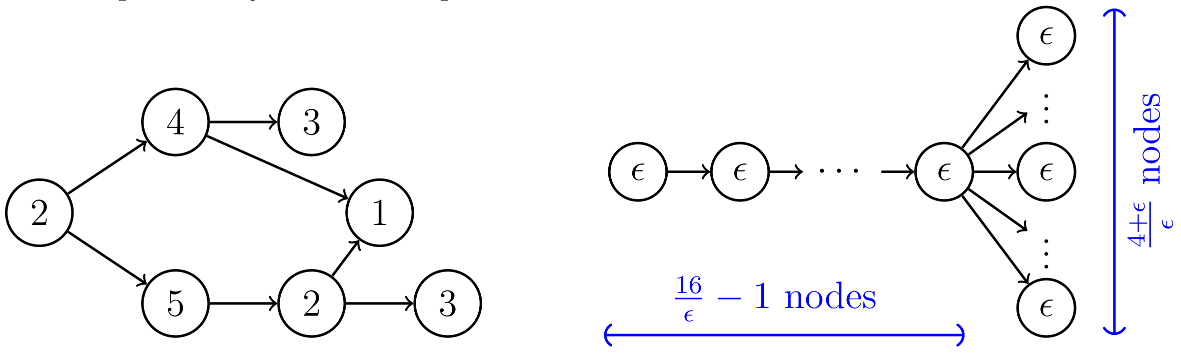

2
4
3
1
5
2
3
(a) original DAG
ε
ε
· · ·
ε
ε
ε
ε
$\frac{16}{\epsilon} - 1$ nodes
$\frac{4+\epsilon}{\epsilon}$ nodes
(b) canonical form: heavy task

Figure 3.1: A high-utilization DAG task τ_i with $L_i = 12$, $C_i = 20$, $T_i = D_i = 16$, and $u_i = 1.25$ and its canonical form, where the number in each node is its execution time.

As an additional analysis tool, we define a hypothetical scheduling strategy $\mathcal{S}_\infty$ that schedules a task set τ on an infinite number of cores, that is, $m = \infty$. With infinite number of cores, the prioritization of the sub-jobs becomes unnecessary and $\mathcal{S}$ can obtain an optimal schedule by simply assigning a sub-job to a core as soon as that sub-job becomes ready for execution. Using this schedule, all the tasks finish within their critical-path length; therefore, if $L_i \leq D_i$ for all tasks τ_i in τ, the task set always meets the deadlines. We denote this schedule as S_∞. Similarly, $S_{\infty,\alpha}$ is the resulting schedule when $\mathcal{A}_\infty$ schedules tasks on cores of speed $\alpha \geq 1$. Note that $S_{\infty,\alpha}$ finishes a job of task τ_i exactly L_i/α time units after it is released.

We now define some notations based on $S_{\infty,\alpha}$. Let $q_i(t,\alpha)$ be the total work finished by $S_{\infty,\alpha}$ between the arrival time r_i of task τ_i and time $r_i + t$. Therefore, in the interval from $r_i + t$ to $r_i + D_i$ (interval of length $D_i - t$) the remaining $C_i - q_i(t,\alpha)$ workload has to be finished.

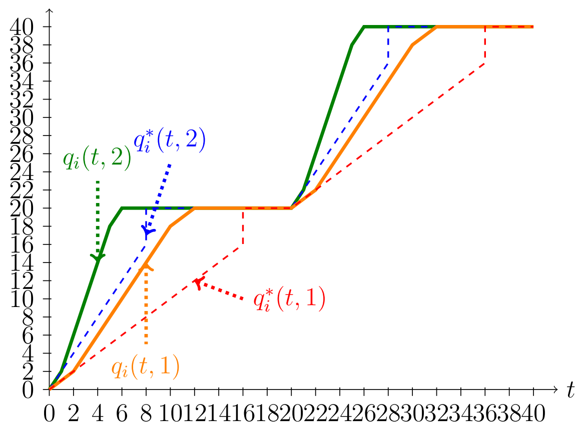
We define *maximum load*, denoted by $work_i(t,\alpha)$, for task i as the maximum amount of work (computation) that $S_{\infty,\alpha}$ must do on the sub-jobs of τ_i in any interval of length t. We can derive $work_i(t,\alpha)$ as follows:

$$work_i(t,\alpha) = \begin{cases} C_i - q_i(D_i - t, \alpha) & t \leq D_i \\ \left\lfloor \frac{t}{D_i} \right\rfloor C_i + work_i(t - \left\lfloor \frac{t}{D_i} \right\rfloor D_i, \alpha) & t > D_i. \end{cases} \tag{3.1}$$

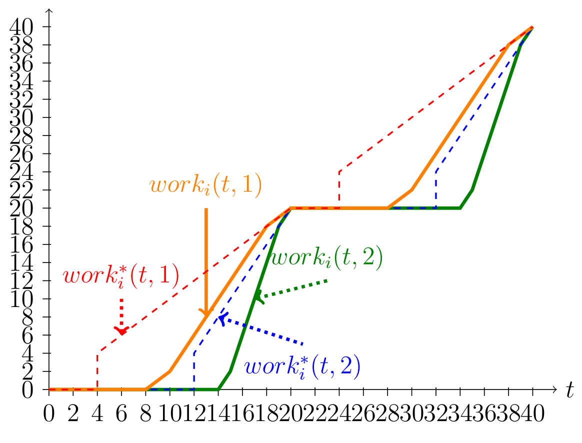
Clearly, both $q_i(t,\alpha)$ and $work_i(t,\alpha)$ for a task depend on the structure of the DAG.

We similarly define $q_i^*(t,\alpha)$ for the canonical form τ_i^*. As the canonical form in task τ_i^* is well-defined, we can derive $q_i^*(t,\alpha)$ directly. Note that ϵ can be arbitrarily small, and, hence, its impact is ignored when calculating $q_i^*(t,\alpha)$.

We can now define the canonical maximum load $work_i^*(t,\alpha)$ as the maximum workload of the canonical task τ_i^* in any interval t in schedule $S_{\infty,\alpha}$. For a low-utilization task τ_i,



(a) $q_i^*(t,\alpha)$ and $q_i(t,\alpha)$



(b) $work_i^*(t,\alpha)$ and $work_i(t,\alpha)$

Figure 3.2: $q_i^*(t,\alpha)$, $q_i(t,\alpha)$, $work_i^*(t,\alpha)$ and $work_i(t,\alpha)$ for the high-utilization task τ_i in Figure 3.1.

where $C_i/D_i < 1$, and τ_i^* is a chain, it is easy to see that the canonical workload is

$$work_i^*(t,\alpha) = \begin{cases} 0 & t < D_i - \frac{C_i}{\alpha} \\ \alpha \cdot (t - (D_i - \frac{C_i}{\alpha})) & D_i - \frac{C_i}{\alpha} \le t \le D_i \\ \left\lfloor \frac{t}{D_i} \right\rfloor C_i + work_i^*(t - \left\lfloor \frac{t}{D_i} \right\rfloor D_i, \alpha) & t > D_i. \end{cases} \tag{3.2}$$

Similarly, for high-utilization tasks, where $C_i/D_i \ge 1$, when ϵ is arbitrarily small, we have

$$work_i^*(t,\alpha) = \begin{cases} 0 & t < D_i - \frac{D_i}{\alpha} \\ C_i - D_i + \alpha \cdot (t - (D_i - \frac{D_i}{\alpha})) & D_i - \frac{D_i}{\alpha} \le t \le D_i \\ \left\lfloor \frac{t}{D_i} \right\rfloor C_i + work_i^*(t - \left\lfloor \frac{t}{D_i} \right\rfloor D_i, \alpha) & t > D_i. \end{cases} \tag{3.3}$$

Figure 3.2 shows the $q_i^*(t,\alpha)$, $q_i(t,\alpha)$, $work_i^*(t,\alpha)$, and $work_i(t,\alpha)$ of the high-utilization task τ_i in Figure 3.1 when $D_i = 16$, $\alpha = 1$, and $\alpha = 2$. Note that $work_i^*(t,\alpha) \ge work_i(t,\alpha)$. In fact, the following lemma proves that $work_i^*(t,\alpha) \ge work_i(t,\alpha)$ for any $t > 0$ and $\alpha \ge 1$.

Lemma 1 *For any $t > 0$ and $\alpha \geq 1$, $work_i^*(t, \alpha) \geq work_i(t, \alpha)$.*

Proof. For low-utilization tasks, the entire work C_i is sequential. When $t < \frac{C_i}{\alpha}$, $q_i^*(t, \alpha)$ is αt, so $q_i(t, \alpha) \geq \alpha t = q_i^*(t, \alpha)$. When $\frac{C_i}{\alpha} \leq t < D_i$, $q_i(t, \alpha) = C_i = q_i^*(t, \alpha)$.

Similarly, for high-utilization tasks, the first D_i units of work is sequential, so when $t < \frac{D_i}{\alpha}$, $q_i^*(t, \alpha) = \alpha t$. In addition, $S_{\infty,\alpha}$ finishes τ_i exactly $\frac{L_i}{\alpha}$ time units after it is released, while it finishes the τ_i^* at $\frac{D_i}{\alpha}$. Since the critical-path length $L_i \leq D_i$ for all τ_i and τ_i^* at unit-speed system, when $t < \frac{L_i}{\alpha}$, $q_i(t, \alpha) \geq \alpha t = q_i^*(t, \alpha)$. When $\frac{L_i}{\alpha} \leq t < \frac{D_i}{\alpha}$, $q_i(t, \alpha) = q_i^*(\frac{D_i}{\alpha}, \alpha) > q_i^*(t, \alpha)$, When $L_i < t \leq D_i$, Lastly, when $\frac{D_i}{\alpha} \leq t < D_i$, $q_i(t, \alpha) = C_i = q_i^*(t, \alpha)$

We can conclude that $q_i^*(t) \leq q_i(t)$ for any $0 \leq t < D_i$. Combining with the definition of $work(t, \alpha)$ (Equation (3.1)), we complete the proof. □

We classify task τ_i as a *light* or *heavy* task. A task is a light task if $u_i = C_i/D_i < \alpha$. Otherwise, we say that τ_i is heavy $(u_i \geq \alpha)$. The following lemmas provide an upper bound on the density (the ratio of workload that has to be finished to the interval length) for heavy and light tasks.

Lemma 2 *For any task τ_i, $t > 0$ and $1 < \alpha$, we have*

$$\frac{work_i(t, \alpha)}{t} \leq \frac{work_i^*(t, \alpha)}{t} \leq \begin{cases} u_i \ (0 \leq u_i < \alpha) \\ \frac{u_i - 1}{1 - \frac{1}{\alpha}} \quad (\alpha \leq u_i) \end{cases} \tag{3.4}$$

Proof. The first inequality in Inequality (3.4) comes from Lemma 1. We now show that the second inequality also holds for any task. Note that the right hand side is positive, since $1 > u_i > 0$. There are two cases:

Case 1: $0 < t \leq D_i$.

- If τ_i is a low-utilization task, where $work_i^*(t,\alpha)$ is defined in Equation (3.2). For any $0 < t \leq D_i$, we have

$$\begin{aligned} work_i^*(t,\alpha) - \frac{C_i}{D_i}\cdot t &\leq \alpha(t - D_i + \frac{C_i}{\alpha}) - \frac{C_i}{D_i}\cdot t \\ &= (t-D_i)(\alpha - \frac{C_i}{D_i}) \leq 0 \end{aligned}$$

  where we rely on assumptions: (a) $t \leq D_i$; (b) since τ_i is a low-utilization task, $\frac{C_i}{D_i} < 1$; and (c) $\alpha > 1$. Then, $\frac{work_i^*(t,\alpha)}{t} \leq \frac{C_i}{D_i} = u_i$.

- If τ_i is a high-utilization task, where $work_i^*(t,\alpha)$ is defined in Equation (3.3). Inequality (3.4) holds trivially when $0 < t < D_i - \frac{D_i}{\alpha}$, since the left side is 0 and right side is positive. For $D_i - \frac{D_i}{\alpha} \leq t \leq D_i$, we have

$$\frac{work_i^*(t,\alpha)}{t} = \frac{C_i + \alpha t - \alpha D_i}{t} = \alpha + D_i(\frac{u_i - \alpha}{t}),$$

  Therefore, $\frac{work_i^*(t,\alpha)}{t}$ is maximized either (a) when $t = D_i - \frac{D_i}{\alpha}$ if $u_i - \alpha \geq 0$ or (b) $t = D_i$ if $u_i - \alpha < 0$.

  If τ_i is a light task with $1 \leq u_i < \alpha$ and hence (b) is true, then we have $\frac{work_i^*(D_i,\alpha)}{D_i} = u_i$.

  If heavy task τ_i with $\alpha \leq u_i$ and hence (a) is true, then

$$\frac{work_i^*(D_i - \frac{D_i}{\alpha},\alpha)}{D_i - \frac{D_i}{\alpha}} = \frac{C_i - D_i}{D_i - \frac{D_i}{\alpha}} = \frac{u_i - 1}{1 - \frac{1}{\alpha}}$$

Therefore, Inequality (3.4) holds for $0 < t \leq D_i$.

Case 2: $t > D_i$ — Suppose that t is $kD_i + t'$, where k is $\left\lfloor \frac{t}{D_i} \right\rfloor$ and $0 < t' \leq D_i$. When $u_i < \alpha$, by Equation (3.2) and Equation (3.3), we have

$$\frac{work_i^*(t,\alpha)}{t} = \frac{kC_i + work_i^*(t',\alpha)}{kD_i + t'} \leq \frac{ku_iD_i + u_it'}{kD_i + t'} = u_i$$

When $\alpha \le u_i$, we can derive that $u_i \le \frac{u_i-1}{1-\frac{1}{\alpha}}$. By Equation (3.3), we have

$$\frac{work_i^*(t,\alpha)}{t} = \frac{kC_i + work_i^*(t',\alpha)}{kD_i + t'} \le \frac{ku_iD_i + \frac{u_i-1}{1-\frac{1}{\alpha}}t'}{kD_i+t'}$$
$$\le \frac{k\frac{u_i-1}{1-\frac{1}{\alpha}}D_i + \frac{u_i-1}{1-\frac{1}{\alpha}}t'}{kD_i+t'} \le \frac{u_i-1}{1-\frac{1}{\alpha}}$$

Hence, Inequality (3.4) holds for any task and any $t > 0$. □

We denote τ_{L} and τ_{H} as the set of light and heavy tasks in a task set, respectively; $\|\tau_{\mathrm{H}}\|$ as the number of heavy tasks in the task set; and total utilization of light and heavy tasks as $U_{\mathrm{L}} = \sum_{\tau_{\mathrm{L}}} u_i$ and $U_{\mathrm{H}} = \sum_{\tau_{\mathrm{H}}} u_i$, respectively.

Lemma 3 *For any task set, the following inequality holds:*

$$W = (\textstyle\sum_{i=1}^{n} work_i(t,\alpha)) \le \left(\frac{\alpha\cdot U_{\sum} - U_L - \alpha\cdot\|\tau_H\|}{\alpha-1}\right)\cdot t \le \left(\frac{\alpha m-\alpha}{\alpha-1}\right)\cdot t \tag{3.5}$$

Proof. By Lemma 2, for any $\alpha > 1$, it is clear that

$$\frac{W}{t} = \textstyle\sum_{\tau_{\mathrm{L}}+\tau_{\mathrm{H}}} \sup_{t>0} \frac{work_i^*(t,\alpha)}{t} \le \sum_{\tau_{\mathrm{L}}} u_i + \frac{\sum_{\tau_{\mathrm{H}}}(u_i-1)}{1-\frac{1}{\alpha}}$$
$$= \frac{\alpha\cdot\sum_{\tau_{\mathrm{L}}} u_i - \sum_{\tau_{\mathrm{L}}} u_i + \alpha\cdot\sum_{\tau_{\mathrm{H}}} u_i - \alpha\cdot\sum_{\tau_{\mathrm{H}}} 1}{\alpha-1} = \frac{\alpha\cdot U_{\sum} - U_{\mathrm{L}} - \alpha\cdot\|\tau_{\mathrm{H}}\|}{\alpha-1}$$

where sup is the supremum of a set of numbers, τ_{L} and τ_{H} are the sets of heavy tasks ($u_i \ge \alpha$) and light tasks ($u_i < \alpha$), respectively.

Note that $\sum_{\tau_{\mathrm{L}}} u_i + \sum_{\tau_{\mathrm{H}}} u_i = U_{\mathrm{L}} + U_{\mathrm{H}} = U_{\sum} \le m$. Since for $\tau_i \in \tau_{\mathrm{H}}$, $u_i \ge \alpha$, $U_{\mathrm{H}} = \sum_{\tau_{\mathrm{H}}} u_i \ge \|\tau_{\mathrm{H}}\|\alpha$, we can derive the following upper bound:

$$U_{\sum} - U_{\mathrm{L}} - \|\tau_{\mathrm{H}}\|\alpha \le \sup_{\tau}(U_{\sum} - U_{\mathrm{L}} - \|\tau_{\mathrm{H}}\|\alpha) = U_{\sum} - \alpha$$

This is because for any task set, there are two cases:

- If $\|\tau_{\mathrm{H}}\| = 0$ and hence $U_{\sum} = U_{\mathrm{L}}$, then $U_{\sum} - U_{\mathrm{L}} - \|\tau_{\mathrm{H}}\|\alpha = 0$.

- If $\|\tau_{\mathrm{H}}\| \geq 1$, then $U_{\sum} - U_{\mathrm{L}} = U_{\mathrm{H}} \leq U_{\sum}$ and $\|\tau_{\mathrm{H}}\|\alpha \geq \alpha$. Therefore, $U_{\sum} - U_{\mathrm{L}} - \|\tau_{\mathrm{H}}\|\alpha \leq U_{\sum} - \alpha$

Together with the definition of $U_{\sum}$ and $\sum_{\tau_{\mathrm{H}}} 1 = \|\tau_{\mathrm{H}}\|$,

$$W \leq \left(\frac{\alpha \cdot U_{\sum} - U_{\mathrm{L}} - \alpha \cdot \|\tau_{\mathrm{H}}\|}{\alpha - 1}\right) t \leq \left(\frac{\alpha \cdot U_{\sum} - \alpha}{\alpha - 1}\right) t \leq \left(\frac{\alpha m - \alpha}{\alpha - 1}\right) t$$

which proves the Inequalities 3.5 and 3.5 of Lemma 3. □

We use Lemma 3 in Sections 3.3 and 3.4 to derive bounds on GEDF and GRM scheduling.

3.3 Capacity Augmentation Bound of Global EDF

In this section, we state the capacity augmentation bound of $(3+\sqrt{5})/2$ for GEDF scheduling of parallel DAG tasks, using the canonical DAG results from Section 3.2. In addition, we also show a matching lower bound when $m \geq 3$.

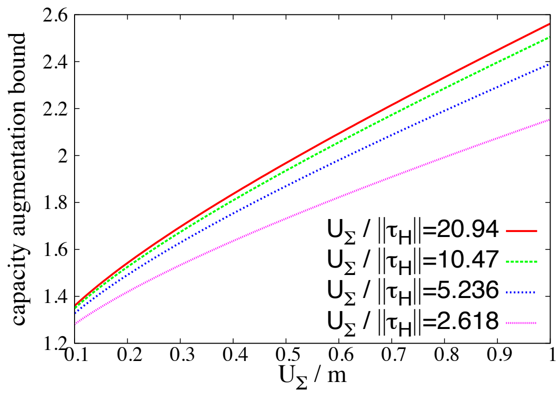
3.3.1 Upper Bound on Capacity Augmentation of GEDF

Our analysis builds on the analysis used to prove the resource augmentation bounds by Bonifaci et al. [34]. We first review the lemma that we will use to achieve our bound.

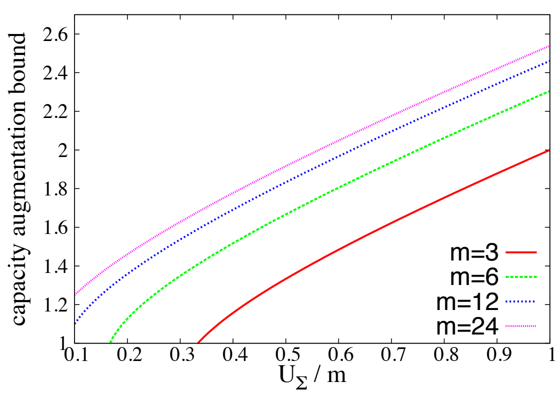
Lemma 4 *If $\forall t > 0$, $(\alpha m - m + 1) \cdot t \geq \sum_{i=1}^{n} work_i(t, \alpha)$, the task set is schedulable by GEDF on speed-α cores.*

Proof. This is based on a reformulation of Lemma 3 and Definition 10 in [34] considering cores with speed α. □

Theorem 5 *The capacity augmentation bound for GEDF is $\frac{3-\frac{2}{m}+\sqrt{5-\frac{8}{m}+\frac{4}{m^2}}}{2}$ ($\approx \frac{3+\sqrt{5}}{2}$, when m is large).*



(a) The required speedup of GEDF when m is sufficiently large and $U_{\rm L} = 0$ (i.e. $U_{\sum} = U_{\rm H}$).



(b) The required speedup of GEDF when $\|\tau_{\rm H}\| = 1$ and $U_{\rm L} = 0$ (i.e. $U_{\sum} = U_{\rm H}$).

Figure 3.3: The required speedup of GEDF under different settings.

Proof. From Lemma 3 Inequality (3.5), we have $\forall t > 0$, $\sum_{i=1}^{n} work_i(t, \alpha) \leq (\frac{\alpha m - \alpha}{\alpha - 1}) \cdot t$. If $\frac{\alpha \cdot m - \alpha}{\alpha - 1} \leq (\alpha m - m + 1)$, by Lemma 4 the schedulability test for G-EDF holds. To calculate α, we solve the derived equivalent inequality

$$m\alpha^2 - (3m - 2)\alpha + (m - 1) \geq 0.$$

which solves to $\alpha = \left(3 - \frac{2}{m} + \sqrt{5 - \frac{8}{m} + \frac{4}{m^2}}\right)/2.$ □

Using the above lemma and the canonical form of tasks, we are able to prove the following theorem for the capacity augmentation bound of GEDF and its generalized corollary relating the more precise bound using more information of a task set.

Corollary 6 *If a task set has total utilization $U_{\sum}$, the total heavy task utilization U_H and the number of heavy task $\|\tau_H\|$, then this task set will be schedulable under GEDF on a m-core machine with speed* $\alpha = \frac{2 + \frac{U_{\sum} - \|\tau_H\| - 1}{m} + \sqrt{\frac{4(U_H - \|\tau_H\|)}{m} + \frac{(U_{\sum} - \|\tau_H\| - 1)^2}{m^2}}}{2}$.

Proof. The proof is the same as in the proof of Theorem 5, but without using the Inequality (3.5). Instead, we directly use Inequality (3.5). If $\frac{\alpha \cdot U_{\sum} - U_{\rm L} - \alpha \cdot \|\tau_{\rm H}\|}{\alpha - 1} \leq (\alpha m - m + 1)$, by Lemma 4 the schedulability test for G-EDF holds for this task set. Solving this, we can get the required speedup α for the schedulability of the task set.

However, note that heavy tasks are defined as the set of all tasks τ_i with utilization $u_i \geq \alpha$. Therefore, given a task set, to accurately calculate α, we start with the upper bound on α, which is $\hat{\alpha} = \left(3 - \frac{2}{m} + \sqrt{5 - \frac{8}{m} + \frac{4}{m^2}}\right)/2$; then for each iteration i, we can calculate the required speedup α^i by using the U_{H}^{i-1} and $\|\tau_{\mathrm{H}}\|^{i-1}$ from the $(i-1)$-th iteration; we iteratively classify more tasks into the set of heavy tasks and we stop when no more tasks can be added to this set, i.e., $\|\tau_{\mathrm{H}}\|^{i-1} = \|\tau_{\mathrm{H}}\|^{i}$. Through these iterative steps, we can calculate an accurate speedup. $\square$

Figure 3.3(a) illustrates the required speedup of GEDF provided in Corollary 6 when m is sufficiently large (i.e., $m = \infty$ and $1/m = 0$) and $U_{\mathrm{L}} = 0$ (i.e. $U_{\sum} = U_{\mathrm{H}}$). We vary $\frac{U_{\sum}}{m}$ and $\frac{U_{\sum}}{\|\tau_{\mathrm{H}}\|}$. Note that $\frac{U_{\sum}}{\|\tau_{\mathrm{H}}\|} = \frac{U_{\mathrm{H}}}{\|\tau_{\mathrm{H}}\|}$ is the average utilization of all heavy tasks, which should be no less than $(3+\sqrt{5})/2 \approx 2.618$. It can be also be seen that the bound is getting closer to $(3+\sqrt{5})/2$, when $\frac{U_{\sum}}{\|\tau_{\mathrm{H}}\|}$ is larger, which results from $\|\tau_{\mathrm{H}}\| = 1$ and $m = \infty$.

Figure 3.3(b) illustrates the required speedup of GEDF provided in Corollary 6 when $\|\tau_{\mathrm{H}}\| \leq 1$ and $U_{\mathrm{L}} = 0$ (i.e. $U_{\sum} = U_{\mathrm{H}}$) with varying m. Note that $\alpha > 1$ is required by the proof of Corollary 6. And $\|\tau_{\mathrm{H}}\| = 1$ can only be true, if $\frac{U_{\sum}}{m} \geq \frac{\alpha}{m}$ (i.e. $\frac{U_{\sum}}{m} \geq 1/3$ for $m = 3$).

3.3.2 Lower Bound on Capacity Augmentation of GEDF

We now provide a lower bound for the capacity augmentation bound for small m.

Consider a task set τ with two tasks, τ_1 and τ_2. Task τ_1 starts with sequential execution for $1 - \epsilon$ time and then *forks* $\frac{m-2}{\epsilon} + 1$ subtasks with execution time ϵ. Here, we assume ϵ is an arbitrarily small positive number and hence it is safe to assume that $\frac{1}{\epsilon m}$ is a positive integer. Therefore, the total work of task τ_1 is $C_1 = m - 1$ and its critical-path length is $L_i = 1$. The minimum inter-arrival time of τ_1 is 1.

Task τ_2 is simply a sequential task with work (execution time) of $1 - \frac{1}{\alpha}$ and minimum inter-arrival time also $1 - \frac{1}{\alpha}$, where $\alpha > 1$ will be defined later. Clearly, the total utilization is m and the critical-path length of each task is at most the relative deadline (minimum inter-arrival time). By considering the schedulability of this task set under GEDF, we are able to prove the lower bound.

Lemma 7 *When* $\alpha < \frac{3-\frac{2}{m}-\delta+\sqrt{5-\frac{12}{m}+\frac{4}{m^2}}}{2}$ *and* $\delta = 2\epsilon + g(\epsilon)$ *and* $m \geq 3$*, then* $\frac{1-2\epsilon}{\alpha} + \frac{m-2}{m\alpha} > 1 - \frac{1-\frac{1}{\alpha}}{\alpha}$ *holds.*

Proof. By solving $\frac{1-2\epsilon}{\alpha} + \frac{m-2}{m\alpha} = 1 - \frac{1-\frac{1}{\alpha}}{\alpha}$, we know that the equality holds when

$$\alpha < \frac{3-\frac{2}{m}-2\epsilon+\sqrt{5-\frac{12}{m}+\frac{4}{m^2}}-g(\epsilon)}{2}$$

where $g(\epsilon)$ is a positive function of $\frac{1}{\epsilon}$, which approaches to 0 when ϵ approaches 0. Now, by setting δ to $2\epsilon + g(\epsilon)$, we reach the conclusion. □

Theorem 8 *The capacity augmentation bound for GEDF is at least* $\frac{3-\frac{2}{m}+\sqrt{5-\frac{12}{m}+\frac{4}{m^2}}}{2}$ *when* $\epsilon \to^{+} 0$.

Proof. Consider the system with two tasks τ_1 and τ_2 defined in the beginning of Section 3.3.2. Suppose that the arrival of task τ_1 is at time 0, and the arrival of task τ_2 is at time $\frac{1}{\alpha} + \frac{\epsilon}{\alpha}$. By definition, the first jobs of τ_1 and τ_2 have absolute deadlines at 1 and $1 + \frac{\epsilon}{\alpha}$. Hence, G-EDF will execute the sequential execution of task τ_1 and the sub-jobs of task τ_1 first, and then execute τ_2.

The finishing time of τ_1 at speed α is not earlier than $\frac{1-\epsilon}{\alpha} + \frac{\frac{m-2}{\epsilon}\epsilon}{m\alpha} = \frac{1-\epsilon}{\alpha} + \frac{m-2}{m\alpha}$. Hence, the finishing time of task τ_2 is not earlier than $\frac{1-\epsilon}{\alpha} + \frac{m-2}{m\alpha} + \frac{1-\frac{1}{\alpha}}{\alpha}$.

If $\frac{1-\epsilon}{\alpha} + \frac{m-2}{m\alpha} + \frac{1-\frac{1}{\alpha}}{\alpha} > 1 + \frac{\epsilon}{\alpha}$, then task τ_2 misses its deadline. By Lemma 7, we reach the conclusion. □

Figure 3.4 illustrates the upper bound of GEDF provided in Theorem 5 and the lower bound in Theorem 8 with respect to the capacity augmentation bound. It can be easily seen that the upper and lower bounds are getting closer when m is larger. When m is 100, the gap between the upper and the lower bounds is roughly about 0.00452.

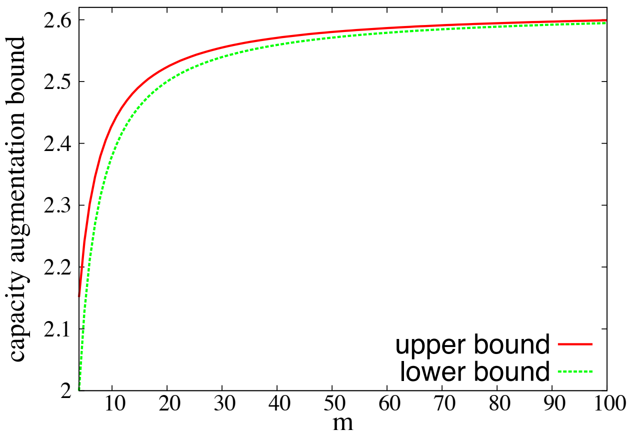


Figure 3.4: The upper bound of GEDF provided in Theorem 5 and the lower bound in Theorem 8 with respect to the capacity augmentation bound.

It is important to note that the more precise speedup in Corollary 6 is tight even for small m. This is because in the above example task set, $U_{\sum} = m$, total high task utilization $U_{\mathrm{H}} = m - 1$ and number of heavy task $\|\tau_{\mathrm{H}}\| = 1$, then according to the Corollary, the capacity augmentation bound for this task set under GEDF is

$$\frac{2+\frac{U_{\sum}-\|\tau_{\mathrm{H}}\|-1}{m}+\sqrt{\frac{4(U_{\mathrm{H}}-\|\tau_{\mathrm{H}}\|)}{m}+\frac{(U_{\sum}-\|\tau_{\mathrm{H}}\|-1)^2}{m^2}}}{2} = \frac{2+\frac{m-1-1}{m}+\sqrt{\frac{4(m-1-1)}{m}+\frac{(m-1-1)^2}{m^2}}}{2} = \frac{3-\frac{2}{m}+\sqrt{5-\frac{12}{m}+\frac{4}{m^2}}}{2}$$

which is exactly the lower bound in Theorem 8 for $\epsilon \rightarrow^{+} 0$.

3.4 Capacity Augmentation of Global RM

This section shows that GRM provides a capacity augmentation bound of $2+\sqrt{3}$ for large m. The structure of the proof is very similar to the analysis in Section 3.3. Again, we use a lemma from [34], restated below, to prove Theorem 10.

Lemma 9 *If* $\forall t > 0,\ 0.5(\alpha \cdot m - m + 1)t \geq \sum_i work_i(t, \alpha)$ *the task set is schedulable by GRM on speed-α cores.*

Proof. This is based on a reformulation of Lemma 6 and Definition 10 in [34]. Note that the analysis in [34] is for deadline-monotonic scheduling, by giving a sub-job of a task higher priority if its relative deadline is shorter. As we consider tasks with implicit deadlines, deadline-monotonic scheduling is the same as rate-monotonic scheduling. □

Proofs like those in Section 3.3.1 give us the bounds below for G-RM scheduling.

Theorem 10 *The capacity augmentation bound for GRM is* $\frac{4-\frac{3}{m}+\sqrt{12-\frac{20}{m}+\frac{9}{m^2}}}{2}$ *($\approx 2+\sqrt{3}$, when m is large).*

Proof. (Similar to the proof of Theorem 5:) First, we know from Lemma 3 that $\forall t > 0$, $\sum_{i=1}^{n} work_i(t,\alpha) \leq (\frac{\alpha m-\alpha}{\alpha-1})t$; Second, if $\frac{\alpha\cdot m-\alpha}{\alpha-1} \leq 0.5(\alpha m - m + 1)$. we can also conclude that the schedulability test for G-RM in Lemma 9 holds. By solving the inequality above, we have $\alpha \geq \frac{4-\frac{3}{m}+\sqrt{12-\frac{20}{m}+\frac{9}{m^2}}}{2}$, and prove Theorem 10. □

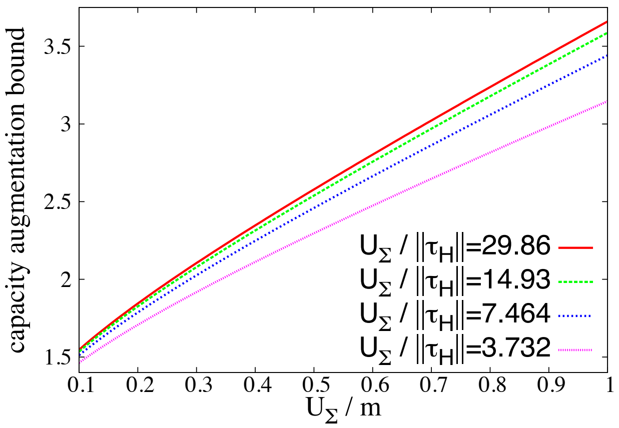


Figure 3.5: The required speedup of GRM when m is sufficiently large and $U_{\mathbf{L}} = 0$ (i.e. $U_{\sum} = U_{\mathbf{H}}$).

The result in Theorem 10 is the best known result for the capacity augmentation bound for global fixed-priority scheduling for general DAG tasks with arbitrary structures. Interestingly, Kim et al. [96] get the same bound of $2+\sqrt{3}$ for global fixed-priority scheduling of parallel synchronous tasks (a subset of DAG tasks).

The strategy used in [96] is quite different. In their algorithm, the tasks undergo a stretch transformation which generates a set of sequential subtask (each with its release time and deadline) for each parallel task in the original task set. These subtasks are then scheduled using a GDM scheduling algorithm. Note that even though the parallel tasks in the original task set have implicit deadlines, the transformed sequential tasks have only constrained deadlines — hence the need for deadline monotonic scheduling instead of rate monotonic scheduling.

Corollary 11 *If a task set has total utilization $U_{\sum}$, the total high task utilization U_H and the number of heavy task $\|\tau_H\|$, then this task set will be schedulable under GRM on a m-core machine with speed* $\alpha = \frac{2+\frac{2U_{\sum}-2\|\tau_H\|-1}{m}+\sqrt{\frac{8(U_H-\|\tau_H\|)}{m}+\frac{(2U_{\sum}-2\|\tau_H\|-1)^2}{m^2}}}{2}$.

Proof. The proof is the same as in the proof of Corollary 6, except that instead of using Lemma 4, it uses Lemma 9. □

Figure 3.5 illustrates the required speedup of GRM provided in Corollary 11 when m is sufficiently large (i.e., $m = \infty$ and $1/m = 0$) and $U_{\mathrm{L}} = 0$ (i.e. $U_{\sum} = U_{\mathrm{H}}$). We vary $\frac{U_{\sum}}{m}$ and $\frac{U_{\sum}}{\|\tau_{\mathrm{H}}\|}$. Again, $\frac{U_{\sum}}{\|\tau_{\mathrm{H}}\|}$ is the average utilization of all heavy tasks. It can be also be seen that the bound is getting closer to $2 + \sqrt{3}$, when $\frac{U_{\sum}}{\|\tau_{\mathrm{H}}\|}$ is larger, which results from $\|\tau_{\mathrm{H}}\| = 1$ and $m = \infty$.

3.5 Parallel GEDF Platform

To demonstrate the feasibility of parallel GEDF scheduling, we implemented a simple prototype platform called **PGEDF** by combining GNU OpenMP runtime system and the LITMUS$^{\mathrm{RT}}$ system. PGEDF is a straightforward implementation based on these off-the-shelf systems and simply sets appropriate parameters for both OpenMP and LITMUS$^{\mathrm{RT}}$ without modifying either. It is also easy to use this platform; the user can write tasks as programs with standard OpenMP directives and compile them using the g++ compiler. In addition, the user provides a task-set configuration file that specifies the tasks in the task-set and their deadlines. To validate the theory we present, PGEDF is configured for CPU intensive workloads and cache or I/O effects are beyond the scope this thesis.

Note that our goal in implementing PGEDF as a prototype platform is to show that GEDF is a good scheduler for parallel real-time tasks. This platform uses the GEDF plug-in of LITMUS^{RT} to execute the tasks. Our experimental results show that this PGEDF implementation has better performances than other two existing platforms for parallel real-time tasks in most cases. Recent work [43] has shown that using massage passing instead of coarse-grain locking (used in LITMUS^{RT}) the overhead of GEDF scheduler can be significantly lowered. Therefore, we potentially can get even better performance using G-EDF-MP as underlying operating system scheduler (instead of LITMUS^{RT}). However, improving the implementation and performance of PGEDF is beyond the scope of this chapter.

We first describe the relevant aspects of OpenMP and LITMUS^{RT} and then describe the specific settings that allow us to run parallel real-time tasks on this platforms. Then we present the API and implementation of PGEDF using OpenMP and LITMUS^{RT}. The empirical evaluations of PGEDF through synthetic task sets will be presented in Chapter 4, which are compared against an existing decomposition-based platform and a federated scheduling platform proposed in Chapter 4.

3.5.1 Background

We briefly introduce the GNU OpenMP runtime system and the LITMUS^{RT} patched Linux operating system, with an emphasis on the particular features that our PGEDF relies on in order to realize parallel GEDF scheduling.

OpenMP Overview

As introduced in Section 2.3, OpenMP is a specification for parallel programs that defines an open standard for parallel programming in C, C++, and Fortran [125]. It consists of a set of compiler directives, library routines and environment variables, which can influence the runtime behavior. Our PGEDF implementation uses a GNU implementation of OpenMP runtime system (**GOMP**), which is part of the GCC (GNU Compiler Collection). For our prototype of PGEDF, we only support parallel synchronous tasks. These tasks are described as a series of segments which can be parallel or sequential. A parallel segment is described as a parallel for-loop while a sequential segment consists of arbitrary sequential code. Therefore, we will restrict our attention to parallel for-loops.

We now briefly describe the OpenMP (specifically GOMP) runtime strategy for such programs. Under GOMP, each OpenMP program starts with a single thread, called the **master thread**. During execution, when the runtime system encounters the first parallel section of the program, the master thread will create a **thread pool** and assign that **team** of threads to the parallel region. The threads created by the master thread in the thread pool are called **worker threads**. The number of worker threads can be set by the user.

The master thread executes the sequential segments. In parallel segments (parallel for-loops), each iteration is considered a unit of work and **maps** (distributes) the work to the team of threads according to the chosen policies, as specified by arguments passed to the `omp_set_schedule()` function call. In OpenMP, there are three different kind of policies: **dynamic**, **static** and **guided** policies. In the **static 1** policy, all iterations are divided among the team of threads at the start of the loop, and iterations are distributed to threads one by one: each thread in the team will get one iteration at a time in a round robin manner.

Once a thread finishes all its assigned work from a particular parallel segment, it **waits** for all other threads in the team to finish before moving on to the next segment of the task. The waiting policy can be set by via the environment variable `OMP_WAIT_POLICY`. Using passive synchronization, waiting threads are blocked and put into the Linux sleep queue, where they do not consume CPU cycle while waiting. On the other hand, active synchronization would let waiting threads spin without yielding the processors, which would consume CPU cycles while waiting.

One important property of the GOMP, upon which our implementation relies, is that the team of threads for each program is **reusable**. After the execution of a parallel region, the threads in the team are not destroyed. Instead, all threads except the master thread wait for the next parallel segment, again according to the policy set by `OMP_WAIT_POLICY`. The master thread continues the sequential segment. When it encounters the next parallel segment GOMP runtime system detects that it already has a team of threads available to it, and simply reuses them for executing this segment, as before.

LITMUS^{RT} Overview

LITMUS^{RT} (Linux Testbed for Multiprocessor Scheduling in Real-Time Systems) is an algorithm-oriented real-time extension of Linux [36]. It focuses on multiprocessor real-time scheduling and synchronization. Many real-time schedulers, including global, clustered, partitioned and semi-partitioned schedulers are implemented as plug-ins for Linux. Users can use these schedulers for real-time tasks, and standard Linux scheduling for non-real-time tasks.

In LITMUS^{RT}, the GEDF implementation is meant for sequential tasks. A typical LITMUS^{RT} real-time program contains one or more **rt_tasks** (real-time tasks), which are released periodically. In fact, each `rt_task` can be regarded as a **rt_thread**, which is a standard Linux thread with real-time parameters. Under the GEDF scheduler, a `rt_task` can be suspended and migrated to a different CPU core according to the GEDF scheduling strategy. The platform consists of three main data structures to hold these tasks: a release queue, a one-to-one processor mapping, and a shared ready queue. The release queue is implemented as a priority queue with a clock tick handler, and is used to hold waiting-to-be-released jobs. The one-to-one processor mapping has the thread that corresponds to each job that is currently executing on each processor. The ready queue (shared by all processors) is implemented as a priority queue by using binomial heaps for fast queue-merge operations triggered by jobs with coinciding release times.

In order to run a sequential task as a real-time task under GEDF, LITMUS^{RT} provides an interface to configure a thread as an `rt_tasks`. The following steps must be taken to properly configure these [42]:

1. First, function `init_rt_thread()` is called to initialize the user-space real-time interface for the thread.
2. Then, the real-time parameters of the thread are set by calling `set_rt_task_param(getid(),&rt_task_param)`: the `getid()` function will return the actual thread ID in the system; the real-time parameters, including period, relative deadline, execution time and budget policy, are stored in the `rt_task_param` structure; these parameters will then be stored in the TCB (thread control block) using the unique thread ID and they will be validated by the kernel.
3. Finally, `task_mode(LITMUS_RT_TASK)` is called to start running the thread as a real-time task.

The periodic execution of jobs of `rt_tasks` is achieved by calling LITMUS^{RT} system calls as well. In particular, after each period, `sleep_next_period()` must be called to ask LITMUS^{RT} to move the thread from the run queue to the release queue. The thread sleeps in the release queue and the GEDF scheduler within the LITMUS^{RT} will automatically move it to the ready queue at its next absolute release time. The thread will eventually be automatically woken up and executed according to GEDF priority based on its absolute deadline.

3.5.2 PGEDF Programming Interface

Now we describe how our PGEDF platform integrates the GOMP runtime with GEDF scheduling in LITMUS^{RT} to execute parallel real-time tasks. The PGEDF platform provides two key features: parallel task execution and real-time GEDF scheduling. The GOMP

```
#include <omp.h>
#include "task.h"
int init(int argc, char *argv[]) {
        //Initialize the task
}
int run(int argc, char *argv[]) {
        //Arbitrary parallel code
}
int finalize(int argc, char *argv[]) {
        //Clean up after the task
}
task_t task = { init, run, finalize };
```

Figure 3.6: Task Program Format

runtime system is used to perform parallel execution of each task, while real-time execution and GEDF scheduling is realized by the LITMUS^{RT} GEDF plug-in.

Currently, PGEDF only supports synchronous task sets with implicit deadlines — tasks which consist of a sequence of segments and each segment is either a parallel segment (specified using a parallel-for loop) or a sequential segment (specified as regular code).

The task structure is shown in Figure 3.6. Tasks are C or C++ programs that include a header file (`task.h`) and conform to a simple structure: instead of a `main` function, a programmer specifies a `run` function, which is executed when a job of the task is invoked. Tasks can also specify optional `initialize` and `finalize` functions, each of which (if not null) will be called once, before the first and after the last call to the run function, respectively. These optional functions let tasks set up and clean up resources as needed.

Additionally, a configuration file must be specified for the task set, specifying runtime parameters (such as program name and arguments) and real-time parameters (such as worst-case execution time, critical-path length, and period) for each task in the task set. This separate specification makes it flexible and easy to maintain; e.g., we do not have to recompile tasks in order to change timing constraints. The configuration file format is shown in Figure 3.7.

```
SystemFirstCore  SystemLastCore
Task1ProgramName  Task1Arg1  Task1Arg2 ...
Task1: WorstCaseExecutionTime CriticalPathLength Period NumIterations
...
TasknProgramName  TasknArg1  TasknArg2 ...
Taskn: WorstCaseExecutionTime CriticalPathLength Period NumIterations
```

Figure 3.7: Format of the Configuration File

3.5.3 PGEDF Operation

Unlike sequential tasks where there is only one thread per `rt_task`, for parallel tasks there is a team of threads generated by OpenMP. Since all the threads in the team belong to the same task, we must set all their deadlines (periods) to be the same. In addition, we must make sure that all the threads of all the tasks are properly executed by the GEDF plug-in in LITMUS. We now describe how to set the parameters of both OpenMP and LITMUS to properly enforce this policy.

We first describe the specific parameter settings we have to use to ensure correct execution: (1) We specify the `static 1` policy within OpenMP to ensure that each thread gets approximately the same amount of work. (2) We also set the OpenMP thread synchronization policy to be passive. As discussed in Chapter 3.5.1, PGEDF cannot allow spinning waiting of threads. By using blocking synchronization, once a worker thread finishes its job, it will go to sleep immediately and yield the processor to threads from other tasks. Then the GEDF scheduler will assign the available core to the thread in the front of the prioritized ready queue. Thus, the idle time of one task can be utilized by other tasks, which is consistent with GEDF scheduling theory. (3) For each task, we set the number of threads to be equal to the number of available cores, m, using the GOMP function call *omp_set_num_threads(m)*. This means that if there are n tasks in the system, we will have a total of mn threads in the system. (4) In LITMUS$^{\text{RT}}$, the `budget_policy` is set equal to

NO_ENFORCEMENT and the execution time of a thread is set to be the same as the relative deadline, as we do not need bandwidth control.

In addition to this parameter settings, PGEDF also does additional work to ensure that all the task parameters are set correctly. In particular, the actual code that is executed by PGEDF for each task is shown in Figure 3.8. In this code, before we run the task's actual run function for the first time, PGEDF performs some additional initialization in the form of a parallel for-loop. In addition, after each periodic execution of the task's run function, PGEDF executes an additional for-loop.

```
#pragma omp parallel for schedule(static, 1)
for (unsigned i = 0; i < num_cores; i++)
        rt_thread(period, deadline);

for (unsigned j = 0; j < num_periods; j++)
{
        task.run(task_argc, task_argv);

        #pragma omp parallel for schedule(static, 1)
        for (unsigned i = 0; i < num_cores; i++)
                sleep_next_period();
}
```

Figure 3.8: Main Structure of Each Real-Time Task in PGEDF

Let us first look at the initial for-loop. This parallel for-loop is meant to set the proper real-time parameters for this task to be correctly scheduled by GEDF plug-in in LITMUS$^{\mathrm{RT}}$. We must set the real-time parameters for the entire team of OpenMP threads of this task. However, OpenMP threads are designed to be invisible to programmers, so we have no direct access to them. We get around this problem by using this initial for-loop, which has exactly m iterations — recall that each task has exactly m threads in its thread pool. Note that before this parallel for-loop, we set the scheduling policy to be **static 1** policy, which is a round robin static mapping between iterations and threads. Therefore, due to the `static 1` policy, each iteration is mapped to exactly 1 thread in the thread pool. Therefore, even though we cannot directly access OpenMP threads, we can still set real-time parameters for them inside the initial parallel for-loop by calling `rt_thread(period, deadline)` within this loop. This function is defined within the PGEDF platform to perform configuration for LITMUS$^{\mathrm{RT}}$. In particular, the configuration steps described in the itemized list in the previous section are performed by this function. Since the thread team is reused for all parallel regions of the same program, we only need to set the real-time parameters for it once during task initialization; we need not set it at each job invocation.

After initialization, each task is periodically executed by `task.run(task_argc, task_argv)`, inside which there could be multiple parallel for-loops executed by the same team of threads. Periodic execution is achieved by the parallel for-loop after the `task.run` function; after each job invocation, this loop ensures that `sleep_next_period()` is called by each thread in the thread pool. Note again that since the number of iterations in this parallel for-loop is m, each thread will get exactly one iteration ensuring that each thread calls this function. This last for-loop is similar to the initialization for-loop, but tells the system that all the threads in the team of this task have finished their work and that the system should only wake them up when next period begins.

We can now briefly argue that these settings guarantee the correct GEDF execution. After we appropriately set the real-time parameters, all the relative deadlines will be automatically converted to absolute deadlines when scheduled by the LITMUS$^{\mathrm{RT}}$. Since each thread in the same team of a particular task has the same deadline, all threads of this task have the same priority. Also, threads of a task with an earlier deadline have higher priority than the threads of the task with later deadlines — this is guaranteed by LITMUS$^{\mathrm{RT}}$ GEDF plug-in. Since the number of threads allocated to each program is equal to the number of cores, as required by GEDF, each job can utilize the entire machine when it is the highest priority task and has enough parallelism. If it does not have enough parallelism, then some of its threads sleep and yield the machine to the job with the next highest priority. Therefore, the GEDF scheduler within the LITMUS^{RT} enforces the correct priorities using the ready queue.

Chapter 4

Federated Scheduling for Parallel Real-Time Tasks

In this chapter, we present a novel **federated scheduling** paradigm for parallel real-time tasks. Again, task sets have hard real-time constraints and worst-case work and critical-path length are used to determine their schedulability (defined in Section 2.4). We prove that federated scheduling has a capacity augmentation bound of 2 for hard real-time scheduling. This is the best known capacity augmentation bound for parallel tasks and it approaches the lower bound of any scheduler when the number of cores is large.

In federated scheduling, each *high-utilization task* (utilization ≥ 1) is allocated a dedicated cluster (set) of cores and all the *low-utilization tasks* share the cluster composed of the remaining cores. The federated strategy works in two stages: Given a task set τ, a *mapping* algorithm either **admits** a task set and outputs a **core assignment** — which consists of a cluster for each high-utilization task and a final cluster for all low-utilization tasks — or declares that it can not guarantee schedulability of the task set. Therefore, a mapping algorithm is automatically a schedulability test. After the mapping is done, the scheduling is straightforward. A **greedy** (work-conserving) strategy is used to schedule high-utilization jobs on their dedicated clusters. Since low-utilization tasks can be run sequentially, a multiprocessor scheduling algorithm, such as global earliest deadline first (global EDF or GEDF), is used to schedule all low-utilization tasks on their shared cluster. Notably, this federated strategy does not require any task decomposition or transformation; therefore, the internal structure of the DAGs need not be known in advance in order to use this strategy.

We present the federated scheduling algorithm in Section 4.1, followed by the proof for its capacity augmentation bound of 2 in Section 4.2. Section 4.3 presents an example showing the lower bound on capacity augmentation of any scheduler for parallel tasks. In Section 4.4, we compare the efficiency of GEDF, GRM and federated scheduling from a practical perspective. Finally, we present a platform providing the federated scheduling service for OpenMP programs and show some preliminary evaluation results in Section 4.5.

4.1 Federated Scheduling Algorithm

Given a task set τ, the *federated scheduling algorithm* works as follows: First, tasks are divided into two disjoint sets: τ_{high} contains all *high-utilization tasks* — tasks with worst-case utilization at least one ($u_i \geq 1$), and τ_{low} contains all the remaining *low-utilization tasks*. Consider a high-utilization task τ_i with worst-case execution time C_i, worst-case critical-path length L_i, and deadline D_i (which is equal to its period T_i). We assign n_i dedicated cores to τ_i, where n_i is

$$n_i = \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil \tag{4.1}$$

We use $n_{\text{high}} = \sum_{\tau_i \in \tau_{\text{high}}} n_i$ to denote the total number of cores assigned to high-utilization tasks τ_{high}. We assign the remaining cores to all low-utilization tasks τ_{low}, denoted as $n_{\text{low}} = m - n_{\text{high}}$. The federated scheduling algorithm *admits* the task set τ, if n_{low} is non-negative and $n_{\text{low}} \geq 2 \sum_{\tau_i \in \tau_{\text{low}}} u_i$.

After a valid core allocation, *runtime scheduling* proceeds as follows: (1) Any greedy (work-conserving) parallel scheduler can be used to schedule a high-utilization task τ_i on its assigned n_i cores. Informally, a *greedy scheduler* is one that never keeps a core idle if some node is ready to execute. (2) Low-utilization tasks are treated and executed as though they are sequential tasks and any multiprocessor scheduling algorithm (such as partitioned EDF [117], or various rate-monotonic schedulers [7]) with a utilization bound of at most 1/2 can be used to schedule all the low-utilization tasks on the allocated n_{low} cores. The important observation is that we can safely treat low-utilization tasks as sequential tasks since $C_i \leq D_i$ and parallel execution is not required to meet their deadlines.[2]

4.2 Capacity Augmentation Bound of 2 for Federated Scheduling

Theorem 12 *The federated scheduling algorithm has a capacity augmentation bound of 2.*

To prove Theorem 12, we consider a task set τ that satisfies Conditions (2.1) and (2.2) from Definition 1 for b =2. Then, we (1) state the relatively obvious Lemma 13; (2) prove that a high utilization task τ_i meets its deadline when assigned n_i cores; and (3) show that n_{low} is non-negative and satisfies $n_{\text{low}} \geq b \sum_{\tau_i \in \tau_{\text{low}}} u_i$ and therefore all low utilization tasks in τ will meet deadlines when scheduled using any multiprocessor scheduling strategy with utilization bound no less than b (i.e. can afford total task set utilization of $m/b = 50\% m$). These three steps complete the proof.

Lemma 13 *A task set τ is classified into disjoint subsets $s_1, s_2, ..., s_k$, and each subset is assigned a dedicated cluster of cores with size $n_1, n_2, ..., n_k$ respectively, such that $\sum_i n_i \leq m$. If each subset s_j is schedulable on its n_j cores using some scheduling algorithm $\mathcal{S}_j$ (possibly different for each subset), then the whole task set is guaranteed to be schedulable on m cores.*

High-Utilization Tasks Are Schedulable. Assume that a machine's execution time is divided into discrete quanta called *steps*. During each step a core can be either idle or performing one unit of work. We say a step is *complete* if no core is idle during that step, and otherwise we say it is *incomplete*. A *greedy* scheduler never keeps a cores idle if there is ready work available. Then, for a greedy scheduler on n_i cores, we can state two straightforward lemmas.

[2]Even if these tasks are expressed as parallel programs, it is easy to enforce correct sequential execution of parallel tasks — any topological ordered execution of the nodes of the DAG is a valid sequential execution.

Lemma 14 *Consider a greedy scheduler running on n_i cores for t time steps. If the total number of incomplete steps during this period is t^*, the total work F^t done during these time steps is at least $F^t \geq n_i t - (n_i - 1)t^*$.*

Lemma 15 *If a job of task τ_i is executed by a greedy scheduler, then every incomplete step reduces the remaining critical-path length of the job by 1.*

From Lemmas 14 and 15, we can establish Theorem 16.

Theorem 16 *If an implicit-deadline deterministic parallel task τ_i is assigned $n_i = \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$ dedicated cores, then all its jobs can meet their deadlines, when using a greedy scheduler.*

Proof. For contradiction, assume that some job of a high-utilization task τ_i misses its deadline when scheduled on n_i cores by a greedy scheduler. Therefore, during the D_i time steps between the release of this job and its deadline, there are fewer than L_i incomplete steps; otherwise, by Lemma 15, the job would have completed. Therefore, by Lemma 14, the scheduler must have finished at least $n_i D_i - (n_i - 1)L_i$ work.

$$\begin{aligned} n_i D_i - (n_i - 1)L_i &= n_i(D_i - L_i) + L_i = \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil (D_i - L_i) + L_i \\ &\geq \frac{C_i - L_i}{D_i - L_i}(D_i - L_i) + L_i = C_i \end{aligned}$$

Since the job has work of at most C_i, it must have finished in D_i steps, leading to a contradiction. □

Low-Utilization Tasks are Schedulable. We first calculate a lower bound on n_{low}, the number of total cores assigned to low-utilization tasks, when a task set τ that satisfies Conditions (2.1) and (2.2) of Definition 1 for $b = 2$ is scheduled using a federated scheduling strategy.

Lemma 17 *The number of cores assigned to low-utilization tasks is $n_{low} \geq 2 \sum_{low} u_i$.*

Proof. Here, for brevity of the proof, we denote $\sigma_i = \frac{D_i}{L_i}$. It is obvious that $D_i = \sigma_i L_i$ and hence $C_i = D_i u_i = \sigma_i u_i L_i$. Therefore,

$$n_i = \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil = \left\lceil \frac{\sigma_i u_i L_i - L_i}{\sigma_i L_i - L_i} \right\rceil = \left\lceil \frac{\sigma_i u_i - 1}{\sigma_i - 1} \right\rceil$$

Since each task τ_i in task set τ satisfies Condition (2.2) of Definition 1 for $b = 2$; therefore, the critical-path length of each task is at most $1/b$ of its relative deadline, that is, $L_i \leq D_i/b \implies \sigma_i \geq b = 2$.

By the definition of high-utilization task τ_i, we have $1 \leq u_i$. Together with $\sigma_i \geq 2$, we know that:

$$0 \leq \frac{(u_i - 1)(\sigma_i - 2)}{\sigma_i - 1}$$

From the definition of ceiling, we can derive

$$\begin{aligned}
n_i &= \left\lceil \frac{\sigma_i u_i - 1}{\sigma_i - 1} \right\rceil < \frac{\sigma_i u_i - 1}{\sigma_i - 1} + 1 = \frac{\sigma_i u_i + \sigma_i - 2}{\sigma_i - 1} \\
&\leq \frac{\sigma_i u_i + \sigma_i - 2}{\sigma_i - 1} + \frac{(u_i - 1)(\sigma_i - 2)}{\sigma_i - 1} = \frac{\sigma_i u_i + \sigma_i - 2 + \sigma_i u_i - 2u_i - \sigma_i + 2}{\sigma_i - 1} \\
&= \frac{2\sigma_i u_i - 2u_i}{\sigma_i - 1} = \frac{2u_i(\sigma_i - 1)}{\sigma_i - 1} = 2u_i \\
&= bu_i
\end{aligned}$$

In summary, for each high-utilization task, $n_i < bu_i$. So, their sum τ_{high} satisfies $n_{\text{high}} = \sum_{\text{high}} n_i < b \sum_{\text{high}} u_i$. Since the task set also satisfies Condition (2.1), we have

$$n_{\text{low}} = m - n_{\text{high}} > b \sum_{\text{all}} u_i - b \sum_{\text{high}} u_i = b \sum_{\text{low}} u_i$$

Therefore, the number of remaining cores allocated to low-utilization tasks is at least $n_{\text{low}} > 2 \sum_{\text{low}} u_i$. □

Corollary 18 *For task sets satisfying Conditions (2.1) and (2.2), a multiprocessor scheduler with utilization bound of at least* 50% *can schedule all the low-utilization tasks sequentially on the remaining* n_{low} *cores.*

Proof. Low-utilization tasks are allocated n_{low} cores, and from Lemma 17 we know that the total utilization of the low utilization tasks is less than $n_{\text{low}}/b = 50\% n_{\text{low}}$. Therefore, any multiprocessor scheduling algorithm that provides a utilization bound of 2 (i.e., can schedule any task set with total worst-case utilization ratio no more than 50%) can schedule it. $\square$

Many multiprocessor scheduling algorithms (such as partitioned EDF or partitioned RM) provide a utilization bound of 1/2 (i.e., 50%) to sequential tasks. That is, given n_{low} cores, they can schedule any task set with a total worst-case utilization up to $n_{\text{low}}/2$. Using any of these algorithms for low-utilization tasks will guarantee that the federated algorithm meets all deadlines with capacity augmentation of 2.

Therefore, since we can successfully schedule both high and low-utilization tasks that satisfy Conditions (2.1) and (2.2), we have proven Theorem 12 (using Lemma 13).

As mentioned before, a capacity augmentation bound acts as a simple schedulability test. However, for federated scheduling, this test can be pessimistic, especially for tasks with high parallelism. Note, however, that the federated scheduling algorithm described in Section 4.1 can also be directly used as a (polynomial-time) schedulability test: given a task set, after assigning cores to each high-utilization task using this algorithm, if the remaining cores are sufficient for all low-utilization tasks, then the task set is schedulable and we can admit it without deadline misses. This schedulability test admits a strict superset of tasks admitted by the capacity augmentation bound test, and in practice, it may admits task sets with utilization greater than $m/2$.

4.3 Lower Bound on Capacity Augmentation of Any Scheduler for Parallel Tasks

On a system with m cores, consider a task set τ with a single task, τ_1, which starts with sequential execution for $1 - \epsilon$ time and then *forks* $\frac{m-1}{\epsilon} + 1$ subtasks with execution time ϵ. Here, we assume ϵ is an arbitrarily small positive number. Therefore, the total work of task τ_1 is $C_1 = m$ and its critical-path length $L_i = 1$. The deadline (and also minimum inter-arrival time) of τ_1 is 1.

Theorem 19 *The capacity augmentation bound for any scheduler for parallel tasks on m cores is at least $2 - \frac{1}{m}$, when $\epsilon \to^{+} 0$.*

Proof. Consider the system defined above. The finishing time of τ_1 by running at speed α is not earlier than $\frac{1-\epsilon}{\alpha} + \frac{\frac{m-1}{\epsilon}\epsilon}{m\alpha} = \frac{2}{\alpha} - \frac{1}{m\alpha} - \frac{\epsilon}{\alpha}$. If $\alpha > 2 - \frac{1}{m}$ and $\epsilon \to^{+} 0$, then $\frac{2}{\alpha} - \frac{1}{m\alpha} - \frac{\epsilon}{\alpha} > 1$, then task τ_1 misses its deadline. Therefore, we reach the conclusion. $\square$

Since Lemma 19 works for any scheduler for parallel tasks, we can conclude that the lower bound on capacity augmentation of federated scheduling is at least 2 , when m is sufficiently large. Since we have shown that the upper bound on capacity augmentation of federated scheduling is also 2, therefore, we have closed the gap between the lower and upper bound of federated scheduling for large m. Moreover, for sufficiently large m, federated scheduling has the best capacity augmentation bound, among all schedulers for parallel tasks.

4.4 Practical Considerations

As have shown in the previous sections, the capacity augmentation bound for federated scheduling, GEDF and GRM are 2, 2.618 and 3.732, respectively. In this section, we consider their run-time efficiency and efficacy from a practical perspective. We consider four dimensions — static vs. dynamic priorities, and global vs. partitioned scheduling, overheads due to scheduling and synchronization overheads, and work-conserving vs. not.

Practitioners have generally found it easier to implement fixed (task) priority schedulers (such as RM) than dynamic priority schedulers (such as EDF). Fixed priority schedulers have always been well-supported in almost all real-time operating systems. Recently, there has been efforts on efficient implementations of job-level dynamic priority (EDF and GEDF) schedulers for sequential tasks [36]. Federated scheduling does not require any priority assignment for high-utilization tasks (since they own their cores exclusively) and can use either fixed or dynamic priority for low-utilization tasks. Thus, it is relatively easier to implement GRM and federated scheduling.

For sequential tasks, in general, global scheduling may incur more overhead due to thread migration and the associated cache penalty, the extent of which depends on the cache architecture and the task sets. In particular, for parallel tasks, the overheads for global scheduling could be worse. For sequential tasks, preemptions and migrations only occur when a new job with higher priority is released. In contrast, for parallel tasks, a preemption and possibly migration could occur whenever a node in the DAG of a job with higher priority is enabled. Since nodes in a DAG often represent a fine-grained units of computation, the number of nodes in the task set can be larger than the number of tasks. Hence, we can expect a larger number of such events. Since federated scheduling is a generalization of partitioned scheduling to parallel tasks, it has advantages similar to partitioning. In fact, if we use a partitioned RM or partitioned EDF strategy for low-utilization tasks, there are only preemptions but no migration for low-utilization tasks. Meanwhile, federated scheduling only allocates the minimum number of dedicated cores to ensure the schedulability of each high-utilization task, so there is no preemptions for high-utilization tasks and the number of migrations is minimized. Hence, we expect that federated scheduling will have less overhead than global schedulers.

In addition, parallel runtime systems have additional parallel overheads, such as synchronization and scheduling overheads. These overheads (per task) usually are approximately linear in the number of cores allocated to each task. Under federated scheduling, a minimum number of cores is assigned. However, depending on the particular implementation, global scheduling may execute a task on all the cores in the system and may have higher overheads.

Finally, note that federated scheduling is not a greedy (work conserving) strategy for the entire task set, although it uses a greedy schedule for each individual task. In many real systems, the worst-case execution times are normally over-estimated. Under federated scheduling cores allocated to tasks with overestimated execution times may idle due to resource over-provisioning. In contrast, work-conserving strategies (such as GEDF and GRM) and can utilize available cores through thread migration dynamically.

4.5 Implementation of a Federated Scheduling Platform

In this section, we describe the design of a scheduling service based on the federated scheduling algorithm presented in Section 4.1 and how we integrated this service into Real-Time Centralized Greedy **RTCG** platform. RTCG has several benefits: (1) It separates the goals of efficient parallel performance and rigorous real-time execution. This separation of concerns allows programmers to re-purpose existing parallel applications to be run with real-time semantics with minimal modifications. (2) It allows the use of existing parallel languages and runtime systems (not designed for real-time programs) to explore the degree of real-time performance one can achieve without implementing an entirely new parallel runtime system. Therefore, we were able to evaluate the performance of the greedy centralized scheduler from

OpenMP for real-time task sets. (3) While RTCG does not explicitly consider cache overheads, the scheduling policy has an inherent advantage with respect to cache locality, since parallel tasks are allocated dedicated cores and never migrate.

Application Programming Interface (API): The RTCG API makes it easy to convert existing parallel programs into real-time programs, which is similar to that of PGEDF in Section 3.5.2. Tasks are C or C++ programs that include a header file (`task.h`) and conform to a simple structure: instead of a `main` function, a `run` function is specified, which is periodically executed when a job of the task is invoked. Tasks can also specify optional `initialize` and `finalize` functions. Each (if defined) will be called once, before and after the periodic run function, respectively. These optional functions let tasks set up and clean up resources as needed. In addition, a configuration file must be provided for the task set, specifying runtime parameters (including program name and arguments) and real-time parameters (including period, work and critical-path length) for each task.

Platform Structure and Operation: RTCG separates the functionalities of parallel scheduling and real-time scheduling. We use two components to enforce these two functionalities, an **real-time scheduler (RT-scheduler)** and a **parallel dispatcher (PL-dispatcher)**.

Specifically, the RT-scheduler provides the real-time performance of a task. Prior to execution, it reads tasks' real-time parameters from the configuration file and calculates a core assignment during offline. The main function (provided by RTCG) binds each task to its assigned cores (by changing the CPU affinity mask). This core assignment ensures that each task has sufficient number of dedicated cores to meet most of its deadline during execution. Moreover, because each parallel task is executed on dedicated cores and no other tasks can introduce CPU interference with it, the PL-dispatcher does not need to be deadline- or priority-aware.

During execution, the PL-dispatcher enforces the periodic invocation of each task and calls an individual GNU OpenMP runtime system to provide parallel execution of each task. Since there are multiple concurrent parallel runtime systems that are unaware of each other, we need to entirely isolate them from each other to minimize scheduling overheads and CPU interference. Therefore, for OpenMP we use *static* thread management to create exactly n_i threads to each task. In other words, there is only one worker thread per core and hence the worker assignment by PL-dispatcher is consistent with the core assignment of the RT-scheduler.

PL-dispatcher executes the (optional) user-specified `initialize` functions. All tasks in the system perform an initial synchronization, and then start execution at their relative release times by calling `run` for each invocation. After the periodic invocations, PL-dispatcher executes the (optional) `finalize` functions.

4.6 Empirical Comparison Between PGEDF and RTCG

In the following experiments, we make comparison of PGEDF in Chapter 3 and RTCG with another platform that we implemented for scheduling parallel real-time tasks, namely RT-OpenMP [71]. Note that all these platforms can schedule OpenMP programs with only parallel for-loops, but each provides a different scheduling strategy. Specifically, RT-OpenMP implements a decomposition-based rate-monotonic scheduling and PGEDF implements global EDF scheduling. We compare these systems with randomly generated synthetic workload with increasing total utilizations.

We now describe our empirical evaluation using randomly generated synthetic *synchronous* tasks in OpenMP. Experiments were conducted on a 16-core machine composed of two Intel Xeon E5-2687W processors. When running experiments, we reserved two cores for operating system services, leaving 14 experimental cores.

We compare these platforms in terms **task set deadline miss ratio** – we define a task set having deadline misses if (1) it cannot be scheduled under the given scheduling policy; or (2) it can be scheduled, but there exists at least one deadline miss among all executed jobs from all tasks in the task set. We find that under many conditions, RTCG provides better schedulability than the other two platforms.

We should note two things here about how we define this criterion in our experiments which may be a little unintuitive. First, even if a task set is theoretically schedulable, a deadline miss may occur due to system overheads. Second, when we say that a task set can not be scheduled, we *do not* mean that the task set fails the theoretical schedulability test of the scheduler. We try to run a task set even if the scheduling algorithm can not guarantee schedulability. For instance, the global EDF algorithm used in PGEDF only guarantees schedulability to task sets with utilization of about 40%, while we only say that the task set fails if it actually misses a deadline. However, if we can not find any way to run the task set using the respective scheduler, we also say the task set fails – which sometimes happens with RTCG and RT-OpenMP, but never with PGEDF.

4.6.1 Task Generation

We randomly generate task sets with synchronous tasks consisting of parallel for-loops. Each task has varying number of segments with varying lengths and numbers of iterations. We ran 4 categories of task sets (shown in Table 4.1), with *T7:LP:LS* using 7 cores and the rest using 14 cores. For each task, period (and deadline) D are chosen uniformly from the list {4ms, 8ms, 16ms, 32ms, 64ms, 128ms} to form harmonic task sets.

Name	Total #Cores	L/D	Avg. #iter	Avg. #Tasks per TaskSet
T7:LP:LS	7	100%	8	3.66
T14:LP:LS	14	100%	8	5.03
T14:HP:LS	14	100%	12	3.38
T14:HP:HS	14	50%	12	5.22

Table 4.1: Task Set Characteristics

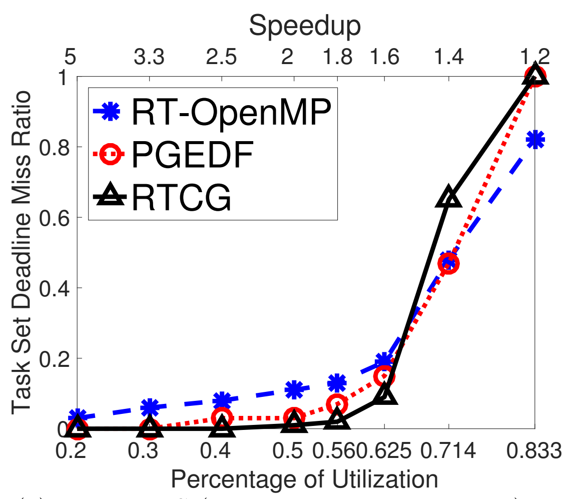
Tasks in Table 4.1 are different in two dimensions: (1) **low-parallelism** (LP) or **high-parallelism** (HP), controlled by the average number of iterations (Avg. #iter) in each parallel for-loop drawn from a log-normal distribution. (2) **low-slack** (LS) or **high-slack** (HS), controlled by the maximum ratio between its critical-path length and deadline (L/D). In general, low-slack tasks are more difficult to schedule. We do not present the combination of low-parallelism and high-slack because this is the easiest setting to schedule and all platforms have roughly the same performance.

For tasks, the maximum execution time of each iteration was chosen from a log-normal distribution with a mean of 700 micro-seconds. Segments were added to the task until adding another segment would make its critical-path length longer than the desired ratio (1/2 for high-slack tasks and 1 for low-slack tasks). Each task set starts empty and adds parallel tasks with utilization larger than 1, until the total utilization was between $m - 1$ and m (the number of cores in the machine). Then a last task with approximately the remaining utilization is added.

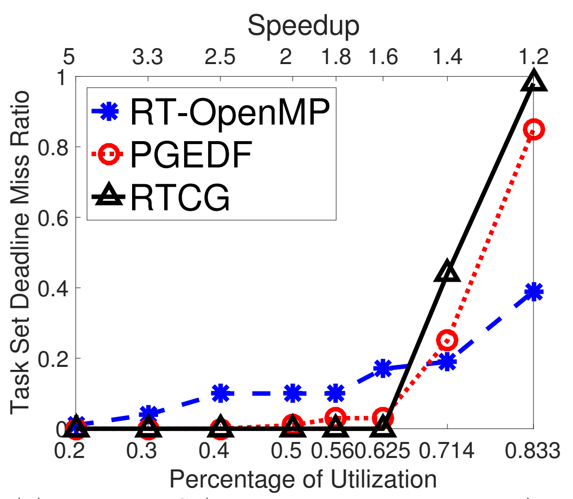
For each experiment setting, we first generated 100 task sets with total utilization approximately m, and then scaled down the execution time by the desired speedup. For example, for a speedup of 2, a iteration with execution time of 700 micro-seconds will be scaled down to 350 micro-seconds, and the total utilization ratio of the task set will be about 50%. In this manner, we achieve the desired total utilizations ratio {20%, 30%, 40%, 50%, 56%, 62.5%, 71.4%, 83.3%} of m, using speedup values {5, 3.3, 2.5, 2, 1.8, 1.6, 1.4, 1.2}.

4.6.2 Baseline Platform

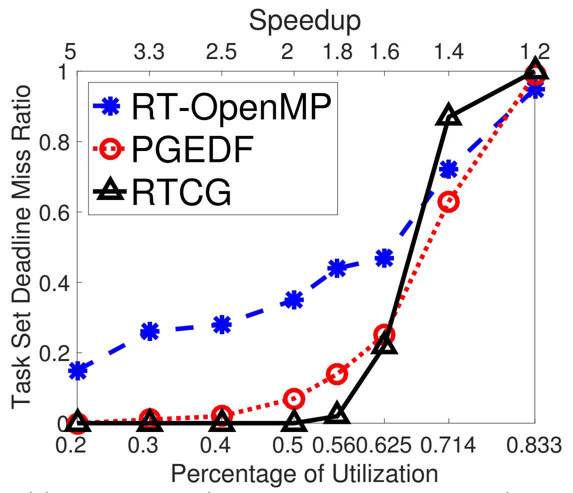
We compared the performance of PGEDF and RTCG with another platform that we implemented, RT-OpenMP from [71] — labeled **RT-OpenMP** — that can schedule parallel synchronous task sets on multicore system. RT-OpenMP is based on a task decomposition



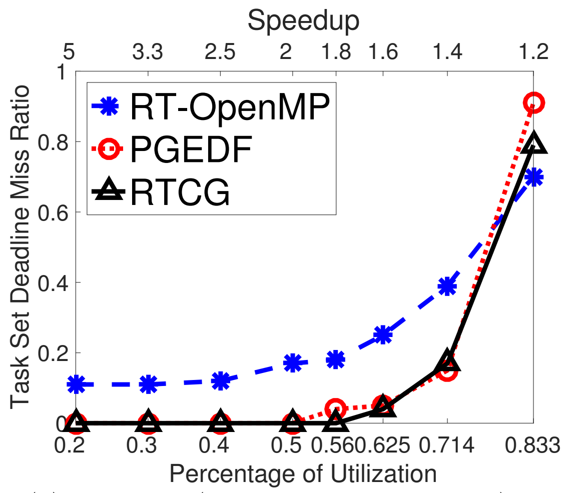
(a) T14:HP:LS (high-parallelism, low-slack).



(b) T14:HP:HS (high-parallelism, high-slack).



(c) T14:LP:LS (low-parallelism, low-slack).



(d) T7:LP:LS (low-parallelism, low-slack).

Figure 4.1: Task Set Deadline Miss ratio of RTCG vs. PGEDF vs. RT-OpenMP with different types of task sets with varying percentages of utilization on 14 and 7 cores.

scheduling strategy: parallel tasks are decomposed into sequential subtasks with intermediate release times and deadlines. These sequential tasks are scheduled using a partitioned deadline monotonic scheduling strategy [72]. This decomposition based scheduler was shown to guarantee a capacity augmentation of 5 [138]. In theory, any valid bin-packing strategy provides this augmentation bound. The original paper [71] compared a worst-fit and best-fit bin-packing strategy for partitioning and found that worst-fit always performed better. Therefore, we only compare PGEDF and RTCG vs. RT-OpenMP with worst-fit bin-packing.

4.6.3 Experimental Results

We can see that when the total utilization ratio is less than 70%, RTCG outperforms PGEDF and both are significantly better than RT-OpenMP. This is mainly because RT-OpenMP has much higher overheads than the other two and even PGEDF has higher overheads than RTCG. For instance, due to overheads, RT-OpenMP misses deadlines even at 20% utilization, even though all task sets with that utilization are guaranteed to be schedulable in theory using the scheduler RT-OpenMP uses. PGEDF also has several task sets miss deadline even when global EDF scheduler theoretically guarantees schedulability, again due to overheads. In contrast, RTCG has the minimum overheads, since there is no preemption or migration overheads for federated scheduling. In fact, we find that if the federated mapping algorithm admits a task set, the task set seldom misses a deadline — in all our experiments only 1 task out of thousands of task sets had any deadline misses and even this task only missed 10 deadlines out of thousands of deadlines.

We note, however, that the deadline miss ratio of RTCG is higher than PGEDF for very high utilization task sets (70% and 80%). This is due to the fact that at these utilizations, the main cause of task set deadline misses comes from the theoretical schedulability of the corresponding scheduling policy. Since federated scheduling is not work-conserving, with high workloads, it cannot generate a feasible core assignment, while global EDF can try to schedule a task set even if there is no guarantee in theory.

Figure 4.1(d) shows the task set level deadline miss ratio for the 7-core task set (low-parallelism, low-slack task sets on 7 cores). We can see that RTCG and PGEDF have comparable performance and they both outperform RT-OpenMP for almost all utilizations. For instance, RTCG starts to miss deadlines at 62.5%, and PGEDF starts to miss deadlines at 56%, while RT-OpenMP cannot schedule 10 task sets even at 20% utilization.

In conclusion, even though RTCG and PGEDF are not designed for scheduling task sets for providing hard real-time guarantees, under stress testing settings, they still provides (almost) hard real-time performance and outperforms the decomposition-based platform under many settings. In particular, the capacity augmentation bound of 2 and 2.618 for the federated scheduling and GEDF scheduler holds for all experiments conducted here, respectively.

Chapter 5

Mixed-Criticality Federated Scheduling

In this chapter, we study the problem of mixed-criticality scheduling of parallel real-time tasks. In **mixed-criticality** systems, tasks with different criticality levels share a computing platform and demand different levels of assurance in terms of real-time performance. For example, when an autonomous vehicle is in danger of an accident, crash-avoidance systems is more safety-critical than route planning or stability enhancing systems. On the other hand, in normal driving conditions all these features are essential and need to meet their deadlines to provide a smooth and stable drive, while infotainment systems only need to make the best effort.

Mixed-criticality model is an emerging paradigm for real-time systems, since it can significantly improve resource efficiency. Specifically, safety-critical tasks must be approved by some *certification authority (CA)* and their schedulability must be guaranteed under possibly pessimistic assumptions about task execution parameters, while system designers usually try to meet the deadlines of all tasks but with less stringent validation. Both the CA and system designers must make estimates of each task's worst-case execution time (work). Thus, a task τ_i may be characterized by two different worst-case work values: a pessimistic *overload work* C_i^O for certification and a less pessimistic *nominal work* C_i^N from empirical measurements. The goal of mixed-criticality scheduling is two-fold: (1) In the *typical-state* — when all tasks exhibit nominal behavior — all tasks must be schedulable. (2) In the *critical-state* — when some task exceeds its nominal work — all safety-critical tasks must still be schedulable, but we need not guarantee schedulability of other tasks.

Three related trends make it increasingly important to accelerate the convergence of mixed-criticality systems and parallel tasks: (1) rapid increases in the number of cores per chip; (2) increasing demand for consolidation and integration of functionality with different levels of criticality on shared multi-core platforms; and (3) increasing computational demands of individual tasks, which makes parallel execution essential to meet deadlines. Although there has been extensive research on the two related problems, namely, mixed-criticality scheduling of sequential tasks (see [39] for a survey); and single-criticality scheduling of parallel tasks [6, 23, 34, 51, 96, 103, 111, 112, 114, 123, 138]. To our knowledge, there has been almost no prior work on the combined problem of mixed-criticality scheduling of parallel tasks, except for [115].

Challenges of scheduling mixed-criticality parallel task sets: Incorporating parallelism presents novel challenges for scheduling. The DAG structure of a task may not be known in advance; in fact, each job of the same task may have a different DAG structure. Thus the schedulability analysis and scheduler cannot rely on tasks' structural information. Moreover, tasks may have utilization much smaller than 1 in the typical-state, but much larger than 1 in the critical-state. To deal with this dramatic change, the scheduler must be able to detect the overload behavior early and increase the resource allotment so that the task can still meet its deadline.

In this chapter, we propose a *mixed-criticality federated scheduling (MCFS)* algorithm and prove its capacity augmentation bound under various conditions. MCFS generalizes federated scheduling in Chapter 4 to mixed-criticality systems. Federated scheduling (and thus MCFS) has the advantage of not requiring task decomposition. Likewise, the scheduler not need to know the internal structure of the tasks, a priori. The work and critical-path length estimates of a task give an abstraction of the DAG. Moreover, they can be empirically measured without knowing the specific structure of all the instances of the task. We assume that task sets are *implicit-deadline sporadic task sets*.

Section 5.3 provides details of the MCFS algorithm and schedulability test for dual-criticality systems where all tasks are *high-utilization tasks* — that is, all tasks have either nominal or overload utilization larger than 1. We consider only high-utilization tasks, since parallelism is essential for them to meet their deadlines. For this dual-criticality system with high-utilization tasks, we prove the correctness of MCFS and also prove that it has a capacity augmentation bound of $2 + \sqrt{2} \approx 3.41$ in Section 5.4. To our knowledge, this is the first known augmentation bound for parallel mixed-criticality tasks.

We demonstrate the applicability of MCFS by implementing a MCFS runtime system in Linux that supports OpenMP parallel programs (Section 5.8). We conduct empirical evaluations to show the practicality of MCFS (Section 5.9). Empirical evaluation shows that the MCFS runtime system not only delivers mixed-criticality scheduling for parallel tasks, but also supports graceful degradation; that is, if a high-criticality task enters its overload state, the system need not immediately discard all low-criticality tasks. Instead, it will gradually discard low-criticality tasks as needed.

5.1 System Model and Background

5.1.1 Mixed-Criticality Parallel Real-Time Tasks Model

Now we formally define the mixed-criticality parallel real-time task model. Each job (instance of a parallel task) can be modeled as a dynamically unfolding *directed acyclic graph (DAG)*, in which each node represents a sequence of instructions and each edge represents a precedence constraint between nodes. A node is *ready* to be executed when all its predecessors have been executed. Note that each job of a task could be a different DAG — it may be completely different structurally. For each job J_i of τ_i, we consider two parameters: (1) the total work (execution time) $\mathcal{C}_i$ of job J_i is the sum of execution times of all nodes in job J_i's DAG; and (2) the critical-path length $\mathcal{L}_i$ of job J_i is the length of the longest path weighted by node execution times.

We consider a task set τ of n independent sporadic mixed-criticality parallel tasks $\{\tau_1, \tau_2, ..., \tau_n\}$. The tuple $(Z_i, C_i^N, C_i^O, L_i^N, L_i^O, D_i)$ characterizes a task τ_i. Z_i represents the criticality of a task. For example, in dual-criticality systems $Z_i \in \{\mathrm{LO}, \mathrm{HI}\}$, where *HI* (high-criticality) and *LO* (low-criticality) are the two criticality levels. C_i^N and C_i^O are the **nominal work** and **overload work**, respectively. C_i^N is the less pessimistic estimate generally expected to occur during normal operation, while C_i^O is the potentially much more pessimistic estimate considered during the certification process. Similarly, L_i^N and L_i^O are, respectively, the nominal and overload critical-path length estimates. In this chapter, we focus on *implicit-deadline sporadic tasks*, where the relative deadline D_i is equal to the minimum inter-arrival time between two consecutive jobs of the same task. For a task τ_i, its nominal utilization is denoted as $u_i^N = C_i^N / D_i$ and its overload utilization is $u_i^O = C_i^O / D_i$.

When a job J_i of task τ_i is released, we do not know its actual work $\mathcal{C}_i$ nor its actual critical-path length $\mathcal{L}_i$ in advance — these are only revealed as J_i executes. If a job J_i has $\mathcal{C}_i \leq C_i^N$ and $\mathcal{L}_i \leq L_i^N$, then we say that the job exhibits *nominal behavior*; otherwise, it exhibits *overload behavior*. We assume that $\mathcal{C}_i$ and $\mathcal{L}_i$ never exceed C_i^O and L_i^O, respectively.

5.1.2 System Model for Dual-Criticality System

A dual-criticality system (we extend it to multi-criticality system in Section 5.5) has two types of tasks: low- and high-criticality tasks. Thus this system has two corresponding states: **typical-state** (low-criticality mode) and **critical-state** (high-criticality mode). Crucially, at runtime the scheduler does not know if a task will exhibit nominal or overload behavior for a particular run. Thus, the system begins in the typical-state, assuming each job will satisfy its nominal C_i^N and L_i^N. If, however, any job overruns, then the system *transitions* to the critical-state. Jobs of the low-criticality tasks may be discarded, but the scheduler must ensure that all high-criticality jobs can still meet all their deadlines even if they all execute for their overload parameters C_i^O and L_i^O.

For the purpose of determining task set feasibility, the total utilization of a task set in the typical-state is the sum of the nominal utilizations of all tasks: $U^N = \sum_{\tau_i \in \tau} u_i^N$. Similarly, the total utilization of a task set in the critical-state is the sum of the overload utilizations of high-criticality tasks $U^O = \sum_{\tau_i \in \tau \text{ and } Z_i = \text{HI}} u_i^O$.

5.1.3 Schedulability Conditions for Dual-Criticality Systems

A mixed-criticality scheduler has two components. Given a task set, the *schedulability test* must first decide whether or not to admit this task set on a machine with m cores. If the test admits the task set, then the *runtime scheduler* must guarantee the following mixed-criticality correctness conditions.

Definition 2 *A dual-criticality scheduler is correct if for every task set it admits, it satisfies the following two conditions:*

(1) If the system stays in the typical-state during its entire execution, all tasks must meet their deadlines.

(2) After the system transitions into critical-state, all high-criticality tasks must meet their deadlines.

The runtime scheduler must transition properly from typical- to critical-state. Since the DAG of a particular job J_i unfolds dynamically during its execution, the runtime scheduler does not know, a priori, whether a particular job will exhibit nominal or overload behavior. Therefore, the runtime scheduler must infer the transition time (if any) of the system from typical- to critical-state dynamically at runtime, based on the execution of the current jobs. Note that low-criticality tasks need not define C_i^O, since the runtime scheduler is allowed to discard all low-criticality tasks in the critical-state. Even if defined, it is not used in our analysis and so we ignore it.

5.1.4 Dual-Criticality Capacity Augmentation Bound

In Chapter 2 a capacity augmentation bound is defined for parallel tasks with single-criticality. We generalize this definition to dual-criticality parallel tasks with implicit deadlines.

Definition 3 *A scheduler provides a* capacity augmentation bound *of b for dual-criticality parallel task systems, if it can schedule any task set which satisfies the following conditions.*

(1) The total nominal utilization of all tasks (high and low-criticality) in typical-state

$$U^N = \sum_{\tau_i \in \tau} C_i^N / D_i \le m/b$$

(2) The total overload utilization of high-criticality tasks in critical-state

$$U^O = \sum_{\tau_i \in \tau \text{ and } Z_i = HI} C_i^O / D_i \le m/b$$

(3) For all tasks, $L_i^N \le L_i^O \le D_i/b$.

Note that no scheduler can guarantee $b < 1$ — thus the capacity augmentation bound, just like utilization bound, provides an indication of how much slack is needed in the system to guarantee schedulability.

5.1.5 Background

We now describe a couple of ideas from prior work that MCFS is based on. In particular, MCFS uses important concepts from two different lines of work. It borrows the idea of *virtual deadlines* from work on mixed-criticality scheduling of sequential tasks [18, 21] and the idea of *federated scheduling* from single-criticality scheduling of parallel tasks in Chapter 4.

Virtual Deadline: MCFS utilizes the idea of *virtual deadline*, first used in single processor mixed-criticality scheduler EDF-VD [18] and also used in multiprocessor mixed-criticality schedulers [21]. At a high level, in these algorithms each high-criticality task is assigned a virtual deadline $D'_i < D_i$. This virtual deadline serves two purposes: (1) It boosts the priority of a high-criticality job so that if the job exhibits overload behaviour, then the scheduler detects it early and transitions into critical-state. (2) It provides enough slack so that after the transition, there is enough time to complete the overload work of the job.

To see how virtual deadlines are used for mixed-critical tasks, we briefly discuss three relevant algorithms, namely, *EDF-VD* [18] — an algorithm for scheduling dual-criticality tasks on a single processor; and *MC-Global* (referred to as Algorithm Global in [21]) and *MC-Partition* [21] — algorithms for scheduling dual criticality sequential tasks on m identical speed processors. In each of these algorithms, the schedulability test occurs as part of a pre-processing phase prior to runtime. At this time, all tasks are assigned a virtual deadline $D'_i \leq D_i$. Using this virtual deadline assignment, the schedulability of the task set is determined. At run-time, jobs are initially dispatched with the expectation that no job will execute for more than its nominal work and all jobs' deadlines are assumed to be D'_i instead of D_i. During runtime, if some job does execute beyond its nominal work, then the system *transitions* into the critical-state. The runtime scheduling and dispatching algorithms are immediately modified in the following way: (1) All currently-active jobs of low-criticality tasks are immediately discarded; henceforth, no job of a low-criticality task will be allowed to execute. (2) The scheduling algorithm now considers the real deadlines D_i rather than the virtual deadlines D'_i when making scheduling and dispatching decisions for high-criticality tasks.

The three algorithms differ in the details of their schedulability tests, calculation of the virtual deadlines, and details of their runtime scheduling and dispatching algorithms. EDF-VD is designed for single-processor systems and uses EDF for both low and high-criticality modes of the system. It guarantees a resource augmentation bound of 4/3. MC-Global algorithm extends the scheduling of sequential mixed-criticality workloads from single-processor to the multiprocessor setting and provides a resource augmentation bound of $\sqrt{5}+1$. MC-Partition partitions tasks among processors and then uses EDF-VD on each processor. In particular, a version of MC-Partition, namely MC-Partition-UT-0.75 provides a utilization bound of about 3/8, by incorporating scheduling tasks with utilizations up to 1 — we use this one as a black box in MCFS to schedule tasks whose utilization never exceeds 1.

Federated Scheduling: MCFS is based on the federated scheduling that assigns dedicated cores to high-utilization tasks and schedules them using a work-conserving scheduler. The key insight of the federated scheduling paradigm is to calculate the minimum number of cores to assign to a job in order to complete its remaining work and meet its deadline, which is given by the following lemma and used throughout this chapter.

Lemma 20 *If job J_i needs to execute C'_i remaining work and L'_i remaining critical-path length, it is schedulable by a work-conserving scheduler and can complete its execution within D'_i time on n_i dedicated cores, where $n_i \geq \frac{C'_i - L'_i}{D'_i - L'_i}$.*

Proof. This Lemma can be proved using arguments very similar to Theorem 16 in Chapter 4. Here, we only provide the intuition. Recall that a work-conserving (greedy) scheduler never keeps a core idle if there is any work available. We say that a time step is *incomplete* if any of the n_i dedicated cores is idle during that time step and the time step is *complete* otherwise. It is straight-forward to see that if the job's remaining critical-path length is at most L'_i, then the total number of incomplete steps is $I \leq L'_i$. In addition, the total number of complete steps is $X \leq \frac{C'_i - L'_i}{n_i} \leq D'_i - L'_i$, if we substitute the value of n_i. Since each step is either complete or incomplete, the total number of time steps to complete the job is $I + X$, which is bounded by D'_i. □

We now describe the basics of the federated scheduling for single-criticality parallel tasks on m identical cores. Each task τ_i has worst case execution time (work) C_i, worst case critical path length L_i, deadline (equal to minimum inter-arrival time) D_i, and utilization $u_i = C_i/D_i$. The federated scheduling algorithm first classifies tasks into either high utilization tasks ($u_i > 1$) or a low-utilization tasks ($u_i \leq 1$). Each high utilization task τ_i is assigned n_i dedicated cores, where $n_i = \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$. Therefore, the total number of cores assigned to high-utilization tasks is $n_{\text{high}} = \sum_{\tau_i \in \tau_{\text{high}}} n_i$. The remaining $n_{\text{low}} = m - n_{\text{high}}$ cores are assigned collectively to low-utilization tasks. The algorithm admits the task set τ, if n_{low} is non-negative and $n_{\text{low}} \geq 2 \sum_{\tau_i \in \tau_{\text{low}}} u_i$.

At runtime, any greedy (work-conserving) parallel scheduler can be used to schedule a high-utilization task τ_i on its assigned n_i cores. Low-utilization tasks are treated as though they are sequential tasks and any multiprocessor scheduling algorithm with a utilization bound of at least 50% can be used to schedule the low-utilization tasks on the allotted n_{low} cores. This approach provides a capacity augmentation bound of 2 — that is, a task set is admitted and meets all deadlines as long as its total utilization is smaller than $m/2$ and for each task τ_i, its worst-case critical-path length $L_i \leq D_i/2$.

5.2 Related Work on Mixed-Criticality Scheduling

In this section, we offer a brief survey of related work on real-time scheduling of mixed-criticality tasks. Since [150] first proposed a formal model for mixed-criticality systems, researchers have studied scheduling sequential tasks on both single processor [20,22,61,67,78,79,102,142] and multiprocessor machines [128,129]. In Section 5.1, we discussed the algorithms most relevant to our work that use virtual deadlines [18,21]. Models where other parameters, such as period and deadline, depend on the criticality of the task, have been investigated in [15,55,145].

None of these schedulers considers intra-task parallelism, however. Baruah [16] has considered limited forms of parallelism (such as that generated by Simulink programs). Most recently, Liu et. al [115] consider scheduling of mixed-criticality synchronous tasks using a decomposition-based strategy. Like all decomposition strategies, the parallel task is decomposed into sequential tasks before runtime, so the task structure must be known in advance and cannot change between different jobs of the same task. In contrast, MCFS considers a more general DAG model and does not assume that the scheduler knows the structure of the DAG in advance, allowing the task to generate different DAG structures in each run.

5.3 Scheduling Dual-Criticality High-Utilization Tasks

The MCFS scheduler consists of two parts: (1) a mapping algorithm (including schedulability test) runs before the tasks start executing; and (2) a runtime scheduler executes tasks if the task set is schedulable. Before runtime, the MCFS scheduler tries to generate a mapping for each criticality state: the *typical-state (low-criticality) mapping* S^T and the *critical-state (high-criticality) mapping* S^C. If it cannot find a valid mapping for both the states, then the task set is declared unschedulable. At runtime, the scheduler performs typical-state mapping S^T when all jobs exhibit nominal behavior. If any job exceeds its nominal parameters, then the system transitions into the critical-state and the scheduler switches to using mapping S^C.

In this section, we only consider dual-criticality systems where all tasks are high-utilization tasks. Intuitively, a task is a high-utilization task, if it requires parallel execution to meet its deadline. We consider task systems that contain both high and low-utilization tasks in Section 5.7. Table 5.1 shows the notation used throughout this chapter.

Definition 4 *A task is a* ***high-utilization task*** *if it is a high-criticality task with overload utilization larger than 1; or it is a low-criticality task with nominal utilization larger than 1.*

Table 5.1: Table of Notations in Chapter 5

Symbol	Meaning in the chapter
C_i^N (C_i^O)	Nominal (overload) work (or execution time) of task τ_i
L_i^N (L_i^O)	Nominal (overload) critical-path length of task τ_i
u_i^N (u_i^O)	Nominal (overload) utilization of task τ_i
D_i	Implicit deadline of sporadic task τ_i
D_i'	Assigned virtual deadline of τ_i for nominal behavior
n_i^N	Number of assigned cores to τ_i for nominal behavior
n_i^O	Number of assigned cores to τ_i for overload behavior
$\tau_{\mathcal{C}}$	Set of tasks in category $\mathcal{C} \in$ {LH, ..., HVH, HMH}
$U_{\mathcal{C}}^N$ ($U_{\mathcal{C}}^O$)	Total nominal (overload) utilization of category $\mathcal{C}$ tasks
$N_{\mathcal{C}}^N$ ($N_{\mathcal{C}}^O$)	Total #cores assigned to $\tau_{\mathcal{C}}$ for nominal (overload) behavior
$S^{\mathcal{S}}$	Mapping in state $\mathcal{S} \in$ {typical, intermediate, ..., critical}

Table 5.2: High-Utilization Task Classification

Task Type	Criticality	Nominal Utilization	Overload Utilization
HMH	High	$\frac{1}{b-1} < u_i^N \le u_i^O$	$1 < u_i^O$
HVH	High	$u_i^N \le \frac{1}{b-1}$	$1 < u_i^O$
LH	Low	$1 < u_i^N$	NA

5.3.1 Mapping Algorithm

MCFS computes two quantities for each task: (1) a virtual deadline; and (2) the number of cores assigned to the task in both the typical- and the critical-state. At a high level, the deadline assignment and the core assignment are designed to carefully balance the requirements in both states. To generate a mapping, MCFS classifies tasks into three categories.

1. **LO-High (LH)** tasks are low-criticality tasks with high-utilization in nominal behavior, i.e. $u_i^N > 1$. Again, these tasks are discarded in critical-state.

Table 5.3: High-Utilization Task Virtual Deadline and Core Assignment

Task Type	b	Virtual Deadline D'_i	Number of assigned cores n_i^N for nominal behavior	Number of assigned cores n_i^O for overload behavior
HMH	$2+\sqrt{2}$	$\frac{2D_i}{b}$	$\max\left\{\left\lceil\frac{C_i^N-L_i^N}{D'_i-L_i^N}\right\rceil, \lceil u_i^O\rceil\right\}$	$\max\left\{n_i^N, \left\lceil\frac{C_i^O-n_i^N D'_i-L_i^O}{D_i-D'_i-L_i^O}\right\rceil\right\}$
HVH	$2+\sqrt{2}$	$\frac{D_i}{b-1}$	$\lfloor u_i^O\rfloor$	$\left\lceil\frac{C_i^O-n_i^N D'_i-L_i^O}{D_i-D'_i-L_i^O}\right\rceil$
LH	2	D_i	$\left\lceil\frac{C_i^N-L_i^N}{D_i-L_i^N}\right\rceil$	NA

2. **HI-VeryLow-High (HVH)** tasks are high-criticality tasks with very low utilization $u_i^N \leq 1/(b-1)$ in nominal behavior and high utilization $u_i^O > 1$ in overload behavior.

3. **HI-Moderate-High (HMH)** tasks are high-criticality tasks with moderate utilization $u_i^N > 1/(b-1)$ in nominal behavior and high-utilization $u_i^O > 1$ in overload behavior.

Table 5.2 shows the classification criterion and Table 5.3 shows the virtual deadline assignments and the core assignments for all the categories. As mentioned earlier, we only consider high-utilization tasks; therefore, the above categories are exhaustive.

5.3.2 Schedulability Conditions of MCFS

The MCFS scheduler declares a task set schedulable, if and only if it is schedulable in both typical- and critical-states. The schedulability of a task set τ can be determined by the following conditions:

- If $L_i^N \geq D'_i$ for any task, or if $L_i^O \geq D_i - D'_i$ for any high-criticality task, then τ is declared unschedulable.

- If there are not enough cores for the typical-state mapping, i.e. $N^N_{\text{LH}} + N^N_{\text{HVH}} + N^N_{\text{HMH}} > m$, then τ is unschedulable.

- If there are not enough cores for the critical-state mapping, i.e. $N^O_{\text{HVH}} + N^O_{\text{HMH}} > m$, then τ is unschedulable.

- If none of above cases occurs, then τ is schedulable.

In the next section, we will show that if the MCFS schedulability test admits a task set, then the runtime scheduler guarantees meeting the correctness conditions from Definition 2. We will also show that MCFS has a capacity augmentation bound of $2+\sqrt{2} \approx 3.41$ for task sets with high-utilization tasks.

5.3.3 MCFS Runtime Execution

At runtime, the system is assumed to start in the typical-state and the runtime scheduler executes jobs according to the typical-state mapping S^T — that is, each task is scheduled by a work-conserving scheduler on n^N_i dedicated cores.

If a high-criticality job J_i does not complete within its virtual deadline D'_i, then the system transitions into the critical-state. All low-criticality jobs can be abandoned and future jobs of low-criticality tasks need not be admitted. The scheduler now executes all jobs according to their critical-state mapping S^C. That is, all high-utilization tasks are now allocated n^O_i dedicated cores and scheduled using a work-conserving scheduler.

Remark: Note that after transitioning to the critical-state, MCFS *needs not* abandon all low-criticality tasks; it can degrade gracefully by abandoning low-criticality tasks *on demand*. If, for instance, a high-criticality job J_i of τ_i exceeds its virtual deadline, it requires $n^O_i - n^N_i$ additional cores. Then, MCFS only needs to suspend enough low-criticality tasks to free up these cores and it can leave the remaining tasks unaffected. In addition, once job J

completes, MCFS can recover from critical-state to typical-state simply by giving n_i^N cores to τ_i and re-admitting the low-criticality tasks that were suspended.

5.4 Proof of Correctness and Capacity Augmentation Bound

We now prove that for high-utilization tasks MCFS guarantees (1) correctness (as described in Definition 2) and; (2) a capacity augmentation bound of $2+\sqrt{2}$ (as described in Definition 3). We first prove properties of the mappings generated by MCFS (from Table 5.3) for each of the three categories of tasks (from Table 5.2). In particular, we will show that each task of a category is schedulable under the generated mapping and the numbers of cores assigned to the task are bounded in both typical- and critical-state. These properties then allow us to prove correctness and the capacity augmentation bound.

Before diving into the proofs, we state two simple mathematical inequalities that will be used throughout the proofs.

(1) If $\frac{a}{b} \geq c > 0$ and $0 \leq x \leq y < b$, then $\frac{a}{b} \leq \frac{a-cx}{b-x} \leq \frac{a-cy}{b-y}$;

(2) If $0 < \frac{a}{b} \leq c$ and $0 \leq x \leq y < b$, then $\frac{a}{b} \geq \frac{a-cx}{b-x} \geq \frac{a-cy}{b-y}$;

5.4.1 LH tasks under MCFS

Recall that an LH task τ_i is a low-criticality task with high utilization ($u_i^N > 1$) under nominal condition. Since these tasks may be discarded in critical-state, we need only consider their typical-state behavior where they are assigned with $n_i^N = \left\lceil \frac{C_i^N - L_i^N}{D_i - L_i^N} \right\rceil$ dedicated cores (see Table 5.3) and a virtual deadline $D_i' = D_i$. Given these facts, the following lemma is obvious from Lemma 20.

Lemma 21 *LH tasks are schedulable under MCFS.*

We now prove that the number of cores assigned to a LH task is bounded.

Lemma 22 *For any $b \geq 3$, if a LH task τ_i has $D_i \geq bL_i^N$, then the number of cores it is assigned in the typical-state is bounded by $n_i^N \leq (b-1)u_i^N$.*

Proof. A LH task has high utilization; therefore, $u_i^N = C_i^N/D_i > 1$.

Since $L_i^N \leq D_i/b$, using Ineq (1), we have

$$\begin{aligned} n_i^N &= \left\lceil \frac{C_i^N - L_i^N}{D_i - L_i^N} \right\rceil < \frac{C_i^N - L_i^N}{D_i - L_i^N} + 1 \leq \frac{C_i^N - D_i/b}{D_i - D_i/b} + 1 \\ &\leq \frac{bu_i^N - 1}{b-1} + 1 = \frac{bu_i^N + b - 2}{b-1} \leq \frac{bu_i^N + (b-2)u_i^N}{b-1} = 2u_i^N \end{aligned}$$

Therefore, $n_i^N \leq (b-1)u_i^N$ for any $b \geq 3$. □

5.4.2 HVH tasks under MCFS

Recall that a HVH task τ_i is a high-criticality task with very low utilization $u_i^N \leq 1/(b-1)$ in the nominal condition and with high utilization $u_i^O \geq 1$ in the overload condition. Table 5.3 shows that MCFS assigns a HVH task $\lfloor u_i^O \rfloor$ dedicated cores in the typical-state and increases the number of allocated cores to $n_i^O = \left\lceil \frac{C_i^O - n_i^N D_i' - L_i^O}{D_i - D_i' - L_i^O} \right\rceil$ in the critical-state. Its virtual deadline is set as $D_i' = D_i/(b-1)$.

Lemma 23 *HVH tasks are schedulable under MCFS.*

Proof. In typical-state, the total work of an HVH task τ_i is $C_i^N = D_i u_i^N \leq D_i/(b-1) = D_i'$. Therefore, a single dedicated core is already enough to complete this work. Since $u_i^O \geq 1$, the number of cores assigned to a HVH task $n_i^N = \lfloor u_i^O \rfloor \geq 1$ is sufficient.

Now let us consider the critical-state. There are two cases:

Case 1: The transition occurred before the release of job j_i. Then the job gets $n_i^O = \left\lceil \frac{C_i^O - n_i^N D_i' - L_i^O}{D_i - D_i' - L_i^O} \right\rceil$ cores. Since $u_i^O > 1$, we have

$$n_i^N = \lfloor u_i^O \rfloor \leq u_i^O = \frac{C_i^O}{D_i} \leq \frac{C_i^O - L_i^O}{D_i - L_i^O}$$

Then by applying Ineq (1), we get

$$n_i^O = \left\lceil \frac{C_i^O - L_i^O - n_i^N D_i'}{D_i - L_i^O - D_i'} \right\rceil \geq \left\lceil \frac{C_i^O - L_i^O}{D_i - L_i^O} \right\rceil$$

Thus, n_i^O are sufficient by Lemma 20.

Case 2: The transition to critical-state occurred during the execution of a job j_i of task τ_i. Say j_i was released at time r_i and the transition to critical-state occurred at time $t \leq r_i + D_i'$ (If t is larger, then j_i would have finished executing already).

Let $e = t - r_i \leq D_i'$ be the duration for which the job executes before the transition. In these e time steps before the transition, say that the job has t^* complete steps (where all cores are busy working) and $e - t^*$ incomplete steps (where the critical-path length decreases). By definition, $t^* \leq e \leq D_i'$. Then, at the transition, it has at most $C^O - n_i^N t^* - e + t^*$ remaining work and $L_i^O - e + t^*$ remaining critical-path length that must be completed in $D_i - e$ time steps. By Lemma 20, τ_i is guaranteed to complete by the deadline, if τ_i is allocated at least n dedicated cores, where:

$$n = \left\lceil \frac{(C_i^O - n_i^N t^* - e + t^*) - (L_i^O - e + t^*)}{(D_i - e) - (L_i^O - e + t^*)} \right\rceil = \left\lceil \frac{C_i^O - L_i^O - n_i^N t^*}{D_i - L_i^O - t^*} \right\rceil$$

Similar to Case 1, since $n_i^N \leq \frac{C_i^O - L_i^O}{D_i - L_i^O}$ and $t^* \leq D_i'$, by Ineq (1) we have

$$n_i^O = \left\lceil \frac{C_i^O - n_i^N D_i' - L_i^O}{D_i - D_i' - L_i^O} \right\rceil \geq \left\lceil \frac{C_i^O - n_i^N t^* - L_i^O}{D_i - t^* - L_i^O} \right\rceil$$

By Lemma 20, the n_i^O cores are enough for it to be schedulable. □

We now bound the number of cores assigned to HVH tasks. Since HVH tasks have high overload utilization, the following lemma is true.

Lemma 24 *For an HVH task τ_i, the number of assigned cores in the typical-state is $n_i^N = \lfloor u_i^O \rfloor \leq u_i^O$.*

Lemma 25 *For an HVH task τ_i, if $D_i \geq bL_i^O \geq bL_i^N$, then the number of cores assigned in the critical-state is $n_i^O \leq bu_i^O$, for all $\frac{7+\sqrt{33}}{4} \leq b \leq \frac{5+\sqrt{17}}{2}$.*

Proof. Since HVH task τ_i satisfies $u_i^O > 1$, we can derive that $n_i^N \geq 1$ and $n_i^N = \lfloor u_i^O \rfloor \leq u_i^O \leq \frac{C_i^O - L_i^O}{D_i - L_i^O}$. Therefore, by Ineq (1) and $L_i^O \leq D_i/b$ we get

$$\begin{aligned} n_i^O &< \frac{C_i^O - n_i^N \frac{D_i}{b-1} - L_i^O}{D_i - \frac{D_i}{b-1} - L_i^O} + 1 \leq \frac{C_i^O - n_i^N \frac{D_i}{b-1} - \frac{D_i}{b}}{D_i - \frac{D_i}{b-1} - \frac{D_i}{b}} + 1 \\ &= \frac{b(b-1)u_i^O - bn_i^N - (b-1) + b^2 - 3b + 1}{b^2 - 3b + 1} \end{aligned}$$

Note that $\forall b \in [\frac{5-\sqrt{17}}{2}, \frac{5+\sqrt{17}}{2}]$ we have $b^2 - 5b + 2 \leq 0$. In addition, for $b \geq \frac{7+\sqrt{33}}{4} > \frac{3+\sqrt{5}}{2}$ it is true that $b^2 - 3b + 1 > 0$ and $b^2 - 3b + 2 > 0$.

Now let us consider two cases:

Case 1: If $u_i^O < 2$, then $n_i^N = 1$. Hence, by Ineq (1) we can derive

$$\begin{aligned} n_i^O &< \frac{b(b-1)u_i^O + b^2 - 5b + 2}{b^2 - 3b + 1} \\ &< \frac{b(b-1)u_i^O + (b^2 - 5b + 2)\frac{u_i^O}{2}}{b^2 - 3b + 1} \qquad \text{[since } b^2 - 5b + 2 \leq 0 \text{ and } \tfrac{u_i^O}{2} < 1] \\ &= \frac{3b^2 - 7b + 2}{2(b^2 - 3b + 1)} u_i^O \end{aligned}$$

Case 2: If $u_i^O \geq 2$, then $n_i^N = \lfloor u_i^O \rfloor > u_i^O - 1$. By Ineq (1), we can derive

$$\begin{aligned} n_i^O &< \frac{b(b-1)u_i^O - b(u_i^O - 1) - (b-1) + b^2 - 3b + 1}{b^2 - 3b + 1} \\ &= \frac{b(b-2)u_i^O + b^2 - 3b + 2}{b^2 - 3b + 1} \\ &\leq \frac{b(b-2)u_i^O + (b^2 - 3b + 2)\frac{u_i^O}{2}}{b^2 - 3b + 1} \qquad \text{[since } b^2 - 3b + 2 > 0 \text{ and } \tfrac{u_i^O}{2} \geq 1] \\ &= \frac{3b^2 - 7b + 2}{2(b^2 - 3b + 1)} u_i^O \end{aligned}$$

Therefore, in both cases, we have

$$n_i^O < \frac{3b^2 - 7b + 2}{2(b^2 - 3b + 1)} u_i^O \tag{5.3}$$

By solving $\frac{3b^2-7b+2}{2(b^2-3b+1)} \leq b$, which is equivalent to $2b^2 - 7b + 2 \geq 0$ for $b > 1$, we can conclude that for all $b \geq \frac{7+\sqrt{33}}{4} \approx 3.19$, we have $n_i^O \leq bu_i^O$.

Finally, by intersecting all the ranges of b, we get the required result $n_i^O \leq bu_i^O$, for all $\frac{7+\sqrt{33}}{4} \leq b \leq \frac{5+\sqrt{17}}{2}$. □

Remark: The classification and core assignment to HVH tasks may seem strange at first glance. Since the tasks have such a low utilization in the typical-state, why do we assign dedicated cores rather than assigning multiple tasks to each core? This is due to balancing core assignments in the typical- and critical-state. Intuitively, if tasks share cores (basically assigning fewer cores per task) in the typical-state, then we must assign more cores in the critical-state. In particular, note that Lemma 25 uses the fact that the task has dedicated cores to prove a lower bound on the amount of work this task completes by its virtual deadline, allowing us to upper bound the amount of left-over work in the critical-state. If we didn't assign dedicated cores, then such a lower bound would be difficult to prove; therefore, MCFS would have to assign more cores to these tasks in the critical-state, giving a worse bound.

5.4.3 HMH tasks under MCFS

Recall that a HMH task τ_i is a high-criticality task with moderate or high utilization $u_i^N > 1/(b-1)$ in the nominal condition and with high utilization $u_i^O > 1$ in the overload condition. Table 5.3 shows that MCFS assigns each HMH task $n_i^N = \max\left\{\left\lceil \frac{C_i^N - L_i^N}{2D_i/b - L_i^N} \right\rceil, \lceil u_i^O \rceil\right\}$ dedicated cores in the typical-state and increases the number of allocated cores to $n_i^O = \max\left\{n_i^N, \left\lceil \frac{C_i^O - n_i^N D_i' - L_i^O}{D_i - D_i' - L_i^O} \right\rceil\right\}$ in the critical-state. Its virtual deadline is $D_i' = 2D_i/b$.

Lemma 26 *HMH tasks are schedulable under MCFS.*

Proof. In the typical-state, by Lemma 20 we know that $\left\lceil \frac{C_i^N - L_i^N}{2D_i/b - L_i^N} \right\rceil$ is sufficient for a HMH task τ_i to complete its C_i^N and L_i^N within its virtual deadline $2D_i/b$. Thus, n_i^N cores are enough for it to be schedulable in the typical-state.

Now let us consider the critical-state. There are two cases:

Case 1: The transition occurred before the release of j_i.

For $n_i^O = \max\left\{ n_i^N, \left\lceil \frac{C_i^O - n_i^N D_i' - L_i^O}{D_i - D_i' - L_i^O} \right\rceil \right\}$, there are two sub-cases:

(a) If $n_i^N > \frac{C_i^O - L_i^O}{D_i - L_i^O}$, then by n_i^N being integer and Ineq (2), we get

$$n_i^O = n_i^N \geq \left\lceil \frac{C_i^O - L_i^O}{D_i - L_i^O} \right\rceil \geq \left\lceil \frac{C_i^O - L_i^O - n_i^N D_i'}{D_i - L_i^O - D_i'} \right\rceil$$

(b) If $n_i^N \leq \frac{C_i^O - L_i^O}{D_i - L_i^O}$, similarly by applying Ineq (1), we get

$$n_i^O = \left\lceil \frac{C_i^O - L_i^O - n_i^N D_i'}{D_i - L_i^O - D_i'} \right\rceil \geq \left\lceil \frac{C_i^O - L_i^O}{D_i - L_i^O} \right\rceil \geq n_i^N$$

Hence, in both sub-cases, we have $n_i^O \geq \left\lceil \frac{C_i^O - L_i^O}{D_i - L_i^O} \right\rceil$. Thus, by Lemma 20, n_i^O cores is sufficient to complete its work C_i^O and critical-path length L_i^O within deadline D_i.

Case 2: The transition to critical-state occurred during the execution of a job j_i of task τ_i. Say j_i was released at time r_i and the transition to critical-state occurred at time $t \leq r_i + D_i'$ (If t is larger, then j_i would have finished executing already).

Let $e = t - r_i \leq D_i'$ be the duration for which the job executes before the transition. In these e time steps before the transition, say that the job has t^* complete steps (where all cores are busy working) and $e - t^*$ incomplete steps (where the critical-path length decreases). By definition, $t^* \leq D_i'$. Similar to Case 2 in Lemma 23, τ_i is guaranteed to complete by the deadline, if τ_i is allocated at least $n = \left\lceil \frac{C_i^O - L_i^O - n_i^N t^*}{D_i - L_i^O - t^*} \right\rceil$ dedicated cores.

Again, there are two cases:

(a) If $n_i^N > \frac{C_i^O - L_i^O}{D_i - L_i^O}$, then by $t^* \leq D_i'$ and Ineq (2), we get

$$n_i^O = n_i^N \geq \left\lceil \frac{C_i^O - L_i^O}{D_i - L_i^O} \right\rceil \geq n \geq \left\lceil \frac{C_i^O - L_i^O - n_i^N D_i'}{D_i - L_i^O - D_i'} \right\rceil$$

(b) If $n_i^N \leq \frac{C_i^O - L_i^O}{D_i - L_i^O}$, then by $t^* \leq D_i'$ and Ineq (1) we get

$$n = \left\lceil \frac{C_i^O - L_i^O - n_i^N t^*}{D_i - L_i^O - t^*} \right\rceil \leq \left\lceil \frac{C_i^O - L_i^O - n_i^N D_i'}{D_i - L_i^O - D_i'} \right\rceil \leq n_i^O$$

Since $n_i^O \geq n$ in both cases, n_i^O cores are enough for a HMH job j_i to be schedulable if the transition happens during its execution. □

In the following two lemmas, we bound the numbers of cores assigned to HMH tasks in both the typical- and critical-state.

Lemma 27 *For each HMH task τ_i, if $D_i \geq bL_i^N$, then the number of cores assigned in typical-state is bounded by $n_i^N \leq (b-1)u_i^N + u_i^O$, for any $b \geq 2$.*

Proof. HMH task τ_i satisfies $u_i^N > 1/(b-1)$ and $u_i^O > 1 \geq 2/b$, for $b \geq 2$. We consider three cases for u_i^N and n_i^N:

Case 1. $\mathbf{u_i^N \leq 2/b}$:

Since $C_i^N \leq 2D_i/b$, from Inequality (2), we have

$$\frac{C_i^N - L_i^N}{2D_i/b - L_i^N} \leq \frac{C_i^N}{2D_i/b} \leq 1 \leq u_i^O \Rightarrow \left\lceil \frac{C_i^N - L_i^N}{2D_i/b - L_i^N} \right\rceil \leq \lceil u_i^O \rceil$$

Since $1/(b-1) < u_i^N$, we know $(b-1)u_i^N > 1$. We can derive

$$n_i^N = \lceil u_i^O \rceil < 1 + u_i^O < (b-1)u_i^N + u_i^O$$

Case 2. $\mathbf{u_i^N > 2/b}$ and $\left\lceil \frac{\mathbf{C_i^N - L_i^N}}{\mathbf{2D_i/b - L_i^N}} \right\rceil \leq \lceil \mathbf{u_i^O} \rceil$:

This case is similar to Case 1; we also have $n_i^N = \lceil u_i^O \rceil < (b-1)u_i^N + u_i^O$.

Case 3. $\mathbf{u_i^N > 2/b}$ and $\left\lceil \frac{\mathbf{C_i^N - L_i^N}}{\mathbf{2D_i/b - L_i^N}} \right\rceil > \lceil \mathbf{u_i^O} \rceil$:

In this case, since $C_i^N \geq 2D_i/b$ and $L_i^N \leq D_i/b$, from Inequality (1),

$$n_i^N = \left\lceil \frac{C_i^N - L_i^N}{2D_i/b - L_i^N} \right\rceil < \frac{C_i^N - L_i^N}{2D_i/b - L_i^N} + 1 \leq \frac{C_i^N - D_i/b}{2D_i/b - D_i/b} + 1 = bu_i^N$$

Since $u_i^N \leq u_i^O$, we get $n_i^N \leq bu_i^N \leq (b-1)u_i^N + u_i^O$. □

Lemma 28 *For a HMH task τ_i, if $D_i \geq bL_i^O \geq bL_i^N$, then the number of cores assigned in the critical-state is bounded by $n_i^O \leq bu_i^O$, for all $4 > b \geq 2 + \sqrt{2}$.*

Proof. We denote $n' = \left\lceil \frac{C_i^O - n_i^N D_i' - L_i^O}{D_i - D_i' - L_i^O} \right\rceil$ and we will show that $n' < bu_i^O$.

$$\begin{aligned} n' &= \left\lceil \frac{C_i^O - n_i^N D_i' - L_i^O}{D_i - D_i' - L_i^O} \right\rceil \\ &\leq \left\lceil \frac{C_i^O - u_i^O D_i' - L_i^O}{D_i - D_i' - L_i^O} \right\rceil \qquad [\text{since } n_i^N \geq \lceil u_i^O \rceil \geq u_i^O] \\ &= \left\lceil \frac{C_i^O - 2C_i^O/b - L_i^O}{D_i - 2D_i/b - L_i^O} \right\rceil \qquad [\text{since } D_i' = 2D_i/b] \\ &< \frac{C_i^O - 2C_i^O/b - L_i^O}{D_i - 2D_i/b - L_i^O} + 1 \end{aligned}$$

Since HMH task satisfies $u_i^O > 1$, so we have

$$C_i^O - \frac{2C_i^O}{b} = (1 - \frac{2}{b})u_i^O D_i > (1 - \frac{2}{b})D_i = D_i - \frac{2D_i}{b}$$

Again by applying Inequality (1), we can get

$$\begin{aligned}
n' &\leq \frac{C_i^O - 2C_i^O/b - D_i/b}{D_i - 2D_i/b - D_i/b} + 1 \qquad [\text{since } L_i^O \leq D_i/b] \\
&= \frac{(b-2)u_i^O - 1}{b-3} + 1 \\
&< \frac{(b-2)u_i^O}{b-3} \qquad [\text{since } 1 - \tfrac{1}{b-3} < 0, \forall 4 > b > 3] \\
&\leq \frac{b(b-3)u_i^O}{b-3} \qquad [\text{since } b(b-3) \geq (b-2), \forall b \geq 2 + \sqrt{2}] \\
&= bu_i^O
\end{aligned} \tag{5.4}$$

In Lemma 27, we know that $n_i^N \leq (b-1)u_i^N + u_i^O \leq bu_i^O$. Thus, by intersecting all the ranges of b, we get the required result $n_i^O = \max\left\{n_i^N, n'\right\} \leq bu_i^O$, for all $4 > b \geq 2 + \sqrt{2}$. □

Remark: At a high-level, the intuition behind the allocation is similar to HVH tasks, albeit more complex. Clearly, n_i^N must be enough to schedule the nominal work; we need $n_i^N \geq \left\lceil \frac{C_i^N - L_i^N}{D_i' - L_i^N} \right\rceil$. In addition, we must balance the utilization in typical- and critical-states by providing a lower bound on the amount of work done in the typical-state. In the first line of Lemma 28, we calculate this lower bound by using the fact that $n_i^N \geq u_i^O$. The overload assignment is somewhat more straightforward; we must assign enough cores to complete the remaining work after the mode transition, i.e. $n_i^O \geq \left\lceil \frac{C_i^O - n_i^N D_i' - L_i^O}{D_i - D_i' - L_i^O} \right\rceil$. In addition, we must be careful that the number of cores *does not decrease* after the transition.

5.4.4 Proof of Correctness

Theorem 29 *MCFS is correct — if the MCFS schedulability test declares a task set schedulable, then the runtime scheduler guarantees that the conditions in Definition 2 are met for all possible executions of the task system.*

The correctness is obvious from Lemmas 21, 23 and 26.

Remark: Note that the correctness of MCFS does not rely on a particular b. In fact, the schedulability test in Section 5.3.2 can use any $b > 2$ to check the schedulability. In fact, it can use different b's for different tasks. If the test passes for any set of b's for various tasks, then the task set is schedulable. Therefore, in principle, one can do an exhaustive search using different values of b's for different tasks to check for schedulability. The particular values of b we used in our description are only in order to provide the capacity augmentation bound. Based on this observation, we provide an improved schedulability test and mapping algorithm for these high-utilization tasks in Section 5.6.

5.4.5 Proof of Capacity Augmentation Bound $2 + \sqrt{2}$

We now show a capacity augmentation bound for a particular $b = 2 + \sqrt{2}$; that is, if the total utilizations in both the typical and critical-state are no more than m/b and the critical-path lengths in both the nominal and overload condition are no more than the deadline divided by b for all tasks, then MCFS always declares the task set schedulable.

We first define some notations, summarized in Table 5.1. The total nominal utilization of all tasks of category $\mathcal{C} \in \{\text{LH, HVH, HMH}\}$ is denoted by $U^N_{\mathcal{C}}$ and their total overload utilization is $U^O_{\mathcal{C}}$. Similarly, the total number of cores assigned to tasks in category $\mathcal{C}$ in the typical-state mapping S^T is $N^N_{\mathcal{C}}$ and in the critical-state mapping S^C is $N^O_{\mathcal{C}}$.

Theorem 30 *MCFS has a capacity augmentation bound of $2+\sqrt{2}$. That is, if the conditions from Definition 3 hold for $b = 2 + \sqrt{2}$, then the task set always satisfies the following conditions (from Section 5.3.2):*

(1) virtual deadline is valid — any task has $L^N_i \leq D'_i$ and any high-criticality task has $L^O_i \leq D_i - D'_i$;

(2) typical-state mapping is valid — $N^N_{LH} + N^N_{HVH} + N^N_{HMH} \leq m$;

(3) critical-state mapping is valid — $N^O_{HVH} + N^O_{HMH} \leq m$.

We prove the theorem by showing that MCFS satisfies each of the required conditions via the following three lemmas.

Lemma 31 *For any $b > 3$, if $L_i^N \leq L_i^O \leq D_i/b$ (Condition 3 of Definition 3), then the virtual deadline is always valid.*

Proof. **LHI tasks:** Virtual deadline $D_i' = D_i$, so $L_i^N \leq D_i'$.

HVH tasks: Virtual deadline $D_i' = D_i/(b-1) > D_i/b \geq L_i^N$ and $D_i - D_i' = (b-2)D_i/(b-1) > D_i/b \geq L_i^O$, since for any $b > 3$, we have $(b-2)/(b-1) > (b-2)/b \geq 1/b$.

HMH tasks: Virtual deadline is $D_i' = 2D_i/b > D_i/b \geq L_i^N$ and $D_i - D_i' = (b-2)D_i/b > D_i/b \geq L_i^O$ for $b > 3$. □

We now argue that a valid mapping S^T can be generated for typical-state for a capacity augmentation bound of $b = 2 + \sqrt{2}$.

Lemma 32 *If the Conditions of Definition 3 hold for any $b \geq 2 + \sqrt{2}$, then $m \geq N_{LH}^N + N_{HVH}^N + N_{HMH}^N$ and the typical state mapping S^T is valid.*

Proof. **LHI tasks:** From Lemma 22, for any $b \geq 2 + \sqrt{2} > 3$ we have

$$N_{\text{LH}}^N = \sum_{\tau_i \in \tau_{\text{LH}}} n_i^N \leq \sum_{\tau_i \in \tau_{\text{LH}}} (b-1)u_i^N = (b-1)U_{\text{LH}}^N$$

HVH tasks: From Lemma 24, for any b we always have

$$N_{\text{HVH}}^N = \sum_{\tau_i \in \tau_{\text{HVH}}} n_i^N \leq \sum_{\tau_i \in \tau_{\text{HVH}}} u_i^O = U_{\text{HVH}}^O$$

HMH tasks: From Lemma 27, for $b \geq 2 + \sqrt{2} > 2$ we have

$$N_{\text{HMH}}^N \leq \sum_{\tau_i \in \tau_{\text{HMH}}} ((b-1)u_i^N + u_i^O) = (b-1)U_{\text{HMH}}^N + U_{\text{HMH}}^O$$

Since the Conditions of Definition 3 hold, we know $U^N = U^N_{\text{LH}} + U^N_{\text{HVH}} + U^N_{\text{HMH}} \leq \frac{m}{b}$ and $U^O = U^O_{\text{HVH}} + U^O_{\text{HMH}} \leq \frac{m}{b}$. Therefore, we can derive

$$\begin{aligned}
&N^N_{\text{LH}} + N^N_{\text{HVH}} + N^N_{\text{HMH}} \\
\leq &(b-1)U^N_{\text{LH}} + U^O_{\text{HVH}} + (b-1)U^N_{\text{HMH}} + U^O_{\text{HMH}} \\
\leq &(b-1)(U^N_{\text{LH}} + U^N_{\text{HVH}} + U^N_{\text{HMH}}) + U^O_{\text{HVH}} + U^O_{\text{HMH}} \\
\leq &(b-1)m/b + m/b \\
= &m
\end{aligned}$$

Thus, typical-state mapping is always valid if $b = 2 + \sqrt{2}$. $\square$

We now argue that the critical-state mapping S^C is valid for $b = 2 + \sqrt{2}$.

Lemma 33 *For a task set in critical-state under MCFS, if the Conditions of Definition 3 hold for $b = 2 + \sqrt{2} \approx 3.41$, then $m \geq N^O_{HVH} + N^O_{HMH}$ and mapping S^C is valid.*

Proof. **HVH tasks:** From Lemma 25, for for $b = 2 + \sqrt{2} \approx 3.41 > \frac{7+\sqrt{33}}{4} \approx 3.19$

$$N^O_{\text{HVH}} = \sum_{\tau_i \in \tau_{\text{HVH}}} n^O_i \leq \sum_{\tau_i \in \tau_{\text{HVH}}} bu^O_i = bU^O_{\text{HVH}}$$

HMH tasks: From Lemma 28, for $b = 2 + \sqrt{2}$ we know

$$N^C_{\text{HMH}} = \sum_{\tau_i \in \tau_{\text{HMH}}} n^O_i \leq \sum_{\tau_i \in \tau_{\text{HMH}}} bu^O_i = bU^O_{\text{HMH}}$$

Since the Conditions of Definition 3 hold, $U^O = U^O_{\text{HVH}} + U^O_{\text{HMH}} \leq m/b$.

$$N^C_{\text{HVH}} + N^C_{\text{HMH}} \leq bU^O_{\text{HVH}} + bU^O_{\text{HMH}} \leq m$$

Thus, critical-state mapping is always valid if $b = 2 + \sqrt{2}$. $\square$

Based on the properties of LH, HVH and HMH tasks, we observe that the bound of $2+\sqrt{2}$ is only required for HMH tasks in the critical-state mapping.

5.4.6 Lower Bound on Capacity Augmentation for High-Utilization Tasks

Now we use an example task set to show the tightness of MCFS capacity augmentation bound.

Theorem 34 *The capacity augmentation bound for MCFS for high-utilization tasks on m cores is at least $\beta = 2 - \frac{3}{2m} + \sqrt{(2-\frac{3}{2m})^2 - 2}$, when $m > 9$. When $m \to \infty$, $\beta \to 2+\sqrt{2}$. When $m \to 9$, $\beta \to 3$.*

Proof. On a system with m cores, consider a task set τ with a two tasks: τ_1 is a low criticality LH task with an utilization of $u_1 = 1+\epsilon_1$, where ϵ_1 is an arbitrarily small positive number; τ_2 is a high criticality HMH task with an overload utilization of $u_2^O = \frac{-3}{\beta^2-4\beta+2}$ and a nominal utilization of $u_2^N = (\frac{2}{b} - \frac{1}{\beta})u_2^O$. In addition, the nominal and overload critical-path length of both tasks equals to $1/\beta$ of their deadlines.

Note that $\beta = 2 - \frac{3}{2m} + \sqrt{(2-\frac{3}{2m})^2 - 2}$ is one of the roots of $\frac{-3}{\beta^2-4\beta+2} = m/\beta$. Hence, the total overload utilization of the task set is $u_2^O = m/\beta$, while the total nominal utilization of the task set is less than m/β. Therefore, the task set satisfies the conditions from Section 5.3.2 for a bound of β.

For HMH task τ_2, we know that by the MCFS mapping $b = 2+\sqrt{2}$ and

$$\left\lceil \frac{C_2^N - L_2^N}{2D_i/b - L_2^N} \right\rceil = \left\lceil \frac{\beta u_2^N - 1}{2\beta/b - 1} \right\rceil = \left\lceil \frac{\beta(\frac{2}{b} - \frac{1}{\beta})u_2^O - 1}{2\beta/b - 1} \right\rceil < \lceil u_2^O \rceil$$

Hence it is assigned with $n_2^N = \lceil u_2^O \rceil$ cores in the typical-state mapping.

Note that the larger the m, the larger the β. When $m \to \infty$, $\beta \to 2+\sqrt{2}$. Hence, $\beta > b$. In the critical-state mapping, τ_2 is assigned with n_2^O cores.

$$\begin{aligned}
n_2^O &= \left\lceil \frac{C_i^O - n_i^N D_i' - L_i^O}{D_i - D_i' - L_i^O} \right\rceil = \left\lceil \frac{\beta u_2^O - \frac{2\beta}{b}\left\lceil u_2^O \right\rceil - 1}{\beta - \frac{2\beta}{b} - 1} \right\rceil \\
&\geq \left\lceil \frac{\beta u_2^O - 2\left\lceil u_2^O \right\rceil - 1}{\beta - 2 - 1} \right\rceil > \left\lceil \frac{\beta u_2^O - 2(u_2^O + 1) - 1}{\beta - 2 - 1} \right\rceil \\
&= \left\lceil \frac{(\beta - 2)u_2^O - 3}{\beta - 3} \right\rceil \geq \frac{(\beta - 2)u_2^O - 3}{\beta - 3}
\end{aligned}$$

Note that for $u_2^O = \frac{-3}{\beta^2 - 4\beta + 2}$, we have $\frac{(\beta-2)u_2^O - 3}{\beta - 3} = \beta u_2^O$. Hence, $n_2^O > \beta u_2^O = m$ and there is not enough cores to assign to task τ_2 in the critical-state mapping. Therefore, we reach the conclusion. $\square$

5.5 MCFS for Multi-Criticality Systems

We now extend MCFS to systems with more than two criticality levels, still assuming that all tasks are high-utilization tasks. We describe the system model with 3 criticality levels, which can be generalized easily to more than three levels. Then, we will argue that MCFS provides a capacity augmentation bound of $(5+\sqrt{5})/2$ for systems with 3 or more criticality levels.

5.5.1 Multi-Criticality System Model

The tuple $(Z_i, C_i^N, C_i^O, L_i^N, L_i^O, D_i)$ still represents a task; that is, in our model, each task still has two behaviours: nominal work C_i^N estimated by the system designer and overload work C_i^O estimated by the certification authorities (similarly for critical-path length).[3] In all the criticality levels that are same or below Z_i, task τ_i exhibits nominal behavior. If

[3]There is another multi-criticality model [150] assuming that a task has more than two work estimates, one for each criticality level.

τ_i overruns its nominal parameters, then the system transitions to the criticality level Z_i. The only exception is for tasks with the lowest criticality level $Z_i = LO$: if they overrun their nominal parameters, they are allowed to miss their deadlines. An example for three criticality levels $Z_i \in \{\mathrm{LO}, \mathrm{ME}, \mathrm{HI}\}$ for low, medium and high, is shown in Table 5.4.

Table 5.4: Tasks' Per-Criticality Work, Critical-Path Length and Core Assignment of a 3-Criticality System

	Work, Critical-Path Length			Core Assignment under MCFS		
Task Criticality	LO-Work	ME-Work	HI-Work	Typical-State	Intermediate-State	Critical-State
LO	C_i^N, L_i^N	-	-	n_i^N	-	-
ME	C_i^N, L_i^N	C_i^O, L_i^O	-	n_i^N	n_i^O	-
HI	C_i^N, L_i^N	C_i^N, L_i^N	C_i^O, L_i^O	n_i^N	n_i^N	n_i^O

A scheduler for a 3-Criticality System must satisfy the following conditions:

(1) If the system remains in the typical-state, then all tasks must meet their deadlines based on their nominal parameters (work and critical-path length);

(2) If any medium-criticality task exceeds its nominal parameters, then the system transitions into the intermediate-state — all medium- and high-criticality tasks must meet their deadlines based on their medium-criticality parameters (including work and critical-path length) shown in the second column in Table 5.4 (i.e. nominal parameters for high-criticality tasks and overload parameters for medium-criticality tasks). The scheduler is allowed to discard all low-criticality tasks;

(3) If any high-criticality task overruns its nominal parameters, then the system transition into the critical-state. High-criticality tasks still meet their deadlines based on their high-criticality parameters shown in the third column in Table 5.4 (i.e. overload parameters). The scheduler is allowed to discard all low and medium-criticality tasks.

Table 5.5: High-Utilization Task Classification of a 3-Criticality System

Task Type	Criticality	Nominal Utilization	Overload Utilization
HMH	High	$\frac{1}{b-1} < u_i^N \leq u_i^O$	$1 < u_i^O$
HVH	High	$u_i^N \leq \frac{1}{b-1}$	$1 < u_i^O$
MMH	**Medium**	$\frac{1}{b-1} < u_i^N \leq u_i^O$	$1 < u_i^O$
MVH	**Medium**	$u_i^N \leq \frac{1}{b-1}$	$1 < u_i^O$
LH	Low	$1 < u_i^N$	NA

Table 5.6: High-Utilization Tasks' Assignments of a 3-Criticality System

Task Type	b	Virtual Deadline D'_i	Number of assigned cores n_i^N for nominal behavior	Number of assigned cores n_i^O for overload behavior
HMH	$2+\sqrt{2}$	$\frac{2D_i}{b}$	$\max\left\{\left\lceil\frac{C_i^N-L_i^N}{D'_i-L_i^N}\right\rceil, \lceil u_i^O\rceil\right\}$	$\max\left\{n_i^N, \left\lceil\frac{C_i^O-n_i^N D'_i-L_i^O}{D_i-D'_i-L_i^O}\right\rceil\right\}$
HVH	$2+\sqrt{2}$	$\frac{D_i}{b-1}$	$\lfloor u_i^O\rfloor$	$\left\lceil\frac{C_i^O-n_i^N D'_i-L_i^O}{D_i-D'_i-L_i^O}\right\rceil$
MMH	$\boldsymbol{(5+\sqrt{5})/2}$	$\frac{2D_i}{b}$	$\max\left\{\left\lceil\frac{C_i^N-L_i^N}{D'_i-L_i^N}\right\rceil, \lceil u_i^O\rceil\right\}$	$\max\left\{n_i^N, \left\lceil\frac{C_i^O-n_i^N D'_i-L_i^O}{D_i-D'_i-L_i^O}\right\rceil\right\}$
MVH	$\boldsymbol{(5+\sqrt{5})/2}$	$\frac{D_i}{b-1}$	$\lfloor u_i^O\rfloor$	$\left\lceil\frac{C_i^O-n_i^N D'_i-L_i^O}{D_i-D'_i-L_i^O}\right\rceil$
LH	2	D_i	$\left\lceil\frac{C_i^N-L_i^N}{D_i-L_i^N}\right\rceil$	NA

5.5.2 Multi-Criticality MCFS Algorithm and Bound

We now generalize the MCFS algorithm to 3-criticality systems. The classification, virtual deadline and core assignments are shown in Table 5.5 and 5.6. Note that the classification is similar to the one shown in Section 5.3. Moreover, the assignments for medium-criticality tasks are almost identical to high-criticality tasks, except for a slightly larger b that is designed to provide the capacity augmentation bound of multi-criticality MCFS.

To calculate the mappings S^T, S^I, and S^C, we simply assign cores according to the task behavior shown in Table 5.4. For instance, in the intermediate-state mapping, a medium-criticality task gets n_i^O cores while a high-criticality task gets n_i^N cores. In the schedulability test, we add an additional condition saying that the total number of cores assigned in the intermediate-state is at most m. The other conditions remain the same. The following theorem gives the capacity augmentation bound.

Theorem 35 *Multi-criticality MCFS with only high-utilization tasks is correct and has a capacity bound of $(5+\sqrt{5})/2$ — if the conditions from Definition 3 hold for $b = (5+\sqrt{5})/2$, then the task set satisfies the following conditions of MCFS schedulability test:*

(1) virtual deadlines for high-utilization tasks are valid — any LH, HVH or HMH task has $L_i^N \leq D_i'$ and any HVH or HMH task has $L_i^O \leq D_i - D_i'$;

(2) typical-state mapping is valid — $n_{LH}^N + n_{MVH}^N + n_{MMH}^N + n_{HVH}^N + n_{HMH}^N \leq m$;

(3) intermediate-state mapping is valid — $n_{MVH}^O + n_{MMH}^O + n_{HVH}^N + n_{HMH}^N \leq m$;

(4) critical-state mapping is valid — $n_{HVH}^O + n_{HMH}^O \leq m$.

Proof. Recall that we noted in Section 5.4.4 that if the schedulability test passes for any $b > 2$, then the task set is schedulable. Since medium-criticality tasks are classified and scheduled just like high-criticality ones with the only modification that $b = (5+\sqrt{5})/2 \approx 3.62$, the correctness is obvious.

The proof of capacity augmentation is similar to Theorem 30: the critical-state mapping is identical to Lemma 33; and the proofs for the typical-state mapping and the virtual deadline are similar to Lemma 32 and Lemma 31, respectively, with the only difference being $b = (5+\sqrt{5})/2$.

Now we only prove the differences. We must bound the number of assigned cores in the intermediate-state mapping and prove that $N_{\text{MVH}}^O + N_{\text{MMH}}^O + N_{\text{HVH}}^N + N_{\text{HMH}}^N \leq m$. Note that in the intermediate-state, medium-criticality tasks are assigned cores according to their overload parameters, while high-criticality tasks are assigned according to their nominal parameters.

For high-criticality tasks, from Lemmas 24 and 27 we have

$$N^N_{\text{HVH}} + N^N_{\text{HMH}} \leq U^O_{\text{HVH}} + (b-1)U^N_{\text{HMH}} + U^O_{\text{HMH}}$$

Medium-criticality tasks are more interesting. In order to use a lemma similar to Lemma 32, we must bound n^O_{MVH} and n^O_{MMH}. Unfortunately, it is not sufficient to bound them by bU^O_{MVH} and bU^O_{MMH} as in Lemmas 25 and 28.

Instead, we show a modified result similar to Lemma 25 and bound n^O_{MVH} by $(b-1)U^O_{\text{MVH}}$. From Lemma 25, we know that Inequality (5.3) is correct for any b. Since we want to bound n^O_{MVH} by $(b-1)U^O_{\text{MVH}}$, we need to solve the inequality $\frac{3b^2-7b+2}{2(b^2-3b+1)} \leq b-1$, which is equivalent to $2b^3 - 11b^2 + 15b - 4 \geq 0$. We denote $f(b) = 2b^3 - 11b^2 + 15b - 4$. Note that $f(b)$ is always increasing for $b \geq (5+\sqrt{5})/2 \approx 3.62 > (11+\sqrt{31})/6 \approx 2.76$ and $f((5+\sqrt{5})/2) = 1 > 0$. Therefore, for $b \geq (5+\sqrt{5})/2$, we have:

$$n^O_i \leq \frac{3b^2 - 7b + 2}{2(b^2 - 3b + 1)} u^O_i \leq (b-1)u^O_i$$

Similarly, we derive a modified result for Lemma 28. The Inequality (5.4) in Lemma 28 is correct for any $4 > b > 3$. Since we want to bound n^O_{MMH} by $(b-1)U^O_{\text{MMH}}$, this requires that $b - 2 \leq (b-1)(b-3)$. Therefore, for $b \geq (5+\sqrt{5})/2 \approx 3.62$ we have

$$n^O_i \leq \frac{(b-2)u^O_i}{b-3} \leq (b-1)u^O_i$$

Thus, for $b \geq (5+\sqrt{5})/2$, we can show that $N^O_{\text{MVH}} \leq (b-1)U^O_{\text{MVH}}$ and $N^O_{\text{MMH}} \leq (b-1)U^O_{\text{MMH}}$. Therefore, the intermediate-state mapping has

$$\begin{aligned}
& n^O_{\text{MVH}} + n^O_{\text{MMH}} + n^N_{\text{HVH}} + n^N_{\text{HMH}} \\
\leq & (b-1)U^O_{\text{MVH}} + (b-1)U^O_{\text{MMH}} + U^O_{\text{HVH}} + (b-1)U^N_{\text{HMH}} + U^O_{\text{HMH}} \\
\leq & (b-1)(U^O_{\text{MVH}} + U^O_{\text{MMH}} + U^N_{\text{HVH}} + U^N_{\text{HMH}}) + U^O_{\text{HVH}} + U^O_{\text{HMH}} \\
\leq & (b-1)m/b + m/b \leq m
\end{aligned}$$

The typical-state mapping is similar to Lemma 32, while the critical-state mapping is the same as Lemma 33. By intersecting all the ranges of b, multi-criticality MCFS has a capacity bound of $(5+\sqrt{5})/2 \approx 3.62$. □

Remark: Why is the capacity augmentation bound $2+\sqrt{2}$ for dual-criticality systems and $(5+\sqrt{5})/2$ for systems with 3 or more criticality levels? In general, when a system is in the criticality level Z_i, the tasks at criticality level Z_i exhibit overload behavior, while all tasks with higher criticality levels exhibit nominal behavior. However, dual-criticality systems have a special property that lets us prove a better bound: when the system is in the typical-state, the low-criticality tasks exhibit nominal behavior (instead of overload behavior).

Generalization to more than 3 criticality levels: Consider the system with l criticality levels $\{\text{LO}, Z_2, ..., Z_i, ..., Z_{l-1}, \text{HI}\}$. The classification, virtual deadline and core assignments for a task with criticality Z_i where $1 < i < l$ are identical to medium-criticality tasks shown in Table 5.5 and 5.6, using $b = (5+\sqrt{5})/2$. In particular, tasks with criticality Z_i where $1 < i < l$ are classified into ZiMH and ZiVH tasks, based on their nominal utilization. Since the assignments for ZiMH tasks are the same as MMH tasks and the assignments for ZiVH tasks are the same as MVH tasks, hence we can prove the correctness and capacity augmentation bound similarly.

Note that this bound is not related to the number of criticality levels. This is because in our model each task has two behaviours: the nominal behaviour estimated by the system designer and the overload behavior estimated by the certification authorities. For a task τ_i with criticality Z_i where $1 < i < l$, all tasks with lower criticality levels than τ_i are discarded; all tasks with higher but not highest criticality level Z_j where $i < j < l$ exhibit nominal behavior and have the same nominal core assignment using $b = (5 + \sqrt{5})/2$, so we do not need to distinguish them. In addition, since Lemma 24 and 27 hold for $b = (5 + \sqrt{5})/2$, for criticality level Z_j we have $n^N_{\text{ZjVH}} + n^N_{\text{ZjMH}} \leq U^O_{\text{ZjVH}} + (b-1)U^N_{\text{ZjMH}} + U^O_{\text{ZjMH}}$. Similar to (3) in Theorem 35, the mapping for criticality level Z_i is valid — $n^O_{\text{MVH}} + n^O_{\text{MMH}} + n^N_{\text{ZjVH}} + n^N_{\text{ZjMH}} + n^N_{\text{HVH}} + n^N_{\text{HMH}} \leq m$. Therefore, the capacity augmentation bound remains the same for more than 3 criticality levels.

5.6 Improve MCFS Algorithm for High-Utilization Tasks

In Sections 5.3 and 5.5, we presented the MCFS mapping algorithm for high-utilization tasks that are designed especially for task sets satisfying the conditions from Section 5.3.2 for a capacity augmentation bound of $2 + \sqrt{2}$ and $(5 + \sqrt{5})/2$ for dual- and multi-criticality task sets, respectively. Because of the design of this particular MCFS mapping algorithm, however, for task sets that violate these conditions it may not be able to find a valid mapping, even though there may exist a mapping that can schedule the task sets.

To improve the schedulability of MCFS algorithm, in this section we present the improved MCFS algorithm as **MCFS-Improve** based on heuristics for finding a valid mapping for task sets having even higher utilizations than is indicated by the capacity augmentation bound. For the rest of this section, we use the dual-criticality system as an example. The proposed MCFS-Improve algorithm can be easily extended from dual- to multi-criticality systems.

```
// First, assign virtual deadlines and cores according to the basic MCFS mapping
$b = 2 + \sqrt{2}$
for each task $\tau_i$ in the task set
    if $\tau_i$ is a LH task:
        if $D_i - L_i^N \le 0$: return unschedulable
        $D'_i = D_i$; $n_i^N = \left\lceil \frac{C_i^N - L_i^N}{D_i - L_i^N} \right\rceil$; $n_i^O = 0$
    elseif $\tau_i$ is a HVH task:
        if $D_i - L_i^N - L_i^O \le 0$: return unschedulable
        elseif $\frac{b-2}{b-1} D_i > L_i^O$: $D'_i = \frac{D_i}{b-1}$; $n_i^N = \lfloor u_i^O \rfloor$; $n_i^O = \left\lceil \frac{C_i^O - n_i^N D'_i - L_i^O}{D_i - D'_i - L_i^O} \right\rceil$
        else : $D'_i = \frac{L_i^N D_i}{L_i^N + L_i^O}$; $n_i^N = \left\lceil \frac{C_i^N - L_i^N}{D'_i - L_i^N} \right\rceil$; $n_i^O = \max\left\{ n_i^N, \left\lceil \frac{C_i^O - n_i^N D'_i - L_i^O}{D_i - D'_i - L_i^O} \right\rceil \right\}$
    elseif $\tau_i$ is a HMH task:
        if $D_i - L_i^N - L_i^O \le 0$: return unschedulable
        elseif $\frac{b-2}{b} D_i > L_i^O$: $D'_i = \frac{2}{b} D_i$; $n_i^N = \max\left\{ \left\lceil \frac{C_i^N - L_i^N}{D'_i - L_i^N} \right\rceil, \lceil u_i^O \rceil \right\}$;
        else : $D'_i = \frac{L_i^N D_i}{L_i^N + L_i^O}$; $n_i^N = \left\lceil \frac{C_i^N - L_i^N}{D'_i - L_i^N} \right\rceil$;
        $n_i^O = \max\left\{ n_i^N, \left\lceil \frac{C_i^O - n_i^N D'_i - L_i^O}{D_i - D'_i - L_i^O} \right\rceil \right\}$
```

Figure 5.1: MCFS-Improve mapping algorithm and schedulability test (initialization step)

Why can't MCFS mapping algorithm generate a valid mapping for task sets that do not satisfy conditions of capacity augmentation bounds? This is mainly caused by the fixed parameter b in table 5.3 and 5.6. As discussed in Section 5.4.4, the correctness of MCFS does not rely on a particular b. The particular value of b is chosen to assign roughly b times the total utilization in nominal-state and also b times the total utilization in critical-state. When conditions of capacity augmentation bound of b are satisfied, the total utilizations in both states are less than m/b, so there are enough cores to assign to each task. However, when the conditions are not satisfied, e.g. the total utilization in critical-state is larger than m/b while the total utilization in typical-state is less than m/b, the available cores in the critical-state are not enough. Note that there are extra cores in the typical-state that are not assigned to any task. These cores could be assigned to some tasks in the typical-state, so that the tasks would need less cores in the critical-state.

```
$n^N = \sum_i n_i^N$, $n^O = \sum_i n_i^O$
// Both critical- and typical-state mappings are valid
if $n^N \leq m$ and $n^O \leq m$: return schedulable
// Both critical- and typical-state mappings are not valid
elseif $n^N > m$ and $n^O > m$: return unschedulable
// There are not enough cores for critical-state mapping
elseif $n^N \leq m$ and $n^O > m$:
    while $n^N = \sum_i n_i^N \leq m$:
        $a = \min\{m - n^N, 1\}$
        for each task $\tau_i$ in the task set, where $\tau_i$ is not a LH task:
            $\bar{n}_i^N = n_i^N + a$, $\bar{D}_i' = \frac{C_i^N - L_i^N}{\bar{n}_i^N} + L_i^N$;
            if $\bar{n}_i^N < \frac{C_i^O - L_i^O}{D_i - L_i^O}$: $\bar{n}_i^O = \left\lceil \frac{C_i^O - \bar{n}_i^N \bar{D}_i' - L_i^O}{D_i - \bar{D}_i' - L_i^O} \right\rceil$
            else : $\bar{n}_i^O = \left\lceil \frac{C_i^O - \bar{n}_i^N (\bar{D}_i' - L_i^N) - L_i^O}{D_i - \bar{D}_i' - (L_i^O - L_i^N)} \right\rceil$
            $x_i = n_i^O - \bar{n}_i^O$
        $\tau_i$ is the task with the maximum decrease $x_i$
        for $\tau_i$, update $D_i' = \bar{D}_i'$, $n_i^N = \bar{n}_i^N$, $n_i^O = \bar{n}_i^O$
        if $a == 0$: for each $\tau_i$, update $D_i' = \bar{D}_i'$, $n_i^N = \bar{n}_i^N$; $n_i^O = \bar{n}_i^O$; break
    if $n^O = \sum_i n_i^O \leq m$: return schedulable
    else : return unschedulable
// There are not enough cores for typical-state mapping
elseif $n^N > m$ and $n^O \leq m$:
    while $n^O = \sum_i n_i^O \leq m$:
        $b = \min\{m - n^O, 1\}$
        for each task $\tau_i$ in the task set, where $\tau_i$ is not a LH task:
            $\bar{n}_i^N = n_i^N - 1$, $\bar{D}_i' = \frac{C_i^N - L_i^N}{\bar{n}_i^N} + L_i^N$;
            if $D_i - \bar{D}_i' - L_i^O \leq 0$: $y_i =$MAX, continue
            if $\bar{n}_i^N < \frac{C_i^O - L_i^O}{D_i - L_i^O}$: $\bar{n}_i^O = \left\lceil \frac{C_i^O - \bar{n}_i^N \bar{D}_i' - L_i^O}{D_i - \bar{D}_i' - L_i^O} \right\rceil$
            else : $\bar{n}_i^O = \left\lceil \frac{C_i^O - \bar{n}_i^N (\bar{D}_i' - L_i^N) - L_i^O}{D_i - \bar{D}_i' - (L_i^O - L_i^N)} \right\rceil$
            $y_i = \bar{n}_i^O - n_i^O$
        $\tau_i$ is the task with the minimum increase $y_i$
        if $y_i \leq b$: for $\tau_i$, update $D_i' = \bar{D}_i'$, $n_i^N = \bar{n}_i^N$, $n_i^O = \bar{n}_i^O$
        else : break
    if $n^N = \sum_i n_i^N \leq m$: return schedulable
    else : return unschedulable
```

Figure 5.2: MCFS-Improve mapping algorithm and schedulability test (adjustment step)

The MCFS-Improve algorithm is designed based on this observation. As shown in Figure 5.1, it starts with the original MCFS mapping in table 5.3, with the exception for tasks with critical-path lengths longer than D_i/b. For these tasks, the originally assigned virtual deadline may not be enough to complete the critical-path length in one of the states. Unlike the basic MCFS that deems such tasks unschedulable, MCFS-Improve still tries to calculate a valid virtual deadline in the initialization step. If the initial core assignments in the typical- and critical-state are already valid, then MCFS-Improve admits this task set. Note that for task sets satisfying the conditions of capacity augmentation bound, the virtual deadline and core assignments are the same, so MCFS-Improve also has a capacity augmentation bound of $2 + \sqrt{2}$.

Moreover, for task sets without a valid mapping in the initialization step, MCFS-Improve adjusts the virtual deadline and core assignment according to some heuristics. At a high level, according to whether the number of cores is insufficient in the typical- or critical-state, MCFS-Improve selectively chooses some tasks, and decreases or increases the number of cores assigned in the typical-state. The tasks are chosen greedily, in order to have the maximum impact on balancing the core assignments in both states. For example, when there are not enough cores in the critical-state, MCFS-Improve chooses a task τ_i to increase the number of cores in the typical-state. With extra cores, τ_i is able to complete its nominal work faster, allowing a shorter virtual deadline and hence more time to complete its overload work. Thus, the number of cores required by τ_i in the critical-state could decrease. Task τ_i is selected, such that it has the maximum decrease of the number of cores in the critical-state. By increasing the total cores assigned in typical-state, the total cores assigned in critical-state could eventually be less than m. The adjustment strategy is similar for the case where there are not enough cores in the typical-state.

The correctness of MCFS-Improve algorithm in Figure 5.2 can be derived from the following lemma, which tells us how to calculate the minimum number of cores required in the critical-state, given the core assignment in the typical-state. Note that for LH tasks, the core assignment is already tight for the typical-state and the LH tasks are not assigned any core in the critical-state, so MCFS-Improve does not change the mapping for LH tasks.

Lemma 36 *If a high-criticality high-utilization task τ_i is assigned with n_i^N cores in the typical-state, then it can complete its nominal work by a virtual deadline of D'_i, where $D'_i = \frac{C_i^N - L_i^N}{n_i^N} + L_i^N$. Given n_i^N and D'_i, it only needs n_i^O cores to complete the overload work by deadline D_i, where*

$$n_i^O = \begin{cases} \left\lceil \frac{C_i^O - n_i^N D'_i - L_i^O}{D_i - D'_i - L_i^O} \right\rceil & n_i^N < \frac{C_i^O - L_i^O}{D_i - L_i^O} \\ \left\lceil \frac{C_i^O - n_i^N (D'_i - L_i^N) - L_i^O}{(D_i - D'_i) - (L_i^O - L_i^N)} \right\rceil & n_i^N \geq \frac{C_i^O - L_i^O}{D_i - L_i^O}. \end{cases}$$

Proof. We can easily prove that the virtual deadline is sufficient by applying Lemma 20. For the number of cores assigned in the critical-state, the proof is similar to that of Lemma 26. Let $t^* \leq D'_i$ be the number of complete steps where all cores are busy working and $D'_i - t^* \leq L_i^N$ be the number of incomplete steps where the critical-path length decreases before the transition. Then, at the transition, the job has $C^O - n_i^N t^* - D'_i + t^*$ remaining work and $L_i^O - D'_i + t^*$ remaining critical-path length that must be completed in $D_i - D'_i$ time steps. Therefore, τ_i is guaranteed to complete by the deadline, if τ_i is allocated at least n dedicated cores, where

$$n = \left\lceil \frac{(C^O - n_i^N t^* - D'_i + t^*) - (L_i^O - D'_i + t^*)}{(D_i - D'_i) - (L_i^O - D'_i + t^*)} \right\rceil = \left\lceil \frac{C_i^O - L_i^O - n_i^N t^*}{D_i - L_i^O - t^*} \right\rceil$$

Now consider two cases:

Case 1: If $n_i^N < \frac{C_i^O - L_i^O}{D_i - L_i^O}$, by applying Ineq (1) and $t^* \leq D'_i$, we get

$$n = \left\lceil \frac{C_i^O - L_i^O - n_i^N t^*}{D_i - L_i^O - t^*} \right\rceil \leq \left\lceil \frac{C_i^O - L_i^O - n_i^N D'_i}{D_i - L_i^O - D'_i} \right\rceil = n_i^O$$

Case 2: If $n_i^N \geq \frac{C_i^O - L_i^O}{D_i - L_i^O}$, by applying Ineq (2) and $t^* \geq D_i' - L_i^N$, we get

$$n = \left\lceil \frac{C_i^O - L_i^O - n_i^N t^*}{D_i - L_i^O - t^*} \right\rceil \leq \left\lceil \frac{C_i^O - L_i^O - n_i^N (D_i' - L_i^N)}{D_i - L_i^O - (D_i' - L_i^N)} \right\rceil = n_i^O$$

Combining the two cases gives us the proof. □

Remark: MCFS-Improve algorithm is a greedy algorithm that runs in polynomial time. The numerical experiments conducted in Section 5.9 are consistent with the analysis that MCFS-Improve can admit more task sets than the original MCFS algorithm and it can admit task sets with utilizations much higher than that indicated by the capacity augmentation bound. Note that due to the integer requirement of core assignment as well as the complexity of the calculation, the derived mapping may not be the global optimal mapping. However, when $C_i^O - L_i^O > C_i^N - L_i^N$ and $(C_i^O - L_i^O)(D_i - L_i^O - L_i^N) > (C_i^N - L_i^N)L_i^N$ (both are reasonable constraints on a parallel task) the calculation is approximately convex, so the algorithm may find the close to optimal mapping.

5.7 General Case for Dual-Criticality MCFS

We can generalize MCFS to dual-criticality task systems with both high- and low-utilization tasks (both the nominal utilization and overload utilization are at most 1). The high-utilization tasks are still scheduled in a similar manner as in Section 5.3, while we treat low-utilization tasks as sequential tasks and schedule them using a mixed-criticality multiprocessor scheduler for sequential tasks. In particular, in addition to the three categories from Section 5.3, we have two additional categories specific to low-utilization tasks. **LO-Low (LL)** tasks are low-criticality tasks with low-utilization in nominal behavior, i.e. $u_i^N \leq 1$. **HI-Low-Low (HLL)** tasks are high-criticality tasks with low-utilization in both behaviors, i.e. $u_i^N \leq u_i^O \leq 1$. We denote the set of low-utilization tasks as $\tau_{\text{Seq}} = \{\tau_i \in \tau_{\text{LL}} \cup \tau_{\text{HLL}}\}$ and their total utilizations in nominal and overload behavior as U_{Seq}^N and U_{Seq}^O, respectively.

Note that these tasks are essentially sequential tasks, since they do not require parallel execution to meet their deadlines in either states. Therefore, MCFS can use any existing mixed-criticality multiprocessor scheduler $\mathcal{S}$ for sequential tasks to schedule these tasks. Here, as an example, we assume that we will use MC-Partition [21] to assign these tasks to cores. Say that the total numbers of assigned cores in typical- and critical-state to these tasks is N^N_{Seq} and N^O_{Seq}; unlike MC-Partition, these may be unequal.

At runtime, in the typical-state, high-utilization tasks still execute on their dedicated cores in parallel, while all tasks in τ_{Seq} execute on shared N^N_{Seq} cores. If any HLL task overruns its nominal work, the system transitions to critical-state and *all* LL tasks are immediately discarded. In addition, if $N^O_{\text{Seq}} > N^N_{\text{Seq}}$, some HLL tasks may need to migrate to cores assigned to some additional low-criticality tasks (from set LH). LH tasks may also be discarded in order to acquire N^O_{Seq} total cores for HLL tasks in overload behavior.

Therefore, even though high-utilization tasks never migrate even when the system transitions to critical-state, migration may be required for low-utilization tasks. This is because two or more HLL tasks may share a core in typical state without exceeding the nominal utilization bound of a single core, but their total overload utilization may increase to more than the capacity of a single core. For example, if we choose to use the MC-Partition-UT-0.75 algorithm presented in [21] to schedule low-utilization tasks, the total numbers of assigned cores N^N_{Seq} and N^O_{Seq} in typical- and critical-state could be different. In such cases, some high-criticality low-utilization tasks may migrate when the system transitions from typical- to critical-state.

Correctness and Capacity Augmentation Bound: Recall that tasks in the LL and HLL categories are scheduled using the chosen scheduler $\mathcal{S}$ and are executed sequentially since they have utilization no more than 1 in both states. Therefore, the correctness of this algorithm follows from Section 5.4.4 and from the correctness of the chosen scheduler $\mathcal{S}$ for low-criticality tasks.

The following theorem proves the capacity augmentation bound.

Theorem 37 *For dual-criticality systems with both high- and low-utilization tasks, MCFS has a capacity augmentation bound of $(s+1)m/(m-1)$ where $\frac{1}{s} \geq \frac{1}{1+\sqrt{2}}$ is the utilization bound of the mixed-criticality multiprocessor scheduler $\mathcal{S}$ used by MCFS to schedule low-utilization tasks. For instance using a modified version of MC-partition [21], we get a bound of $11/3 \times m/(m-1) \approx 3.67$ for large m.*

Proof. The proof for capacity augmentation is obtained by simply noticing that all the relevant lemmas for high-utilization tasks, namely Lemmas 22, 24, 27, 25, and 28, work for $s+1 \geq 2+\sqrt{2}$. In addition, Section 5.3 shows that virtual deadlines are valid for any $b > 3$. Here we denote $U^N = U^N_{\text{Seq}} + U^N_{\text{LHi}} + U^N_{\text{HVH}} + U^N_{\text{HMH}} \leq m/b$ and $U^O = U^O_{\text{Seq}} + U^O_{\text{HVH}} + U^O_{\text{HMH}} \leq m/b$.

As $\frac{1}{s}$ is the utilization bound of the chosen scheduler $\mathcal{S}$ for low-utilization tasks, the number of cores assigned to them in nominal and critical-states are

$$N^N_{\text{Seq}} = \lceil sU^N_{\text{Seq}} \rceil < sU^N_{\text{Seq}} + 1 \text{ and } N^O_{\text{Seq}} = \lceil sU^O_{\text{Seq}} \rceil < sU^N_{\text{Seq}} + 1$$

As $b = (s+1)/(1-1/m)$, for typical-state mapping we can derive

$$\begin{aligned}
& N^N_{\text{Seq}} + N^N_{\text{LHi}} + N^N_{\text{HVH}} + N^N_{\text{HMH}} \\
\leq & sU^N_{\text{Seq}} + 1 + sU^N_{\text{LHi}} + U^O_{\text{HVH}} + sU^N_{\text{HMH}} + U^O_{\text{HMH}} \\
\leq & sU^N + 1 + U^O \leq sm/b + 1 + U^O \\
\leq & \left((1-1/m)b - 1\right) m/b + 1 + m/b = m
\end{aligned}$$

For critical-state mapping, we can also derive that

$$N^O_{\text{Seq}} + N^O_{\text{HVH}} + N^O_{\text{HMH}} \leq sU^O + 1 \leq sm/b + 1 \leq m$$

For instance, since MC-Partition-UT-0.75 has a utilization bound[4] of $\frac{3}{8}$, MCFS using MC-Partition has a capacity augmentation bound approaching $\frac{11}{3}$, when m is large and $m/(m-1) \approx 1$. □

In principle, the *MCFS scheduler*, can handle low-utilization tasks by utilizing any mixed-criticality multiprocessor scheduling strategy for sequential tasks. However, as far as we know, there is no prior work that shows a utilization bound for systems with more than 2 criticality levels; therefore, for such task sets we cannot prove a capacity-augmentation bound and restrict ourselves to tasks which exhibit high-utilization at least in their overload state.

5.8 Implementation of a MCFS Runtime System

We demonstrate the applicability of MCFS, as described in Section 5.3, by implementing a real-time MCFS runtime system for a dual-criticality system with high-utilization tasks. This reference implementation supports parallel programs written in OpenMP [125]. It uses Linux with the RT_PREEMPT patch as the underlying RTOS and the OpenMP parallel concurrency platform to manage threads and assign work at runtime.

Three key requirements are derived for the MCFS runtime: (1) the system must detect when any high-criticality task has overrun its virtual deadline; (2) it must modify the core allocation to give more cores to high-criticality tasks in the event of a virtual deadline miss; and (3) since the number of active threads in the system fluctuates with its criticality state, it must provide a state-aware concurrency mechanism to facilitate parallel programming — i.e., a state-aware barrier.

Overrun Detection: The MCFS runtime system detects that a high-criticality task overruns its virtual deadline via Linux's `timer_create` and `timer_settime` API. These timers are set and disarmed at the start and end of each period by each high-criticality task while

[4]As proved in [21], it has a speedup of $\frac{8}{3}$ compared to 100% utilization.

in the typical-state, so expiration only occurs in the event of an overrun. Timer expirations are delivered via signals and signal handlers. To make sure that the timer expiration is noticed promptly, kernel `ksoftirq` threads are given higher real-time priority than all other threads.[5]

Core Reallocation: A key requirement of MCFS is to increase the allocation of cores to a high-criticality task when it exceeds its virtual deadline, by taking cores away from low-criticality tasks. This is accomplished in four parts. (1) At the start of execution, each high-criticality task τ_i creates the maximum number of threads it would need in the critical-state (n_i^O). Each low criticality task creates n_i^N threads. (2) When the runtime system initializes (in typical-state), only n_i^N threads are awake for each task and they are pinned to distinct cores[6]. (3) The remaining $n_i^O - n_i^N$ threads of high-criticality tasks are put to sleep with the `FUTEX_WAIT` system call, while also pinned to their cores (which may be shared with a low-criticality task). These threads sleep at a priority higher than any low-criticality thread on the same core. (4) When a job of high-criticality task τ_i overruns its virtual deadline, its sleeping threads are awoken with `FUTEX_WAKE` and they preempt the low-criticality thread on the same core and begin executing.

Note that the set of cores assigned by the typical-state mapping to τ_i is a subset of the cores assigned by the critical-state mapping; therefore, the system needs no migration for the high-utilization tasks.

In this design, the threads of each task must be activated and deactivated each period via the OpenMP directive `#pragma omp parallel`. Thus, this approach of maintaining a pool of unused, high-criticality threads does impose an additional overhead on the system, even if it never transitions into critical-state, due to these activations and deactivations. However,

[5]This could be a potential source of criticality inversion; however, in our system, this is not a major source of overhead. The alternative, thread-context notification, can be subject to unsuitably long delays.

[6]In order to pin threads to cores, before the task execution, we use an initial `#pragma omp parallel` directive where individual threads make a call to Linux's `sched_setaffinity` and pin themselves to the assigned cores.

these overheads are only imposed on low-criticality tasks by high-criticality tasks, so there is no criticality inversion.

When a job of high-criticality task τ_i overruns its virtual deadline and preempts the low-criticality tasks on the shared cores, the current jobs of these low-criticality tasks may continue to execute when the higher-priority threads from high-criticality tasks are idling. If, however, the start times of the these low-criticality jobs are already later than their absolute deadlines, such jobs are dropped voluntarily by low-criticality tasks. Therefore, when the system is able to recover from critical-state to typical-state, there is little backlog of low-criticality jobs and the future arriving jobs of the same task are able to resume normal execution. Note that for systems that can tolerate tardiness for low-criticality jobs, an alternative implementation would not drop these backlogged jobs.

The primary reason for allowing current low-criticality jobs to run at a lower priority instead of directly killing the threads of these job is to avoid the cost of creating new threads during system operation, but it also allows the low-criticality threads to make progress on a best-effort basis. Note that since we allow low-criticality threads to continue executing after a mode transition has occurred, they will continue to interfere with high-criticality threads through cache pollution, resource contention, and other effects. Even so, allowing low-criticality threads to continue progressing seems appropriate for a soft real-time system. The other options are to kill these processes or to suspend them, but we do not investigate these options here.

Since high-criticality tasks do not share cores in MCFS, if a high-criticality task receives a timer signal, indicating that it has overrun its virtual deadline, it does not initiate a system-wide mode switch. Instead, it simply wakes up its sleeping $n_i^O - n_i^N$ threads and acquires the necessary additional cores from a subset of low-criticality tasks. If a low-criticality task overruns its deadline, it need not do anything. This natural implementation leads to graceful degradation since not all low-criticality tasks are discarded on entering critical-state.

Latency due to mode transition: The most important factor to optimize for ensuring the safe operation of high-criticality tasks is the *high-criticality activation latency*— the delay between when a mode transition is detected and when the additional $n_i^O - n_i^N$ high-criticality threads that were sleeping in the typical mode wake up and are ready to perform work. We measure this by inducing a mode transition at a fixed time, and the extra threads perform a time-stamp as soon as they wake up. The difference between the mode switch time and the latest time-stamp gives the latency. This latency was very low in general and increases with the increasing number of threads. We varied the number of awoken threads from one to fourteen, measuring the latency 400 times for each setting, and the maximum observed latency was 84 microseconds.

Note that this mode transition latency may occur only once for each high-criticality job in the critical-state. To incorporate it into schedulability analysis, we subtract it from the deadline of each high-criticality task.

Impact of high-criticality tasks on low-criticality tasks: As discussed in Section 5.8, low-criticality tasks incur overhead when they share a core with a high-criticality task. The low-criticality task is subject to interruption by high-criticality threads that must sleep and awake at the start and end of every period, which involves two context switches, the start and end of a `#pragma omp parallel` directive, and interactions with a Linux futex. We compare the wall-clock execution time of the low-criticality task with the Linux clock source `CLOCK_THREAD_CPUTIME_ID` to infer the total amount of time the low-criticality task was preempted. The maximum observed overhead was relatively high at 1555 microseconds per preemption. In our system, it was important to incorporate this overhead into the schedulability test to ensure that low-criticality tasks meet their deadlines. This overhead is only incurred when a high-criticality task's sleeping thread is sharing a core with a low-criticality task in the typical-state. In addition, the preemption only occurs once per period of the high-criticality task. Therefore, we can calculate the maximum number of preemptions and subtract the appropriate time from the low-criticality task's deadline. This allows the scheduler to assign the correct number of cores to low-criticality tasks.

Discussion:: For tasks in our experiments on the simple prototype platform, we were able to mitigate the effect of this overhead by incorporating it into the schedulability test. For tasks at smaller time scales, this overhead may be unacceptably high. It is mostly attributed to the cost of entering and exiting the `#pragma omp parallel` each period as shown in Figure 5.3. For a reference system like we have described here, the choice of including the `parallel` directive within the periodic invocation greatly simplifies programming and reasoning about the system, as well as allows the user to use existing parallel programs with little modification, but the overhead may be unsuitably high for practical systems. In a traditional OpenMP program, the `parallel` directive would be used once or just a few times— calling it once every period exposes an important limitation of this standard parallel concurrency platform when used in real-time systems.

```
perioidic_iteration(){
    #pragma omp parallel
    {
    if(typical_state && high_crit_task)
        sleep_extra_threads()

    // Do parallel program
    #pragma omp for schedule(dynamic) nowait
    for (j = 0; j < num_strands; ++j)
    {
        // Perform work
        busy_work()
    }

    mc_barrier_wait()
    wake_extra_threads()
    }
}
```

Figure 5.3: Periodic Task Invocation Psuedocode

```
// Called asynchronously by signal handler
barrier_state_switch()
    needs_switch = true

check_needs_updating()
    if ( needs_switch )
        atomically_claim_switcher()
        if ( switcher )
            verify_barrier_inactive()
            update_barrier_count()
            needs_switch = false
            release_spinwaiters()
        else spinwait()

mc_barrier_wait()
    check_needs_updating()
    do_barrier_wait()
```

Figure 5.4: Mode Aware Barrier Psuedocode

State-Aware Barrier Implementation: One side-effect of mixed-criticality model for parallel tasks is that counting-based thread synchronization methods such as barriers will not work properly as the number of active threads fluctuates. For OpenMP in particular, if some threads in an OpenMP team are sleeping (as in our implementation), the implicit barrier at the end of each `#pragma omp for` loop may deadlock, if the sleeping threads never arrive.

We address this by removing the implicit barrier with the OpenMP clause `nowait`, as shown in Figure 5.3 and implementing a state-aware barrier shown in Figure 5.4, which operates as follows. When a task begins a transition, its signal handler sets a variable indicating that the barrier needs updating before waking the extra high-criticality threads. The next thread to encounter the barrier checks this variable and claims responsibility for updating with an atomic compare-and-swap on a boolean flag. Other threads arriving after that will spin-wait. The update thread will then verify that the barrier is not currently being modified by any thread that arrived before the transition, spin-waiting otherwise, and finally will increment the barrier count when it is safe to do so. It then releases any threads that are spin-waiting so that they may proceed through the barrier.

This imposes a small, constant overhead every time a thread accesses the barrier, since threads must check to see if the barrier needs updating. However, it allows us to use the same barrier in both states, and the barrier can be updated even if some threads are currently waiting on the barrier. Without such an arrangement, the transition overhead could be unbounded, since the additional $n_i^O - n_i^N$ high-criticality threads could not be released while any barrier was in an indeterminate state.

Recover from critical-state to typical-state: As observed at the end of Section 5.3, the MCFS scheduling theory naturally supports tasks that may transition between the typical-state and critical-state many times over the life of the system. This is desirable as it allows low-criticality tasks to continue executing on a best-effort basis. Otherwise, a high-criticality task transitioning into critical-state would permanently impair any low-criticality task it happened to share a processor core with, even if the conditions that lead to the state transition were transient.

Reverting to typical-state is straightforward compared with transitioning into the critical-state, because the MCFS theory allows this to happen at a time of our choosing and not in response to any external event. Thus, a particularly convenient time for this to occur is outside the execution of any job of the task, because the task's team of parallel threads is not active during those times. Modifying the system while a parallel computation is underway is the major source of complexity for the critical-state transition and is what requires the complex core reallocation and state-aware barrier mechanisms that are discussed above.

Affecting the transition to typical-state requires resetting the state-aware barrier and reducing the number of threads that will participate in future job invocations. Since this process occurs outside the execution of any job, it is guaranteed that the barrier is not in use and that no parallel threads are active. Thus there are no concurrency issues to resolve, and reversion is accomplished without synchronization. In particular, the state-aware barrier is reconfigured to expect the number of threads that should be active in the typical-state (i.e. a modified version of `update_barrier_count()` from Figure 5.4 may be called without protection). Second, a global flag is set which indicates to the critical-state threads that they should sleep with `FUTEX_WAIT` upon activation rather than immediately participating.

Under the MCFS theory this reversion may be performed as often as the finish of each individual job that has entered the critical-state. In effect, the critical-state transition occurs on a per-job basis rather than a per-task basis, and all new jobs start in the typical-state but may transition to the critical-state as needed, allowing for very fine grained control over the system criticality and providing the minimum interruption to low-criticality tasks. Such low-criticality tasks operate on a best-effort basis but are not guaranteed in the face of interference from a task in the critical-state. In Section 5.9, we construct benchmark task set to test and evaluate the recovery to typical-state feature of MCFS runtime system.

5.9 Numerical Evaluation

We first conduct an extensive study comparing the schedulability tests of basic MCFS, MCFS-Improve and capacity augmentation bound of $2+\sqrt{2}$ on a wide range of parameters. In particular, we investigate the impact of the following parameters on the schedulabilities of MCFS and MCFS-Improve: the number of cores m, the total nominal utilization U^N, the total overload utilization U^O, and the maximum ratio p_{max} of the overload critical-path length over period. For each of these settings, we ran the MCFS schedulability test on 1000 task sets and show the fraction of schedulable task sets admitted by MCFS and MCFS-Improve.

Task Set Generation

First, we explain how we generate task sets for the evaluation. We vary the number of cores m from 16, 32, 64 and 128. Given m cores, we vary both the total nominal utilization U^N and overload utilization U^O of task sets from 1 to m.

With the desired total nominal and overload utilization U^N and U^O of each setting, we first add high-criticality tasks until U^O is reached and then add low-criticality tasks until U^N is reached. To limit high-criticality tasks' total nominal utilization to U^N, for each setting we calculated a maximum "nominal over overload utilization ratio" $r_{max} = U^N/U^O$. High-criticality tasks' nominal over overload utilizations will not exceed the maximum ratio.

To evaluate the impact of critical-path length, we also vary the maximum ratio p_{max} of the overload critical-path length over period as $p_{max} = [\frac{0.25}{b}, \frac{0.5}{b}, \frac{0.75}{b}, \frac{1}{b}, \frac{1.25}{b}, \frac{1.5}{b}, \frac{1.75}{b}, \frac{2}{b}, \frac{2.25}{b}, \frac{2.5}{b}, \frac{2.75}{b}, \frac{3}{b}]$, where $b = 2 + \sqrt{2}$. Note that the capacity augmentation bound of b requires that $p_{max} \leq \frac{1}{b}$. The larger the p_{max}, the harder to schedule the task set under the family of MCFS algorithms.

Note that for the two MCFS algorithms, the schedulability only depends on the work and critical-path length of tasks rather than their parallel structures (whether they are synchronous or DAG tasks). Therefore, for the numerical experiments we directly generate these parameters for each task.

Our task set generation proceeds as follows.

(1) Criticality z_i: 50% high-criticality and 50% low-criticality.

(2) Nominal and overload utilization ratio r_i for high-criticality task: uniformly from 0.01 and a calculate r_{max}; This ratio r_i for low-criticality task is always 1.

(3) Overload utilization u_i^O: randomly chosen from a log normal distribution with mean of $1 + \sqrt{m}/3$.

(4) Nominal utilization $u_i^N = r_i u_i^O$.

(5) Implicit deadline D_i: uniformly from 100ms to 1000ms.

(6) Max overload critical-path length L': 40%, 50%, 70% and 100% of $D_i p_{max}$, with probability of 0.4, 0.3, 0.2 and 0.1.

(7) Overload critical-path length L_i^O: uniformly chosen from $[0, L']$.

(8) Nominal critical-path length $L_i^N = r_i L_i^O$.

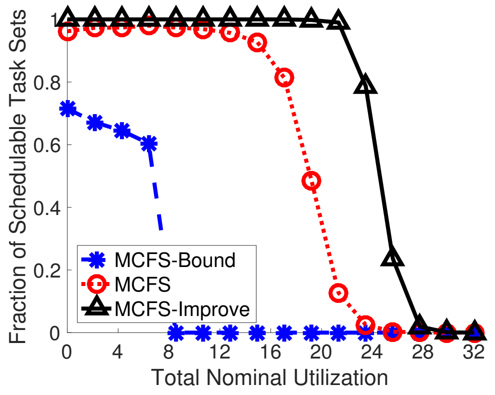
With the above parameters, we can calculate the nominal and overload work, which are used in the schedulability tests.

Impact of varying U^N and U^O

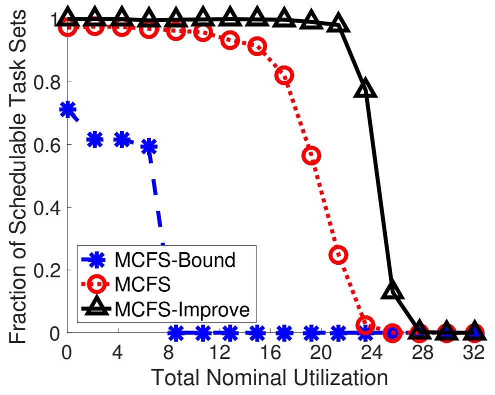
We first evaluate the schedulabilities of MCFS-Bound MCFS and MCFS-Improve on a 32-core setting with $p_{max} = \frac{1.5}{b}$ in Figure 5.5 and 5.6. In particular, we select 6 representative total overload utilizations U^O ranging from $[12.5\%m, 25\%m, 37.5\%m, 50\%m, 62.5\%m, 75\%m]$, as shown in Figure 5.5(a) to 5.5(f). In each figure, we increase the total nominal utilization and plot the fraction of schedulable task sets of MCFS-Bound, MCFS and MCFS-Improve.

By the definition of capacity augmentation bound $2 + \sqrt{2}$, the task sets that are deemed schedulable by ***MCFS-Bound*** are those with *both* L_i^N and L_i^O no more than $\frac{D_i}{2+\sqrt{2}}$ and with *both* U^N and U^O no more than $\frac{m}{2+\sqrt{2}} \approx 29\%m \approx 9.4$ on 32 cores. So only some of the task sets in the left one third of Figure 5.5(a), 5.5(b), 5.6(a) and 5.6(b) are schedulable by MCFS-Bound. Apparently, both MCFS and MCFS-Improve can admit many more task sets than is indicated by the capacity augmentation bound.

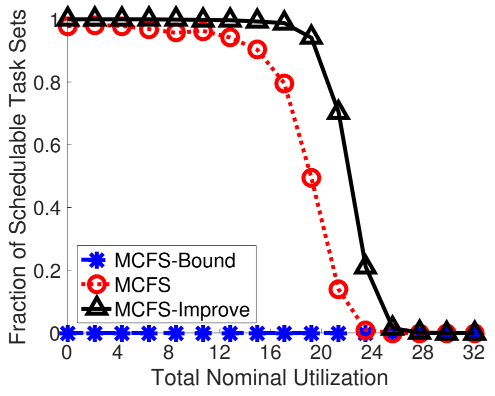
The trend of the Figure 5.5 and 5.6 shows that both MCFS and MCFS-Improve can schedule more task sets, when the total utilizations are lower. Moreover, we can observe that MCFS-Improve can schedule more task sets than MCFS, especially when the load of the system is high. For example, MCFS-Improve admits almost twice the number of task sets admitted by the basic MCFS schedulability test for $U^O = 75\%m$ in Figure 5.5(f) and 5.6(e); in Figure 5.6(f), almost no task sets are schedulable under basic MCFS, while MCFS-Improve can still schedule most task sets with overload utilization up to 8.



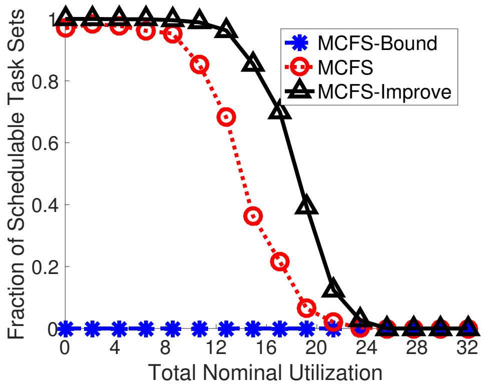
(a) For $U^O = 12.5\%m$ and $m = 32$.



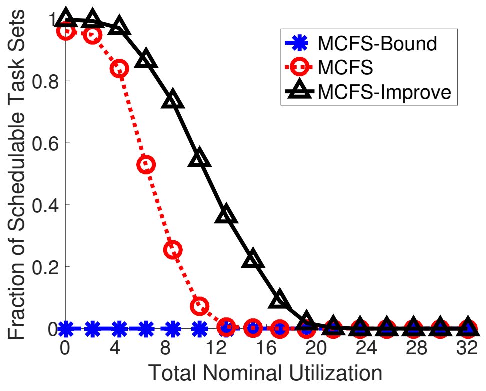
(b) For $U^O = 25\%m$ and $m = 32$.



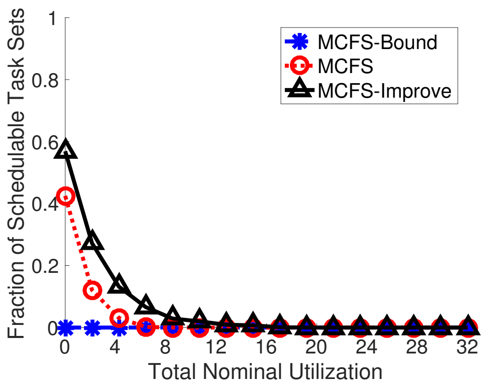
(c) For $U^O = 37.5\%m$ and $m = 32$.



(d) For $U^O = 50\%m$ and $m = 32$.

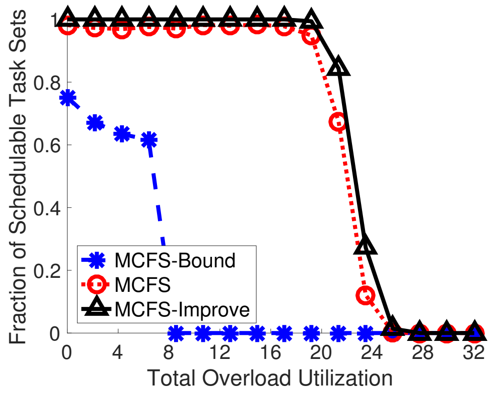


(e) For $U^O = 62.5\%m$ and $m = 32$.

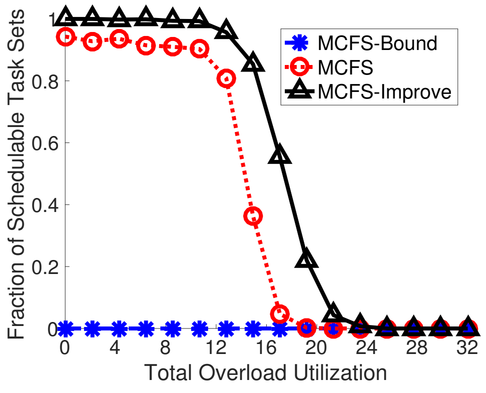


(f) For $U^O = 75\%m$ and $m = 32$.

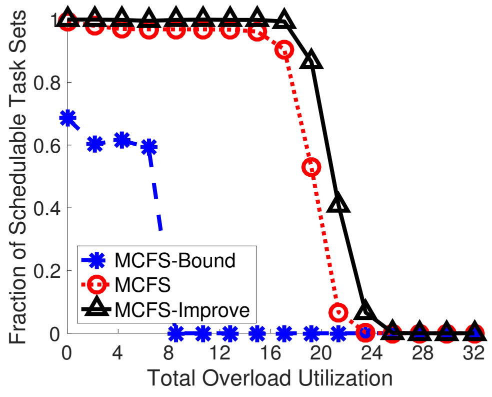
Figure 5.5: Fraction of schedulable task sets of MCFS-Bound vs. MCFS vs. MCFS-Improve on *32 cores* setting with $\mathbf{p_{max}} = \frac{\mathbf{1.5}}{\mathbf{b}}$. From (a) to (f), figures show the results for total *nominal utilizations* ranging from [12.5%m, 25%m, 37.5%m, 50%m, 62.5%m, 75%m], respectively. Each figure shows the results for increasing total *overload utilizations*.



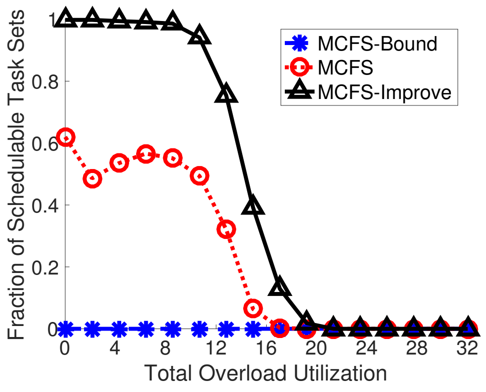
(a) For $U^N = 12.5\%m$ and $m = 32$.



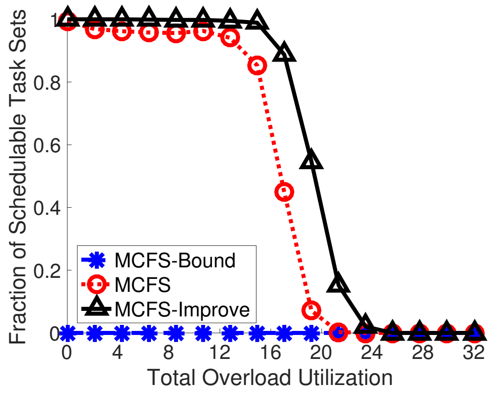
(d) For $U^N = 50\%m$ and $m = 32$.



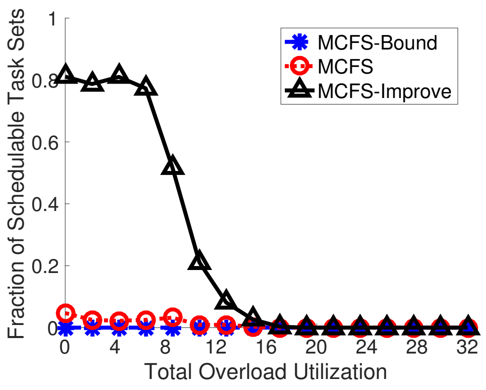
(b) For $U^N = 25\%m$ and $m = 32$.



(e) For $U^N = 62.5\%m$ and $m = 32$.

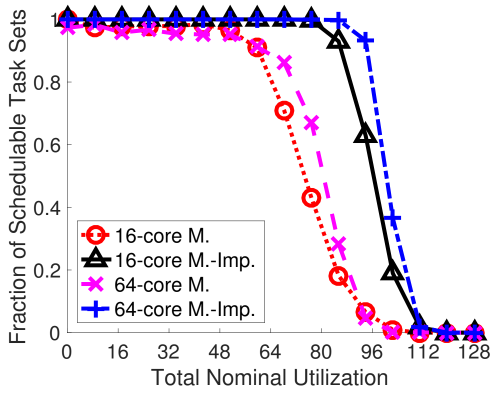


(c) For $U^N = 37.5\%m$ and $m = 32$.

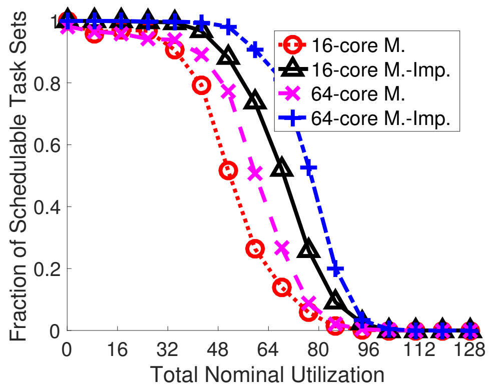


(f) For $U^N = 75\%m$ and $m = 32$.

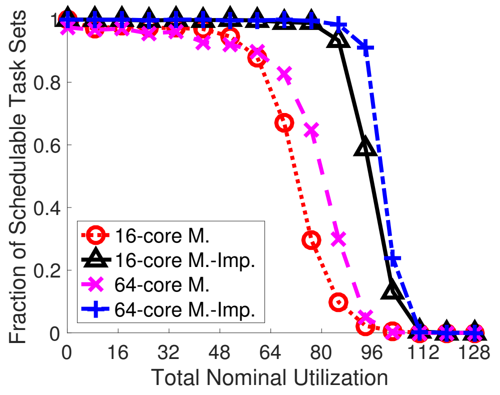
Figure 5.6: Fraction of schedulable task sets of MCFS-Bound vs. MCFS vs. MCFS-Improve on *32 cores* setting with $\mathbf{p_{max} = \frac{1.5}{b}}$. From (a) to (f), figures show the results for total *overload utilizations* ranging from [12.5%m, 25%m, 37.5%m, 50%m, 62.5%m, 75%m], respectively. Each figure shows the results for increasing total *nominal utilizations*.



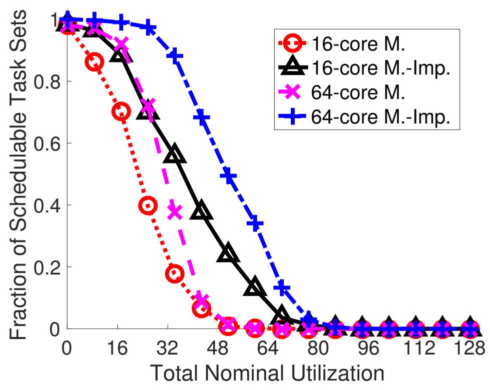
(a) For $U^O = 12.5\%m$.



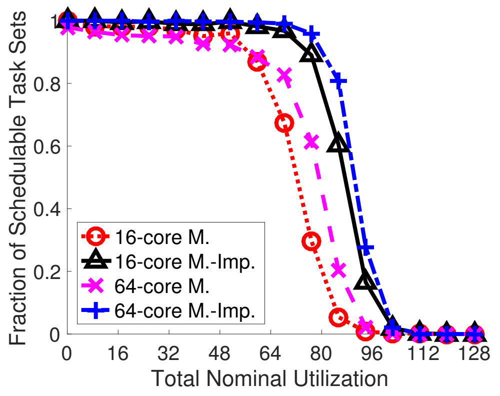
(d) For $U^O = 50\%m$.



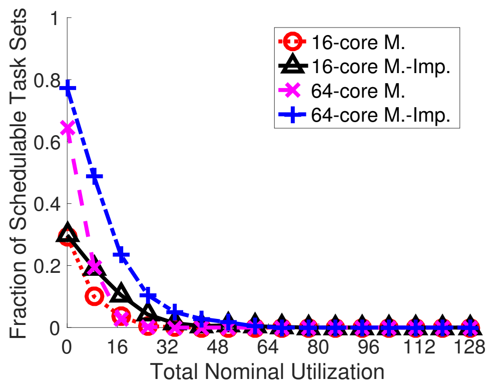
(b) For $U^O = 25\%m$.



(e) For $U^O = 62.5\%m$.

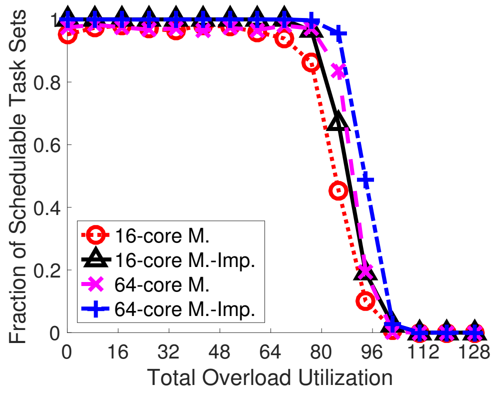


(c) For $U^O = 37.5\%m$.

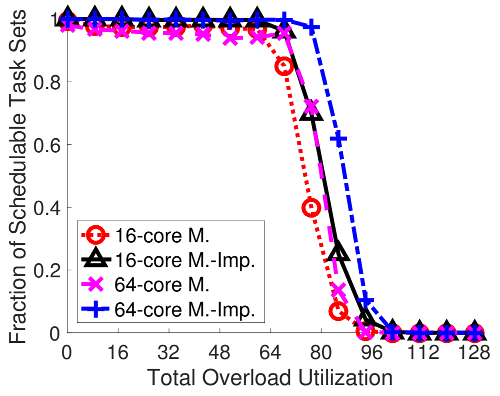


(f) For $U^O = 75\%m$.

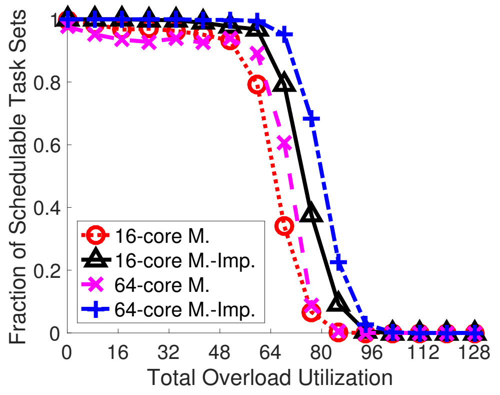
Figure 5.7: Fraction of schedulable task sets of MCFS (labeled "M.") vs. MCFS-Improve (labeled "M.-Imp.") on *16 cores* (labeled "16-core") vs. *64 cores* (labeled "64-core") with $\mathbf{p_{max}} = \frac{1.5}{b}$. From (a) to (f), figures show the results for varying total *overload utilizations*. Each figure shows the results for increasing total *nominal utilizations*.



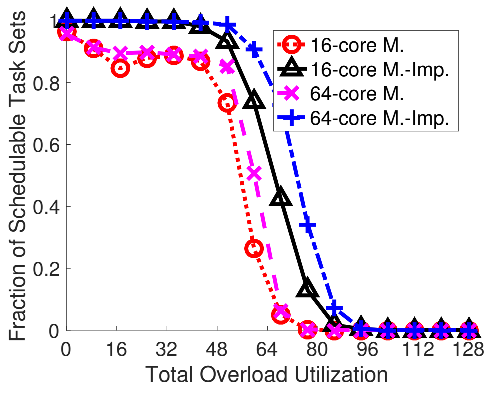
(a) For $U^N = 12.5\%m$.



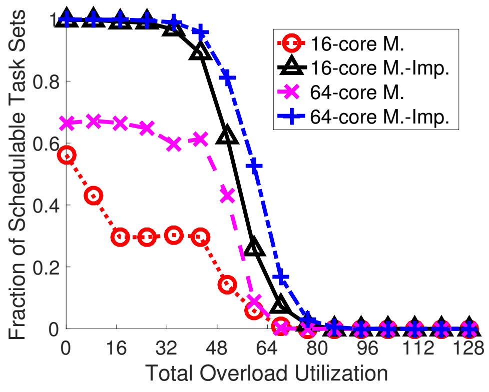
(b) For $U^N = 25\%m$.



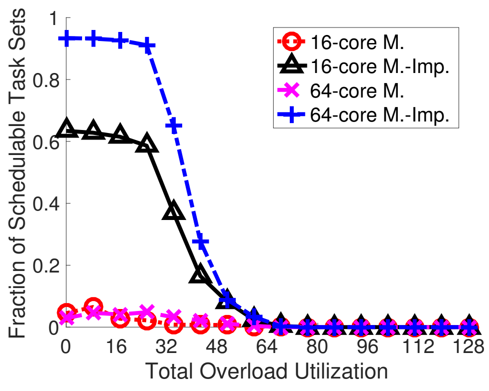
(c) For $U^N = 37.5\%m$.



(d) For $U^N = 50\%m$.



(e) For $U^N = 62.5\%m$.



(f) For $U^N = 75\%m$.

Figure 5.8: Fraction of schedulable task sets of MCFS (labeled "M.") vs. MCFS-Improve (labeled "M.-Imp.") on *16 cores* (labeled "16-core") vs. *64 cores* (labeled "64-core") with $\mathrm{p_{max}} = \frac{1.5}{\mathrm{b}}$. From (a) to (f), figures show the results for varying total *nominal utilizations*. Each figure shows the results for increasing total *overload utilizations*.

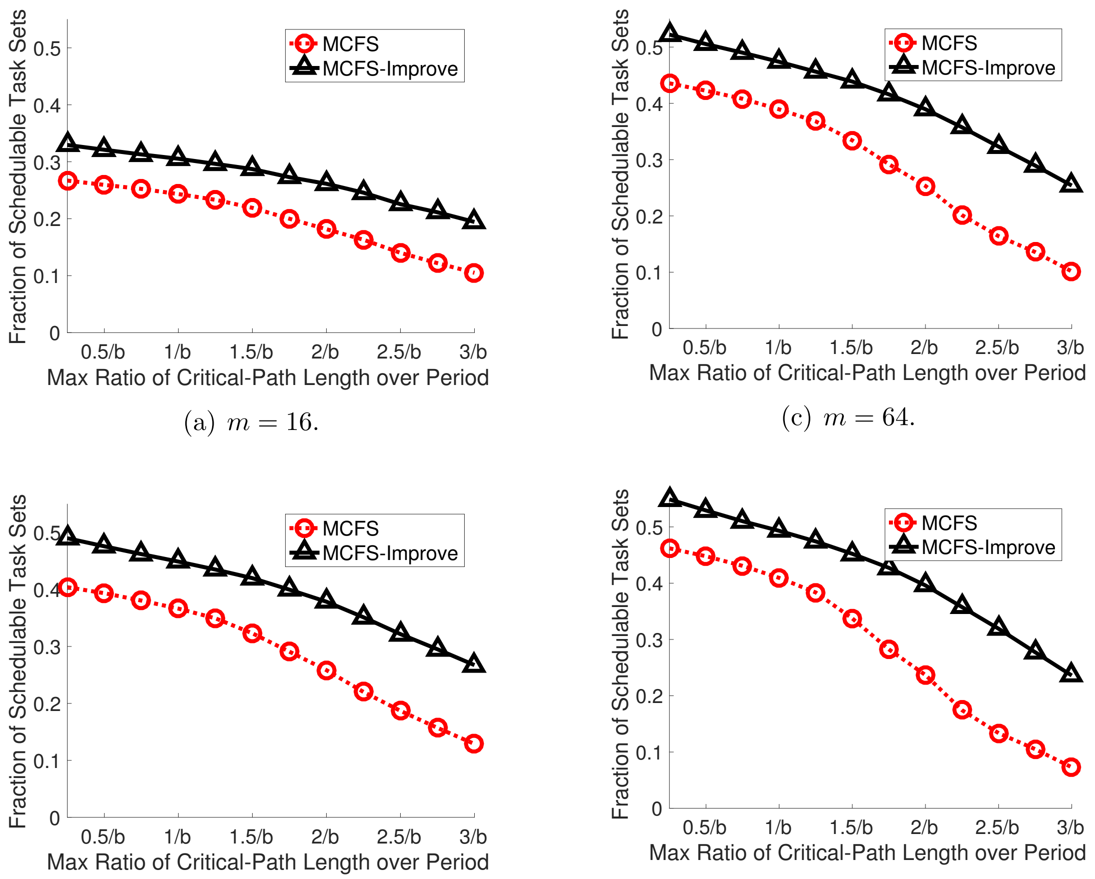
Impact of varying m

The trend of the schedulability comparison between MCFS and MCFS-Improve for 32 cores is similar to those of 16, 64 and 128 cores. In Figure 5.7 and 5.8, we can compare the fraction of schedulable task sets between the 16-core and 64-core setting on task sets with varying loads. We can observe that when the number of cores increases, both MCFS and MCFS-Improve can admit more task sets. Moreover, MCFS-Improve significantly improves over MCFS when the nominal utilization is high and the overload utilization is low in Figures 5.8(e) and 5.8(f). This is because in these cases there are many low-criticality tasks requiring many dedicated cores in the typical state. Hence, there are less cores remains for high-criticality tasks in typical state, while all cores are available for their low total overload utilizations in the critical state. For such cases, MCFS-Improve is able to find better virtual deadlines to decrease the number of cores assigned in the typical state and increase the number of cores assigned in the critical state, balancing the total core assignments in both states.

Impact of critical-path length

In Figure 5.9, we present the results with varying number of cores m and maximum ratio p_{max} of critical-path length over period. Each data point in each figure shows the average fraction of schedulable task sets of all the settings with varying nominal and overload utilizations. For example, we have randomly generated and evaluated 256,000 task sets for the setting $m = 64$ and $p_{max} = \frac{2.5}{b} \approx 0.8$. MCFS can schedule 16.4% of these task sets, while MCFS-Improve can schedule 32.3% task sets. In this setting, MCFS-Improve has a relative improvement almost 100% over that of MCFS.

The ratio p_{max} has large impact on the schedulability of the family of MCFS algorithms, since it affects the parallelism of tasks. This is because given the same nominal and overload work, tasks with lower p_{max} have higher parallelism. In addition, when p_{max} decreases, tasks have more slack before the implicit deadline to complete their parallel work, so they



(a) $m = 16$.

(c) $m = 64$.

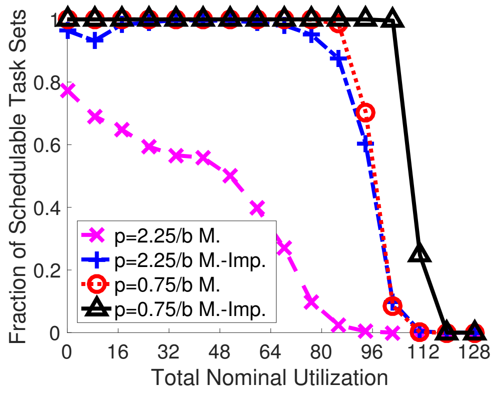
(b) $m = 32$.

(d) $m = 128$.

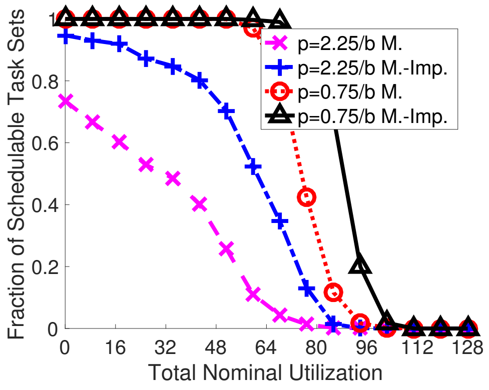
Figure 5.9: Fraction of schedulable task sets of MCFS (dotted line) vs. MCFS-Improve (solid line) with varying number of cores. Each data point in each figure shows the average fraction of schedulable task sets of all the different settings (varying nominal and overload utilizations), given m and maximum ratio p_{max} of critical-path length over period.

require less dedicated cores. Therefore, MCFS and MCFS-Improve can schedule more task sets when p_{max} is lower. In addition, the improvement of MCFS-Improve increases with increasing p_{max}. This is because MCFS-Improve can adjust the virtual deadline to balance the slacks before the virtual deadline and real deadline. This balances the core assignments in typical and critical states, so it results in considerably better schedulability.

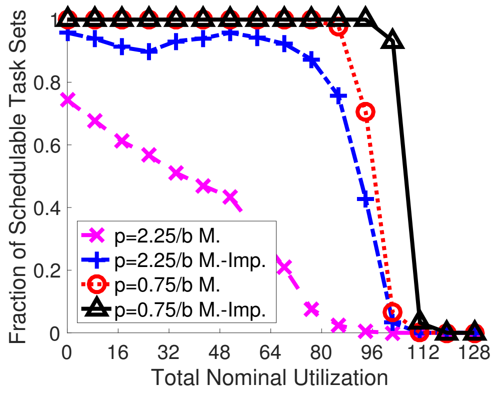
To have a detailed look at the influences of the maximum critical-path length ratio p_{max}, we plot the results for 128-core with $p_{max} = \frac{0.75}{b}$ and $p_{max} = \frac{2.25}{b}$ in Figure 5.10 and Figure 5.11. Again, we can observe that MCFS and MCFS-Improve can admit more task



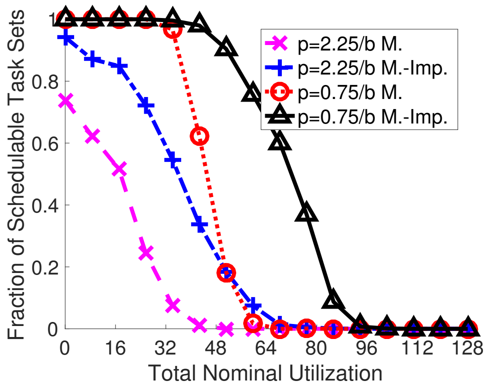
(a) For $U^O = 12.5\%m$ and $m = 128$.



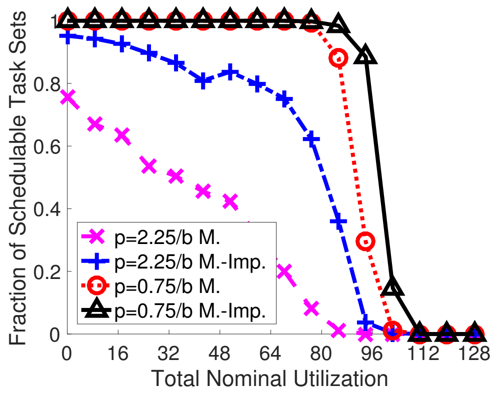
(d) For $U^O = 50\%m$ and $m = 128$.



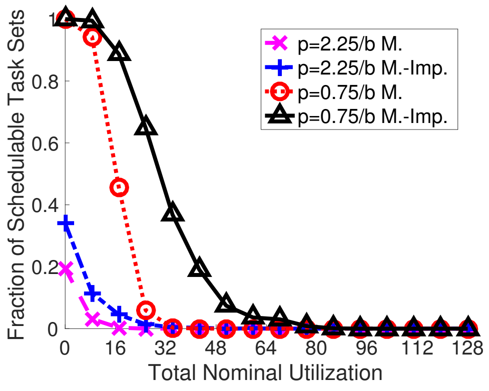
(b) For $U^O = 25\%m$ and $m = 128$.



(e) For $U^O = 62.5\%m$ and $m = 128$.

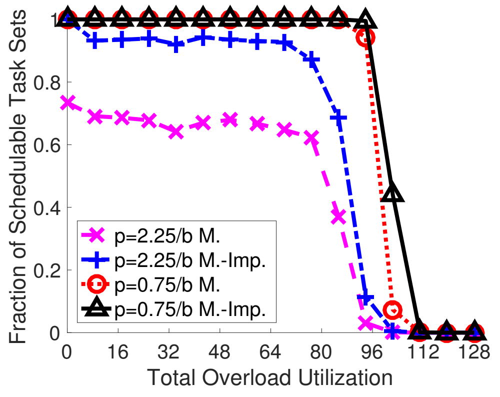


(c) For $U^O = 37.5\%m$ and $m = 128$.

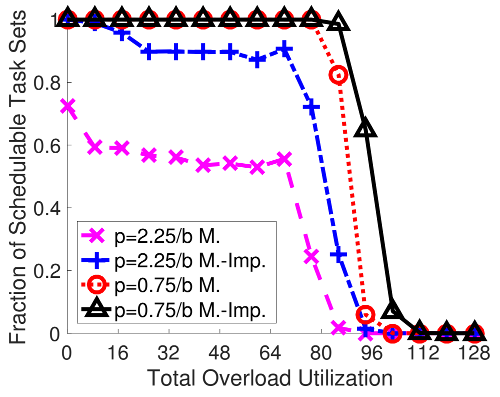


(f) For $U^O = 75\%m$ and $m = 128$.

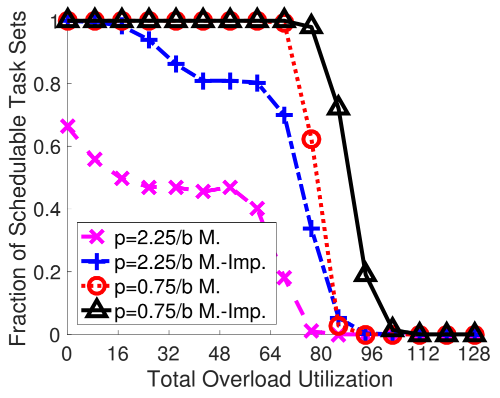
Figure 5.10: Fraction of schedulable task sets of MCFS (labeled "M.") vs. MCFS-Improve (labeled "M.-Imp.") on *128 cores* with $p_{max} = \frac{0.75}{b}$ (labeled "p=0.75/b") vs. $p_{max} = \frac{2.25}{b}$ (labeled "p=2.25/b"). From (a) to (f), figures show the results for varying total *overload utilizations*. Each figure shows the results for increasing total *nominal utilizations*.



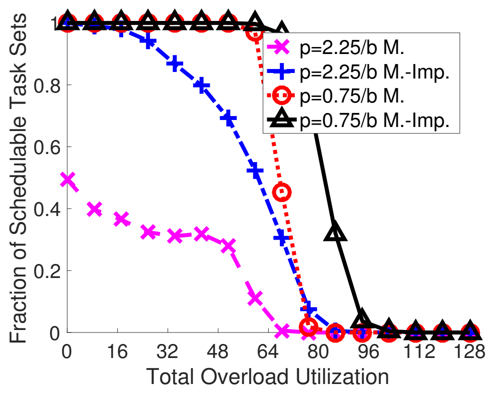
(a) For $U^N = 12.5\% m$ and $m = 128$.



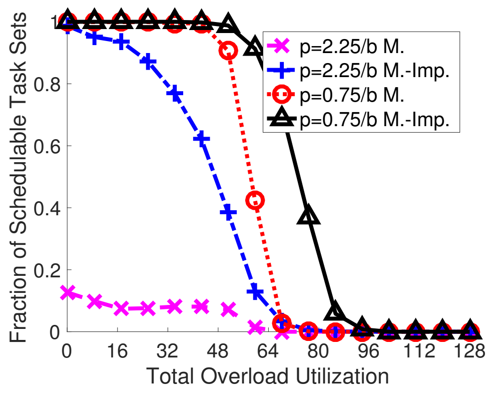
(b) For $U^N = 25\% m$ and $m = 128$.



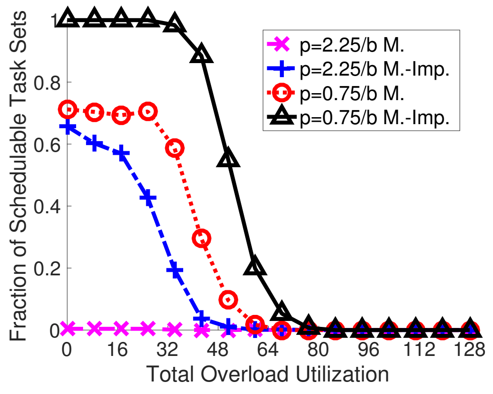
(c) For $U^N = 37.5\% m$ and $m = 128$.



(d) For $U^N = 50\% m$ and $m = 128$.



(e) For $U^N = 62.5\% m$ and $m = 128$.



(f) For $U^N = 75\% m$ and $m = 128$.

Figure 5.11: Fraction of schedulable task sets of MCFS (labeled "M.") vs. MCFS-Improve (labeled "M.-Imp.") on *128 cores* with $p_{max} = \frac{0.75}{b}$ (labeled "p=0.75/b") vs. $p_{max} = \frac{2.25}{b}$ (labeled "p=2.25/b"). From (a) to (f), figures show the results for varying total *nominal utilizations*. Each figure shows the results for increasing total *overload utilizations*.

sets when p_{max} is lower. However, p_{max} affect the performance of MCFS more than MCFS-Improve. By comparing these figures with previous figures, we can conclude that MCFS-Improve significantly improves over MCFS when tasks' critical-path lengths are relatively long, the number of cores is large, and/or the total nominal utilization is high.

5.10 Empirical Evaluation

We also evaluate our implementation of the MCFS runtime system described in Section 5.8 using synthetic workloads written in OpenMP. Experiments were conducted on a 16-core machine composed of two Intel Xeon E5-2687W processors (each with 8 cores). When running the experiments, we reserved two cores for operating system services, leaving 14 experimental cores. Linux with RT_PREEMPT patch version 4.1.7-rt8 was the underlying RTOS. For each setting, we randomly generate 100 task sets, each of which runs for 5 minutes — 300× the maximum period.

Task Set Generation

Now we explain how we generate task sets for the empirical evaluation. In these empirical experiments, the number of cores m is 14. We construct a task set by keep adding randomly generated tasks until MCFS schedulability test cannot admit any more tasks. Tasks are either high- or low-criticality with equal probability.

Note that the synthetic tasks in the empirical experiments are written in OpenMP. Each task has a sequence of parallel for loops, or segments. Each iteration of a segment is called a strand. We generate a task by first randomly choosing a desired overload critical-path length L', and then keep adding randomly generated segments until L' is reached.

The task parameters generation process is similar to [138]. To generate tasks with large parallelism, we fix the maximum ratio p_{max} of the overload critical-path length over period as $p_{max} = \frac{1}{2(2+\sqrt{2})}$.

(1) Criticality z_i: 50% high-criticality and 50% low-criticality.

(2) Nominal and overload utilization ratio r_i for high-criticality tasks: uniformly from $[0.025, 0.25]$; The ratio r_i for low-criticality tasks is 1.

(3) Implicit deadline D_i: uniformly from 100ms to 1000ms.

(4) Max overload critical-path length L': 40%, 50%, 70% and 100% of $D_i p_{max}$, with probability of 0.4, 0.3, 0.2 and 0.1.

(5) Number of strands of a segment $s_{i,j}$: randomly chosen from a log normal distribution with mean of $1 + \sqrt{m}/3$.

(6) Overload length of strands of a segment $t^O_{i,j}$: randomly chosen from a log normal distribution with mean of 5ms.

(7) Nominal length of strands of a segment $t^N_{i,j} = r_i t^O_{i,j}$.

With above parameters, we can calculate the nominal and overload work and critical-path length, which are used in MCFS schedulability test.

Stress Testing

We first conducted experiments to stress test the performance of the MCFS runtime system in both typical- and critical-states. In the typical-state stress testing, both high- and low-criticality task, execute exactly their worst-case *nominal* work and critical-path length. Experimental results are consistent with the correctness condition; no mode transition occurs and all high- and low-criticality tasks meet all their deadlines. In the critical-state stress testing, each task executes exactly its worst-case *overload* work and critical-path length. Again, in this worst case behavior, the result is also consistent with the correctness condition; every high-criticality task successfully transitions to critical-state and has no deadline miss. Some low-criticality tasks are preempted by high-criticality tasks, suspend some of their jobs and hence have deadline misses, which is allowed in critical-state.

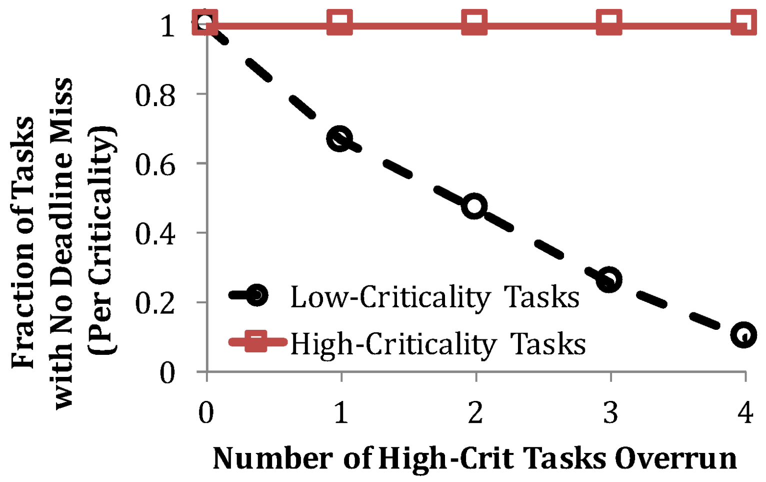


Figure 5.12: Fraction of tasks with no deadline miss, for the sets of tasks with high- and low-criticality, respectively, when increasing the number of high-criticality tasks that overrun their nominal parameters.

Graceful Degradation

The mixed-criticality correctness condition allows us to discard all low-criticality tasks as soon as any task misses its virtual deadline and the system transitions to critical-state. However, the MCFS need not do so as discussed in Section 5.3.3. Figure 5.12 demonstrates that the MCFS runtime system can continue to run many low-criticality tasks even after some high-criticality jobs transitions to critical-state. Here, we pick task sets with at least 4 high-criticality tasks. For each set, we run 5 experiments: either 0, 1, 2, 3 or 4 high-criticality tasks execute for their overload parameters and the remaining for their nominal parameters. We plot the fraction of tasks with no deadline miss. We can see that all high-criticality tasks always meet their deadlines. In contrast, the low-criticality task performance does not drop abruptly to zero as soon as the transition occurs. For instance, when only 1 high-criticality task overruns, only about 33% low-criticality tasks miss their deadlines.

Recovery from Critical-State to Typical-State

As discussed in Section 5.3, high-criticality tasks may optionally revert to the typical-state under the MCFS theory. Otherwise the transition to critical-state would be permanent

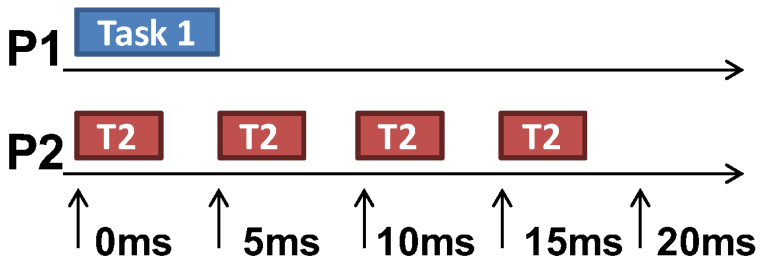


Figure 5.13: One hyper-period of the experimental task system where Task 1 only requires 5ms of computational time.

and any low-criticality tasks sharing a processor would be permanently impaired. Here we construct a task system to illustrate the capability of the MCFS runtime system to recover from critical-state to typical-state, as shown below.

Table 5.7: Task Set Parameters

Task	C_i^N	C_i^O	D_i	D_i'	S^N	S^O
τ_1	5ms	20ms	20ms	10ms	1	1,2
τ_2	3ms	-	5ms	-	2	-

The task τ_1 is a high-criticality task constructed with the `rand()` function so that approximately 20% of the time it will require 20ms of computational effort, but the remainder of the time it will only require 5ms. Thus, roughly one-fifth of the time jobs of this task overruns its virtual deadline at 10ms, trigger a transition to critical-state, and involve the shared processor to help finish its computation on time. Task τ_1 is assigned with a single core (core 1) in the typical-state and it requires one additional core (core 2) in the critical-state. The computation (dense matrix multiplication) of the task is embarrassingly parallel, so, once activated, both threads will contribute nearly equal amounts of computational effort. Task τ_2 is a low-criticality task, which is assigned to core 2 in the typical-state. Hence, core 2 is shared among task τ_1 and τ_2.

The one hyper-period of the typical and critical situations are depicted in Figure 5.13 and Figure 5.14, respectively. If the system was not capable of recovering from the critical- to typical-state, task τ_1 would trigger a transition nearly immediately and task τ_2 would miss approximately half of its deadlines. However, with state recovery enabled, task τ_2 should

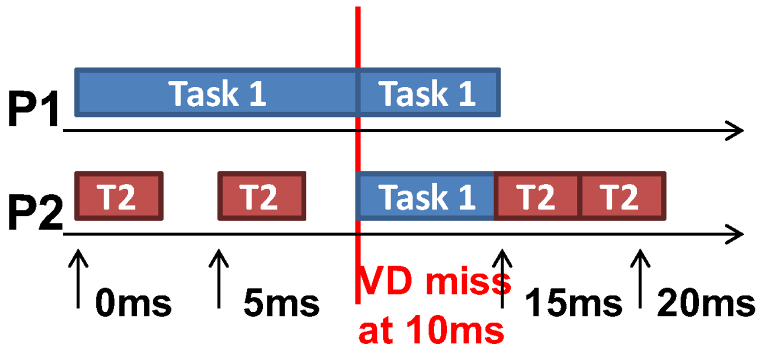


Figure 5.14: One hyper-period of the experimental task system where Task 1 requires 20ms of computational time, necessitating a transition to critical-state.

miss two deadlines for each overrun of τ_1, but recover in those hyper-periods where τ_1 runs entirely in the typical-state. Indeed, when executed this is exactly what was found.

The task system was run for 100,000 hyper-periods. Task τ_1 missed zero deadlines but entered the critical state 21,087 times. Task τ_2 missed 42,174 deadlines out of 400,000, i.e., approximately 10.5% of the time. As confirmation, we also configured task τ_1 so that it always required 20ms of execution time and thus would always enter the critical-state. As expected, task τ_2 missed exactly half of its deadlines.

Chapter 6

Federated Scheduling for Stochastic Parallel Real-time Tasks

Different applications have different real-time requirements and characteristics. Some have a strict constraint that all deadlines must be met and **hard real-time** guarantees should be provided; others have **soft real-time** constraints and can tolerate tardiness as long as it can be bounded. In this chapter, we explore the soft real-time performance of federated scheduling and address average-case workloads instead of worst-case ones. In particular, we consider **stochastic tasks**, which may have large variability in their parameters; the worst-case execution time could be orders of magnitude larger than the mean execution time. In this case, the system may suffer significant utilization loss if we use only worst-case execution time when analyzing schedulability. Instead, we could use the mean and variance of task parameters for analysis. For these tasks, our schedulers guarantee that the expected tardiness of tasks is bounded.

For stochastic parallel real-time tasks with soft real-time constraints, we choose to analyze the federated scheduling because of two reasons. First, as Chapter 4 has shown that federated scheduling has the best known capacity augmentation bound of 2 for hard real-time parallel tasks, it is interesting to analyze its performance for the average case. Second, the federated scheduling is a generalization of partitioned scheduling to parallel tasks. It assigns dedicated cores to each high-utilization task, which makes each high-utilization task being isolated from any interference from other tasks. Therefore, we could analyze each of them individually and directly apply result from single server queueing theory to bound the expected tardiness.

Note that to guarantee hard real-time schedulability, worst-case values are used for task parameters. In constrast, stochastic tasks are based on average-case workload (formally defined in Section 6.2). In this case, we use **bounded expected tardiness** as the schedulability criterion. We define a **stochastic capacity augmentation bound** that uses expected values for utilization and critical-path length to provide bounded expected tardiness. In contrast to using a hard real-time bound, it allows the system to be over-utilized in the worst case, as long as in average total workload is less than the bound.

Section 6.3 presents the stochastic federated scheduling strategy and expected tardiness calculation, and proves that expected tardiness is bounded. Section 6.3.4 present one federated mapping algorithm that guarantees bounded expected tardiness and provide the same capacity augmentation bound 2 for stochastic tasks which is proved in Section 6.4.

6.1 Related Work on Soft Real-Time Scheduling

Real-time multiprocessor scheduling for tasks with *worst-case* task parameters has been studied extensively (see Section 2.4 for an introduction). In this Section, we survey the related work considering soft real-time guarantees for tasks.

Most prior work on bounded tardiness (and other soft real-time guarantees) considers sequential tasks with worst-case parameters [58]. For these tasks, an earliest-pseudo-deadline-first scheduler [143] and GEDF [59, 68] both provide bounded tardiness with no utilization loss; these results were generalized to many global schedulers [110]. Lateness guarantees also have been studied for GEDF-like scheduling [69]. For parallel tasks, Liu et al. [114] for the first time provide a soft real-time response time analysis for GEDF.

For stochastic analysis, there is some prior work on sequential stochastic tasks. For a resource reservation scheduler, a lower bound on the probability of deadline misses was derived in [126]. For multiprocessor scheduling, [122] shows that GEDF guarantees bounded tardiness to sequential tasks if the total expected utilization is smaller than the number of cores. We use this result directly in our algorithms and analysis to guarantee bounded tardiness to low-utilization tasks. There also has been some work on stochastic analysis of a system via Markov processes or approximation [60, 116].

6.2 System Model for Stochastic Parallel Real-Time Tasks

In this section, we formalize the **stochastic task model** in which execution time and critical-path length are described using probabilistic distributions, which is consistent with the task model for sequential tasks in existing work on stochastic real-time analysis [122]. We also define the capacity augmentation bound for stochastic tasks with soft real-time tardiness constraint. Throughout this chapter, we use the calligraphic letters to represent random variables.

Like ordinary real-time tasks, stochastic tasks have a fixed relative deadline D_i ($= P_i$, the period, for implicit deadline tasks). However, each stochastic task is described using its **stochastic work** $\mathcal{C}_i$ — total execution time on 1 core, and **stochastic critical-path length** $\mathcal{L}_i$ — execution time when it is running on a machine with an in infinite number of cores. Note that both $\mathcal{C}_i$ and $\mathcal{L}_i$ are random variables.

In this chapter, the internal structure of each DAG task is not assumed or used to derive the schedulability analysis. The execution time of each node from the same task could vary in different jobs (execution instances), which would result in varying execution times and critical-path lengths. Moreover, the internal structure of each job from the same task could also be different each time. For example, a parallel for-loop in a program could have different numbers of iterations given different inputs, resulting in a different DAG structure.

We assume that the expectations $\mathrm{E}\left[\mathcal{C}_i\right]$ and $\mathrm{E}\left[\mathcal{L}_i\right]$ of these random variables are known. Given $\mathrm{E}\left[\mathcal{C}_i\right]$ and $\mathrm{E}\left[\mathcal{L}_i\right]$, we can calculate the *expected utilization* of a stochastic task τ_i as $\mathrm{E}\left[\mathcal{U}_i\right] = \mathrm{E}\left[\mathcal{C}_i\right]/D_i$, and the total expected utilization of the entire task set as $\sum_i \mathrm{E}\left[\mathcal{U}_i\right]$.

We now specify a few additional parameters that are needed only if we wish to calculate an upper bound on the tardiness. First, for all tasks, we must know the standard deviations $\delta_{\mathcal{C}_i}$ and $\delta_{\mathcal{L}_i}$ of the execution time and the critical-path length. Second, for low-utilization tasks, we need the finite worst-case execution time $\widehat{c}_i$ for calculating tardiness. Finally, for high-utilization tasks, we need the covariance $\sigma(\mathcal{C}_i, \mathcal{L}_i)$ between work and critical-path length.

The exact distributions of $\mathcal{C}_i$ and $\mathcal{L}_i$ are not explicitly required in all three schedulability tests. Our linear-time algorithm can calculate mappings that provide bounded tardiness using just these parameters. With the distributions, another algorithm can generate potentially better mappings.

In addition, for analysis purposes, we define some job specific parameters: $c_{i,j}$ is the actual execution time of job j of task i and $l_{i,j}$ is its actual critical-path length; these are drawn from distributions $\mathcal{C}_i$ and $\mathcal{L}_i$ respectively.

We say that the *release time* of job j of task i is $r_{i,j}$ and its *response time* (or *completion time*) is $t_{i,j}$. *Tardiness* $T_{i,j}$ of job $\tau_{i,j}$ is defined as $\max(0, t_{i,j} - D_i)$. Tardiness $\mathcal{T}_i$ of a task τ_i is also a random variable; $\mathrm{E}[\mathcal{T}_i]$ is its expected value.

We now define the capacity augmentation bound for stochastic tasks. In particular, we consider the schedulability criterion of **bounded expected tardiness**; that is, a task set τ is deemed *schedulable* by a scheduling algorithm $\mathcal{S}$ if the expected tardiness of each task is guaranteed to be bounded under $\mathcal{S}$.

Definition 5 *A scheduling algorithm $\mathcal{S}$ provides a **stochastic capacity augmentation bound** of b if, given m cores, $\mathcal{S}$ can guarantee bounded expected tardiness to any task set τ as long as it satisfies the following **conditions:***

$$\textit{Total available cores, } m \geq b \sum \mathrm{E}[\mathcal{U}_i] \tag{6.1}$$

$$\textit{For each task, } D_i \geq b(\mathrm{E}[\mathcal{L}_i] + \epsilon_i) \tag{6.2}$$

where ϵ_i is 0 if the variances of $\mathcal{C}_i$ and $\mathcal{L}_i$ are 0, and is an arbitrarily small positive constant otherwise.

Note that when $\mathcal{C}_i$ and $\mathcal{L}_i$ are deterministic, the variance of $\mathcal{C}_i$ and $\mathcal{L}_i$ is 0, so $\epsilon_i = 0$ and the definition of stochastic capacity augmentation bound reduces to the definition for hard real-time constraints based on worst-case task parameters.

6.3 Stochastic Federated Scheduling Guarantees Bounded Tardiness

In this section, we first describe stochastic federated scheduling; Then we prove that if federated scheduling algorithm can produce a valid mapping, then it guarantees bounded expected tardiness; Finally, we calculate the expected tardiness.

6.3.1 Stochastic Federated Scheduling Strategy

Just like the corresponding federated scheduling strategy for hard real-time tasks, the stochastic federated scheduling strategy classifies tasks into two sets: τ_{high} contains all **high-utilization tasks** — tasks with expected utilization at least 1 ($\mathrm{E}[\mathcal{U}_i] \geq 1$), and τ_{low} contains all the remaining **low-utilization tasks**. The federated scheduling strategy works in two stages:

1. Given a task set τ, a **mapping algorithm** either **admits** τ and outputs a core assignment, or declares that it cannot guarantee schedulability of τ. Different mapping algorithms differ in the assignment of n_i dedicated cores to each high-utilization task τ_i, but $n_i > \frac{\mathrm{E}[\mathcal{C}_i]-\mathrm{E}[\mathcal{L}_i]}{D_i-\mathrm{E}[\mathcal{L}_i]}$ is always required. All low-utilization tasks share the remaining $n_{\text{low}} = m - \sum_{\tau_i \in \tau_{high}} n_i$ cores. Each mapping algorithm only admits a task set if $n_{\text{low}} > \sum_{\tau_i \in \tau_{\text{low}}} \mathrm{E}[\mathcal{U}_i]$ always holds.

2. Once the mapping is done, the scheduling is straightforward. The high-utilization tasks are scheduled on their dedicated cores using a greedy (work-conserving) scheduler. The

low-utilization tasks are scheduled and executed sequentially on the remaining cluster of cores via a GEDF scheduler.

Note that we chose GEDF to schedule low-utilization tasks, because of an existing result that shows that GEDF provides bounded tardiness to sequential stochastic tasks [122]; we can apply this result directly to low-utilization tasks since they are executed sequentially by our federated scheduler. Other multiprocessor scheduling algorithms can be used only if they provide guarantees of bounded tardiness for sequential tasks.

6.3.2 Mapping Algorithms Guarantee Bounded Tardiness

We first analyze high-utilization tasks. Since each of them has dedicated cores and does not suffer any interference from other tasks, we can analyze each task τ_i individually. We use the following result from queueing theory [98] which indicates that if the service time of jobs is less than the inter-arrival time, then the expected waiting time is bounded.

Lemma 38 [KING70] *For a D/G/1 queue, customers arrive with minimum inter-arrival time Y, and the service time $\mathcal{X}$ is a distribution with mean $\mathrm{E}[\mathcal{X}]$ and variance $\delta_{\mathcal{X}}^2$. If $\mathrm{E}[\mathcal{X}] < Y$, then the queue is stable and the expected waiting time $\mathcal{W}$ is bounded $\mathrm{E}[\mathcal{W}] \le \frac{\delta_{\mathcal{X}}^2}{2(Y-\mathrm{E}[\mathcal{X}])}$.*

In our context, for each high-utilization task, jobs are the customers; the inter-arrival time is $Y = D_i$ $(= P_i)$; the response time $\mathcal{X} = t_{i,j}$ is the service time for job j of task τ_i. For a high-utilization job $\tau_{i,j}$, its tardiness $T_{i,j}$ depends on its response time $t_{i,j}$, the tardiness $T_{i,j-1}$ of previous job $\tau_{i,j-1}$ and deadline D_i. In particular, we have $T_{i,j} \le \max\{0, T_{i,j-1} + t_{i,j} - D_i\}$. Therefore, the waiting time $\mathcal{W}$ is a bound on the tardiness $\mathcal{T}$.

For a greedy scheduler on n_i cores, we can easily bound the finish time $t_{i,j}$.

Lemma 39 *If a job $J_{i,j}$ executes by itself under a greedy scheduler on n_i identical cores and it takes $t_{i,j}$ time to finish its execution, then $t_{i,j} \le (c_{i,j} + (n_i - 1)l_{i,j})/n_i$.*

Thus the response time for a job is bounded by $(c_{i,j} + (n_i - 1)l_{i,j})/n_i$. Using properties of mean and variance, we get

$$\mathrm{E}\left[\mathcal{X}\right] = \mathrm{E}\left[t_{i,j}\right] \leq (\mathrm{E}\left[\mathcal{C}_i\right] + (n_i - 1)\mathrm{E}\left[\mathcal{L}_i\right])/ni \tag{6.3}$$

$$\delta^2_{\mathcal{X}} = \delta^2_{t_{i,j}} \leq \delta^2_{\mathcal{L}_i}((n_i - 1)/n_i)^2 + \delta^2_{\mathcal{C}_i}/n_i^2 + 2\sigma(\mathcal{L}_i, \mathcal{C}_i)(n_i - 1)/n_i^2 \tag{6.4}$$

Note that Lemma 38 states that if $\mathrm{E}\left[\mathcal{X}\right] < Y$, then the queue is stable and the tardiness is bounded. Therefore, to prove the bounded expected tardiness of high-utilization task, we only need to prove $\mathrm{E}\left[\mathcal{X}\right] = (\mathrm{E}\left[\mathcal{C}_i\right] + (n_i - 1)\mathrm{E}\left[\mathcal{L}_i\right])/n_i < D_i = Y$.

Theorem 40 *A mapping algorithm for stochastic federated scheduling guarantees bounded tardiness to high-utilization task τ_i, if the assigned number of cores $n_i > \frac{\mathrm{E}[\mathcal{C}_i] - \mathrm{E}[\mathcal{L}_i]}{D_i - \mathrm{E}[\mathcal{L}_i]}$.*

Proof. We first prove $(\mathrm{E}\left[\mathcal{C}_i\right] + (n_i - 1)\mathrm{E}\left[\mathcal{L}_i\right])/n_i < D_i$.

$$\begin{aligned} & D_i n_i - (n_i - 1)\mathrm{E}\left[\mathcal{L}_i\right] = n_i(D_i - \mathrm{E}\left[\mathcal{L}_i\right]) + \mathrm{E}\left[\mathcal{L}_i\right] \\ > \; & \frac{\mathrm{E}\left[\mathcal{C}_i\right] - \mathrm{E}\left[\mathcal{L}_i\right]}{D_i - \mathrm{E}\left[\mathcal{L}_i\right]}(D_i - \mathrm{E}\left[\mathcal{L}_i\right]) + \mathrm{E}\left[\mathcal{L}_i\right] \\ > \; & \mathrm{E}\left[\mathcal{C}_i\right] \end{aligned}$$

Hence, $\mathrm{E}\left[\mathcal{X}\right] = \mathrm{E}\left[t_{i,j}\right] = (\mathrm{E}\left[\mathcal{C}_i\right] + (n_i - 1)\mathrm{E}\left[\mathcal{L}_i\right])/n_i < D_i = Y$ and by Lemma 38 the tardiness of τ_i is bounded. □

In the stochastic federated scheduling strategy, $n_i > \frac{\mathrm{E}[\mathcal{C}_i] - \mathrm{E}[\mathcal{L}_i]}{D_i - \mathrm{E}[\mathcal{L}_i]}$ is always required for any mapping algorithm. We will show later that for the proposed mapping algorithm, this is indeed satisfied for each high-utilization task.

Now we analyze the tardiness of low-utilization tasks, since they share n_{low} cores and are executed sequentially using GEDF scheduler. In [122], the following Lemma has been established.

Lemma 41 [Mills10] *If a set of sequential tasks τ_{low} is scheduled on n_{low} cores using GEDF and $n_{low} > \sum_{\tau_i \in \tau_{low}} \mathrm{E}[\mathcal{U}_i]$, then the expected tardiness of each task is bounded.*

Since the different mapping algorithms only admit a task set if $\mathrm{E}[\mathcal{U}_{\text{low}}] = \sum_{\tau_i \in \tau_{\text{low}}} \mathrm{E}[\mathcal{U}_i] < n_{\text{low}}$ and then schedule these tasks using GEDF, we can conclude that the expected tardiness of low-utilization tasks is also bounded.

Any task set that the mapping algorithm admits can be scheduled while guaranteeing bounded expected tardiness; hence, the mapping algorithm serves as a **schedulability test**.

6.3.3 Calculating Expected Tardiness

Here, we explain how the tardiness is calculated. Even though all the mapping algorithms provide bounded expected tardiness, the actual (upper bound on) tardiness can be different, because the corresponding core assignments (n_i for each high-utilization task and n_{low} for all low-utilization tasks) are different.

Note that from Section 6.3.4, we can see that for the BASIC and FAIR mapping algorithms, the tardiness calculation is not necessary for producing core assignments. It is only needed in ILP mapping or to get the actual expected tardiness.

Tardiness of High-Utilization Tasks

For each high-utilization tasks with n_i assigned dedicated cores, by Lemma 38 and Inequality (6.4), the bounded expected tardiness is:

$$\mathrm{E}[\mathcal{T}_i] \leq \frac{\delta^2_{\mathcal{X}}}{2(Y - \mathrm{E}[\mathcal{X}])} \leq \frac{\delta^2_{\mathcal{L}_i}(n_i-1)^2/n_i^2 + \delta^2_{\mathcal{C}_i}/n_i^2 + 2\sigma(\mathcal{L}_i, \mathcal{C}_i)(n_i-1)/n_i^2}{2(D_i - (\mathrm{E}[\mathcal{L}_i](n_i-1) + \mathrm{E}[\mathcal{C}_i])/n_i)} \tag{6.5}$$

Tardiness of Low-Utilization Tasks

Since low-utilization tasks are executed sequentially using GEDF, we can use the linear-programming procedure described in [122] directly.

We first restate a couple of lemmas from [122] in our terminology. The first lemma bounds the tardiness of a hypothetical processor-sharing (**PS**) scheduler which always guarantees an execution rate of $\hat{u}_i$ (henceforth called the **PS rate allocation**) to each task τ_i.

Lemma 42 **[Mills10]** *For a given PS rate allocation such that* $\mathrm{E}[\mathcal{U}_i] \leq \hat{u}_i \leq 1$ *and* $\sum \mathrm{E}[\mathcal{U}_i] \leq n_{low}$*, the PS scheduler has a bounded tardiness* $\mathrm{E}[\mathcal{F}_i] \leq \frac{\delta^2_{\mathcal{C}_i}/\hat{u}_i^2}{2(D_i - \mathrm{E}[\mathcal{C}_i]/\hat{u}_i)}$.

Using this tardiness bound, they then provide a bound on the tardiness provided by GEDF for low-utilization tasks.

Lemma 43 **[Mills10]** *For low-utilization tasks scheduled by a GEDF scheduler on* n_{low} *cores, the expected tardiness of each task* $\mathrm{E}[\mathcal{T}_i] \leq \mathrm{E}[\mathcal{F}_i] + \frac{\eta + n_{low}M}{n_{low} - v} + \hat{c}_i$*, where* $\mathrm{E}[\mathcal{F}_i]$ *is the expected tardiness of a hypothetical PS scheduler,* $\hat{c}_i$ *is the worst-case execution time of the task,* η *is the sum of the* $n_{low} - 1$ *largest* $\hat{c}_i$*,* M *is the maximum tardiness in PS, and* v *is the sum of* $n_{low} - 1$ *largest assigned* $\hat{u}_i$ *in PS.*

All the parameters except $\mathrm{E}[\mathcal{F}_i]$ are known or measurable (and bounded). In order to calculate $\mathrm{E}[\mathcal{F}_i]$, we must calculate the PS rate allocation $\hat{u}_i$ for each task τ_i.

As we will show in Section 6.3.4, for the BASIC mapping, there exists a simple calculation of $\hat{u}_i$; while for FAIR and ILP mappings, the following linear program (LP) from [122] (can be derived using Lemma 42) is used to calculate the PS rate allocations.

$$\begin{aligned}
\max \quad & \zeta \\
\text{s.t.} \quad & D_i \hat{u}_i - \frac{\delta^2_{\mathcal{C}_i}}{2}\zeta \geq \mathrm{E}[\mathcal{C}_i] \quad \forall i, \mathrm{E}[\mathcal{U}_i] < 1 \\
& \sum_{i, \mathrm{E}[\mathcal{U}_i] < 1} \hat{u}_i \leq \hat{n}_{\text{low}} \\
& u_i \leq \hat{u}_i \leq 1 \quad \forall i, \mathrm{E}[\mathcal{U}_i] < 1
\end{aligned}$$

where $\zeta^{-1} \geq \max_i \left(\frac{\delta^2_{\mathcal{L}_i}}{2(\hat{u}_i D_i - \mathrm{E}[\mathcal{L}_i])}\right) = \max_i \mathrm{E}[\mathcal{F}_i]$. Therefore, solving the linear program providesthe PS rate allocations $\hat{u}_i$ as well as a bound on the expected tardiness $\mathrm{E}[\mathcal{F}_i]$ of the PS scheduler. Given these values, we can calculate the tardiness of low-utilization tasks using Lemma 43.

6.3.4 A Mapping Algorithm for Stochastic Federated Scheduling

We propose three mapping algorithms for stochastic federated scheduling. The three algorithms differ in their calculation of n_i for high-utilization tasks. They have increasing computational complexity and also have increasing schedulability performance or decreasing upper bound on expected tardiness: The first algorithm, **BASIC**, assigns cores based on utilization; The second algorithm, **FAIR**, assumes that the distributions of execution time and critical-path length are known and assigns cores based on the values with the same cumulative probability from task parameter distributions among all tasks; The last (ILP-Based) algorithm, (**ILP**), tries to minimize the maximum expected tardiness.

6.3.5 BASIC Federated Mapping Algorithm

For a high-utilization tasks τ_i, this mapping algorithm calculates n_i, the number of cores assigned to τ_i as follows:

$$n_i = \begin{cases} \left\lceil \frac{\mathrm{E}[\mathcal{C}_i] - \mathrm{E}[\mathcal{L}_i] - \alpha_i}{D_i - \mathrm{E}[\mathcal{L}_i] - \alpha_i} \right\rceil & (\mathrm{E}\left[\mathcal{U}_i\right] > 1) \\ 2 & (\mathrm{E}\left[\mathcal{U}_i\right] = 1) \end{cases} \tag{6.6}$$

where $\alpha_i = D_i/b - \mathrm{E}\left[\mathcal{L}_i\right] > 0$ and $b = 2$.

The remaining $n_{\text{low}} = m - \sum_{\text{high}} n_i$ cores are assigned to the low-utilization tasks. The mapping algorithm admits a task set as long as $\mathrm{E}\left[\mathcal{U}_{\text{low}}\right] = \sum_{\text{low}} \mathrm{E}\left[\mathcal{U}_i\right] \leq n_{\text{low}}/b$ for $b = 2$.

Note that the major difference between this n_i and the one in Chapter 4 is the extra term α_i. α_i is used to accommodate the variation of execution time and critical-path length. We set this value of α_i to assign roughly the same number of cores relative to utilization. Hence, variances are not required to assign cores.

Bounded Tardiness (Schedulability Test): The tardiness can be bounded for any positive α_i since:

1. For $\mathrm{E}[\mathcal{U}_i] = 1$, $\frac{\mathrm{E}[\mathcal{C}_i]-\mathrm{E}[\mathcal{L}_i]}{D_i-\mathrm{E}[\mathcal{L}_i]} = 1$, so $n_i = 2 > \frac{\mathrm{E}[\mathcal{C}_i]-\mathrm{E}[\mathcal{L}_i]}{D_i-\mathrm{E}[\mathcal{L}_i]}$.

2. For $\mathrm{E}[\mathcal{U}_i] > 1$, since $D_i - \mathrm{E}[\mathcal{L}_i] > \alpha_i > 0$, we have

$$n_i \quad \geq \tfrac{\mathrm{E}[\mathcal{C}_i]-\mathrm{E}[\mathcal{L}_i]-\alpha_i}{D_i-\mathrm{E}[\mathcal{L}_i]-\alpha_i} > \tfrac{\mathrm{E}[\mathcal{C}_i]-\mathrm{E}[\mathcal{L}_i]}{D_i-\mathrm{E}[\mathcal{L}_i]} > \tfrac{\mathrm{E}[\mathcal{C}_i]}{D_i} = \mathrm{E}[\mathcal{U}_i] > 1$$

3. For $\mathrm{E}[\mathcal{U}_i] < 1$, $\mathrm{E}[\mathcal{U}_{\mathrm{low}}] \leq n_{\mathrm{low}}/2 < n_{\mathrm{low}}$.

By Theorem 40 and Lemma 41, this mapping algorithm can guarantee bounded tardiness for both high and low-utilization tasks. Hence, the it serves as a schedulability test that runs in linear time.

Tardiness calculation: Now we describe a faster and simpler method to calculate the upper bound on the expected tardiness of low-utilization tasks when using the BASIC mapping. This method relies on the requirement that $n_{\mathrm{low}} \geq b \sum_{\mathrm{low}} \mathrm{E}[\mathcal{U}_i]$ for $b = 2$. We can simply set PS rate allocation $\hat{u}_i = \min(b\mathrm{E}[\mathcal{U}_i], 1)$. This allocation satisfies the requirement in Lemma 42; therefore, the PS tardiness is

$$\mathrm{E}[\mathcal{F}_i] \leq \frac{\delta^2_{\mathcal{C}_i}}{2(\hat{u}_i^2 D_i - \hat{u}_i \mathrm{E}[\mathcal{C}_i])},$$

and by Lemma 43 the expected tardiness of low-utilization task under GEDF can be calculated directly as

$$\mathrm{E}[\mathcal{T}_i] \leq \frac{\delta^2_{\mathcal{C}_i}}{2(\hat{u}_i^2 D_i - \hat{u}_i \mathrm{E}[\mathcal{C}_i])} + \frac{\eta + n_{\mathrm{low}} M}{n_{\mathrm{low}} - v} + \hat{e}_i, \tag{6.7}$$

Unlike the FAIR and ILP algorithms, this tardiness calculation here does not require solving a linear program; it can be done in linear time.

6.3.6 FAIR Federated Mapping Algorithm

We now present the FAIR mapping, which admits more task sets than the BASIC one, while still providing the same theoretical guarantees. The FAIR mapping utilizes the distributions of execution time and critical-path length and assigns cores based on the values with the same cumulative probability from distributions among all tasks to provide fairness in core assignment. The schedulability test in FAIR still runs in linear time; however, the calculations for core assignment and expected tardiness are more complex, requiring near linear time and linear programming respectively.

We denote $\mathcal{C}_i(p)$ as the value c_i of random variable $\mathcal{C}_i$ when its cumulative distribution function (CDF) $\mathbb{F}_{\mathcal{C}_i}(c_i) = p$ (meaning that the probability that $\mathcal{C}_i \leq c_i$ is equal to p). We denote $\mathcal{L}_i(p)$ and $\mathcal{U}_i(p)$ similarly.

Note that when $p = 0.5$, $\mathcal{C}_i(p) = \mathrm{E}[\mathcal{C}_i]$ and $\mathcal{L}_i(p) = \mathrm{E}[\mathcal{L}_i]$. Additionally, $\mathcal{C}_i(p)$ and $\mathcal{L}_i(p)$ will increase when p increases.

In the FAIR mapping, the number of cores assigned to high-utilization task τ_i (represented by $\hat{n}_i$) is calculated below.

$$\hat{n}_i(p) = \left\lfloor \frac{\mathcal{C}_i(p) - \mathcal{L}_i(p)}{D_i - \mathcal{L}_i(p)} + 1 \right\rfloor \tag{6.8}$$

$$= \begin{cases} \left\lceil \frac{\mathcal{C}_i(p)-\mathcal{L}_i(p)}{D_i-\mathcal{L}_i(p)} \right\rceil & \left(\frac{\mathcal{C}_i(p)-\mathcal{L}_i(p)}{D_i-\mathcal{L}_i(p)} \text{ is not integer} \right) \\ \frac{\mathcal{C}_i(p)-\mathcal{L}_i(p)}{D_i-\mathcal{L}_i(p)} + 1 & \left(\frac{\mathcal{C}_i(p)-\mathcal{L}_i(p)}{D_i-\mathcal{L}_i(p)} \text{ is integer} \right) \end{cases}$$

where p is the same probability for all tasks and $0.5 \leq p < 1$.

The FAIR mapping will admit a task set if $n_{\text{low}} = m - \sum_{\text{high}} \hat{n}_i(p) > \sum_{\text{low}} \mathrm{E}[\mathcal{U}_i(p)]$ for $p = 0.5$.

Bounded Tardiness (Schedulability Test): It is obvious that for $p = 0.5$, each $\hat{n}_i(p = 0.5) = \left\lfloor \frac{\mathrm{E}[\mathcal{C}_i]-\mathrm{E}[\mathcal{L}_i]}{D_i-\mathrm{E}[\mathcal{L}_i]} + 1 \right\rfloor > \frac{\mathrm{E}[\mathcal{C}_i]-\mathrm{E}[\mathcal{L}_i]}{D_i-\mathrm{E}[\mathcal{L}_i]}$ for high-utilization task. Also, for all low-utilization tasks,

$n_{\text{low}} > \sum_{\text{low}} \mathrm{E}\left[\mathcal{U}_i(p = 0.5)\right] = \sum_{\text{low}} \mathrm{E}\left[\mathcal{U}_i\right]$. By Theorem 40 and Lemma 41, FAIR guarantees bounded tardiness for all tasks and serves as a linear time schedulability test.

Dominance in Schedulability: In Section 6.4, we will show that $\hat{n}_i(p = 0.5) \le n_i$ (of BASIC mapping) for any task τ_i and hence $\hat{n}_{low} \ge n_{low}$. Also, the FAIR algorithm allows $\mathrm{E}\left[\mathcal{U}_{\text{low}}\right]$ to be as high as $\hat{n}_{low}$ (instead of $n_{low}/2$ allowed by BASIC). Therefore, FAIR admits strictly more tasks than BASIC.

Core Allocation: $\hat{n}_i(p = 0.5)$, the *minimum core assignment*, is the minimum number of cores required to guarantee bounded tardiness for high-utilization tasks. However, directly using it will result in large tardiness for high-utilization tasks, because more cores are assigned to low-utilization tasks. To be fair to all tasks, the FAIR mapping further improves the *minimum core allocation* by increasing p until the largest $\hat{p}$ while still satisfying $n_{\text{low}} = m - \sum_{\text{high}} \hat{n}_i(\hat{p}) > \sum_{\text{low}}(\mathcal{C}_i(\hat{p})/D_i)$. In this way, FAIR in fact increases the core assignment and PS rate allocation for each task by the same amount according to the CDF of execution time and critical-path length. This ensures fairness among all tasks, because $\hat{p}$ is independent of τ_i. The complexity of this core assignment depends on the number of values of p tested until reaching $\hat{p}$. In practice, a binary search only needs at most 6 tests to find $\hat{p}$ with an accuracy of 0.01.

6.3.7 ILP-Based Federated Mapping Algorithm

We now present a third, ILP-Based, mapping algorithm for stochastic federated scheduling. This algorithm admits exactly the same task sets as FAIR (though it may find a different mapping for these task sets); therefore, it also provides the same theoretical guarantees. However, BASIC and FAIR make no attempt to balance maximum tardiness explicitly among high and low-utilization tasks. Compared to FAIR, the ILP mapping does not require the distributions of execution time and critical-path length. Instead, the stander deviations and covariance are used in ILP.

The ILP algorithm converts the mapping problem for high-utilization tasks into an *integer linear program* (ILP) that tries to minimize the maximum tardiness; when combined with the linear program for low-utilization tasks stated in Section 6.3.3, the resulting mixed linear program indirectly tries to balance the tardiness among all tasks.

We convert Inequality (6.5) into a form similar to the expected tardiness of the PS schedule; that is, we define ζ_i where $\zeta^{-1} = \max_i \mathrm{E}\left[\mathcal{T}_i\right]$ and ζ is defined in terms of n_i. First, for task τ_i, let $\delta_i^2 = max\big(\delta_{\mathcal{L}_i}^2 (m-1)^2/m, \delta_{\mathcal{C}_i}^2/2, \sigma(\mathcal{L}_i, \mathcal{C}_i)(m-1)/m\big)$. Note that δ_i^2 is bounded and can be calculated using only the expectation and variance of the task's execution time and critical-path length without knowing n_i. Now we use the fact that $2 \leq n_i \leq m$ for high-utilization task τ_i and see that

$$\begin{aligned}
\delta_i^2 &\geq \delta_{\mathcal{L}_i}^2 (m-1)^2/m = \delta_{\mathcal{L}_i}^2 (m-1)(1-1/m) \\
&\geq \delta_{\mathcal{L}_i}^2 (n_i-1)(1-1/n_i) = \delta_{\mathcal{L}_i}^2 (n_i-1)^2/n_i \\
\delta_i^2 &\geq \delta_{\mathcal{C}_i}^2/2 \geq \delta_{\mathcal{C}_i}^2/n_i \\
\delta_i^2 &\geq \sigma(\mathcal{L}_i, \mathcal{C}_i)(m-1)/m = \sigma(\mathcal{L}_i, \mathcal{C}_i)(1-1/m) \\
&\geq \sigma(\mathcal{L}_i, \mathcal{C}_i)(1-1/n_i) = \sigma(\mathcal{L}_i, \mathcal{C}_i)(n_i-1)/n_i.
\end{aligned}$$

Now we calculate the upper bound on the variance of $\delta_{t_{i,j}}^2$ (from Inequality (6.4)) using δ_i^2

$$\begin{aligned}
\delta_{t_{i,j}}^2 &= \delta_{\mathcal{L}_i}^2 (n_i-1)^2/n_i^2 + \delta_{\mathcal{C}_i}^2/n_i^2 + 2\sigma(\mathcal{L}_i, \mathcal{C}_i)(n_i-1)/n_i^2 \\
&= \frac{\delta_{\mathcal{L}_i}^2 (n_i-1)^2/n_i + \delta_{\mathcal{C}_i}^2/n_i + 2\sigma(\mathcal{L}_i, \mathcal{C}_i)(n_i-1)/n_i}{n_i} \\
&\leq 4\delta_i^2/n_i
\end{aligned}$$

By Corollary 38, the expected tardiness is bounded by

$$
\begin{aligned}
\mathrm{E}\left[\mathcal{T}_i\right] &\leq \frac{\delta_{\mathcal{X}}^2}{2(Y - \mathrm{E}\left[\mathcal{X}\right])} \\
&\leq \frac{4\delta_i^2/n_i}{2(D_i - (\mathrm{E}\left[\mathcal{L}_i\right](n_i-1) + \mathrm{E}\left[\mathcal{C}_i\right])/ni)} \\
&\leq \frac{2\delta_i^2}{n_i D_i - (\mathrm{E}\left[\mathcal{L}_i\right](n_i-1) + \mathrm{E}\left[\mathcal{C}_i\right])} \\
&= \frac{2\delta_i^2}{n_i(D_i - \mathrm{E}\left[\mathcal{L}_i\right]) - (\mathrm{E}\left[\mathcal{C}_i\right] - \mathrm{E}\left[\mathcal{L}_i\right])} \qquad (6.9)
\end{aligned}
$$

Now we can set $\zeta^{-1} \geq \max_i \left(\frac{2\delta_i^2}{n_i(D_i - \mathrm{E}[\mathcal{L}_i]) - (\mathrm{E}[\mathcal{C}_i] - \mathrm{E}[\mathcal{L}_i])}\right) \geq \max_i \mathrm{E}\left[\mathcal{T}_i\right]$ for high-utilization tasks and get inequality (6.11).

Combining this definition of ζ with the linear program in Section 6.3.3, we get the following mixed linear program:

$$
\begin{aligned}
\max \quad & \zeta \\
\text{s.t.} \quad & D_i\hat{u}_i - \frac{\delta_{\mathcal{C}_i}^2}{2}\zeta \geq \mathrm{E}\left[\mathcal{C}_i\right] && \forall i, \mathrm{E}\left[\mathcal{U}_i\right] < 1 && (6.10) \\
& (D_i - \mathrm{E}\left[\mathcal{L}_i\right])n_i - 2\delta_i^2\zeta \geq \mathrm{E}\left[\mathcal{C}_i\right] - \mathrm{E}\left[\mathcal{L}_i\right] && && \\
& && \forall i, \mathrm{E}\left[\mathcal{U}_i\right] \geq 1 && (6.11) \\
& \sum_{i, \mathrm{E}[\mathcal{U}_i] < 1} \hat{u}_i + \sum_{i, \mathrm{E}[\mathcal{U}_i] \geq 1} n_i \leq m && && (6.12) \\
& u_i \leq \hat{u}_i \leq 1 && \forall i, \mathrm{E}\left[\mathcal{U}_i\right] < 1 && (6.13) \\
& \hat{n}_i(p = 0.5) \leq n_i && \forall i, \mathrm{E}\left[\mathcal{U}_i\right] \geq 1 && (6.14) \\
& n_i \text{ is integer} && \forall i, \mathrm{E}\left[\mathcal{U}_i\right] \geq 1 && (6.15)
\end{aligned}
$$

We solve this ILP to calculate: integral n_i— the number of cores assigned to high utilization task τ_i; fractional $\hat{u}_i$ — a valid PS rate allocation for low-utilization task τ_i; and ζ. Using the resulting n_i for high utilization tasks, we can calculate $n_{\text{low}} = m - \sum_{\text{high}} n_i$, the number of cores assigned to low-utilization tasks.

Explanation of Constraints: Constraints (6.14) and (6.15) guarantee that each high-utilization task τ_i gets at least $\hat{n}_i(p = 0.5)$ dedicated cores; therefore Theorem 40 guarantees its bounded tardiness. Constraint (6.13) guarantees that the PS rate allocation is larger than the utilization of low-utilization tasks; therefore Lemma 42 guarantees bounded tardiness of these tasks. Constraint (6.12) guarantees that $n_{\text{low}} + n_{\text{high}} \leq m$. Finally, Constraint (6.10) is inherited from the LP in Section 6.3.3.

Optimal Greedy Solution to the ILP: General ILP problems can be hard to solve. However, there is a unique property of the above ILP — ζ will decrease if at least one n_i or $\sum_{\text{low}} \hat{u}_i$ increases and the rest remain the same. Relying on this, we can easily see that a greedy algorithm — starting with the core assignment (n_i and $\hat{u}_i(p = 0.5)$) from the minimum core allocation of the FAIR mapping, iteratively increases the n_i or $\sum_{\text{low}} \hat{u}_i$ (a high utilization task or the sum of low utilization tasks) with largest tardiness by 1 until just before Constraint (6.12) would not hold — will successfully find the optimal solution to this ILP problem (provided that the LP in Section 6.3.3 can directly calculate an optimal solution). By applying the greedy solution, we can reduce the mixed-ILP problem to an iterative LP problem. Obviously, the maximum number of iterations needed by the greedy algorithm is m.

Relationship to FAIR: The ILP mapping algorithm admits exactly the same task sets that FAIR admits: if FAIR admits a task set ($\hat{n}_i(p = 0.5)$ and $n_{\text{low}} = m - \sum_{\text{high}} \hat{n}_i(p = 0.5)$), then that mapping is a trivially feasible solution to the ILP since it satisfies all constraints for $\zeta = 0$. On the other hand, if the FAIR algorithm cannot find a solution, then there is no feasible solution to the ILP. Therefore, since FAIR provides a capacity augmentation bound of 2, so does this algorithm.

Faster Schedulability Test: As a consequence of the relationship with FAIR, we do not have to solve the ILP to check if the task set is schedulable using this ILP-based mapping; we can simply run FAIR to check for schedulability and only solve the ILP to find the mapping if the task set is, in fact, schedulable.

Tardiness Calculation: On solving the mixed linear program, we get n_i for each high utilization task and $\hat{u}_i$ for each low utilization task. Therefore, we can use Inequalities (6.5) and (6.7) to calculate the tardiness of these tasks, respectively.

Note that the mixed linear program criterion is a little imprecise; maximizing ζ does not directly optimize the overall tardiness bound. Instead, it only tries to balance parts of the tardiness. After applying Inequalities (6.7) and (6.5) for calculating tardiness, the resulting tardiness of each high-utilization task is actually less than the optimized bound ζ^{-1}, while the tardiness of low-utilization tasks is actually higher than ζ^{-1}.

To further balance the overall tardiness, instead of using the strict upper bound of $\delta^2_{t_{i,j}}$ (from Inequality (6.9)) in the calculation of ζ, we can approximate it. The reason we cannot directly use Inequality (6.4) to calculate $\delta^2_{t_{i,j}}$ is because we do not know n_i before we solve the integer linear program. However, we can approximate $\delta^2_{t_{i,j}}$ by using $\hat{n}_i(p = 0.5)$ instead of n_i. Then, we have $\delta^2_{t_{i,j}} = \frac{\delta^2_{\mathcal{L}_i}(\hat{n}_i - 1)^2/\hat{n}_i + \delta^2_{\mathcal{C}_i}/\hat{n}_i + 2\sigma(\mathcal{L}_i, \mathcal{C}_i)(\hat{n}_i - 1)/\hat{n}_i}{n_i} = \frac{\delta^2_i}{n_i}$. This may provide a better tardiness bound for all tasks.

However, when the worst-case execution time of a low-utilization task is large, the achieved mapping may still result in a larger maximum tardiness (for that task) than optimal.

6.4 Stochastic Capacity Augmentation of 2 for Stochastic Federated Scheduling

6.4.1 Stochastic Capacity Augmentation Bound for BASIC

Theorem 44 *The federated scheduling algorithm has a stochastic capacity augmentation bound of b =2.*

In order to prove Theorem 44, we first prove that the BASIC mapping strategy always admits all *eligible* task sets — task sets that satisfy Conditions (6.1) and (6.2) in Definition 5 for for b =2.

The BASIC mapping algorithm admits a task set if, $\mathrm{E}[\mathcal{U}_{\text{low}}] \leq n_{\text{low}}/b$ for $b = 2$. Therefore, we must prove that for all task sets that satisfy Conditions (6.1) and (6.2), n_{low} is large enough to admit the task set.

First, we prove that the number of cores assigned to high-utilization tasks n_{high} is bounded by $b \sum_{\text{high}} \mathrm{E}[\mathcal{U}_i]$.

Lemma 45 *For a high-utilization task τ_i ($1 \leq \mathrm{E}[\mathcal{U}_i]$), if $D_i > b\mathrm{E}[\mathcal{L}]_i$ (Condition (6.2)), then the number of assigned cores $n_i \leq b\mathrm{E}[\mathcal{U}_i]$ with $b = 2$.*

Proof. For $\mathrm{E}[\mathcal{U}_i] > 1$, since $b(\mathrm{E}[\mathcal{L}_i] + \alpha_i) = D_i$, so

$$\begin{aligned} \mathrm{E}[\mathcal{C}_i] &= b(\mathrm{E}[\mathcal{L}_i] + \alpha_i)\mathrm{E}[\mathcal{U}_i] \\ \Rightarrow \quad D_i - \mathrm{E}[\mathcal{L}_i] - \alpha_i &= (b-1)(\mathrm{E}[\mathcal{L}_i] + \alpha_i) \end{aligned}$$

Therefore, we can bound n_i by

$$\begin{aligned} n_i &= \left\lceil \frac{\mathrm{E}[\mathcal{C}_i] - \mathrm{E}[\mathcal{L}_i] - \alpha_i}{D_i - \mathrm{E}[\mathcal{L}_i] - \alpha_i} \right\rceil < \frac{\mathrm{E}[\mathcal{C}_i] - \mathrm{E}[\mathcal{L}_i] - \alpha_i}{D_i - \mathrm{E}[\mathcal{L}_i] - \alpha_i} + 1 \\ &= \frac{2(\mathrm{E}[\mathcal{L}_i] + \alpha_i)\mathrm{E}[\mathcal{U}_i] - (\mathrm{E}[\mathcal{L}_i] + \alpha_i)}{\mathrm{E}[\mathcal{L}_i] + \alpha_i} + 1 \\ &= 2\mathrm{E}[\mathcal{U}_i] \end{aligned}$$

For $\mathrm{E}[\mathcal{U}_i] = 1$, $n_i = 2 = 2\mathrm{E}[\mathcal{U}_i]$. Therefore, $n_{\text{high}} = \sum_{\text{high}} n_i \leq b \sum_{\text{high}} \mathrm{E}[\mathcal{U}_i]$ for $b = 2$. □

Since task set τ satisfies Condition (6.1), the total utilization $\sum \mathrm{E}[\mathcal{U}_i] \leq m/b$ for $b = 2$. So we have

$$n_{\text{low}} = m - n_{\text{high}} \geq b \sum_i \mathrm{E}[\mathcal{U}_i] - b \sum_{\text{high}} \mathrm{E}[\mathcal{U}_i] = b \sum_{\text{low}} \mathrm{E}[\mathcal{U}_i]$$

Hence, BASIC's admission criterion is satisfied and it admits any task set satisfying Conditions (6.1) and (6.2). Since BASIC always provides bounded tardiness to task sets it admits (Section 6.3.2), by Definition 5 this establishes Theorem 44.

6.4.2 Stochastic Capacity Augmentation Bound for FAIR

Theorem 46 *The FAIR federated scheduling algorithm has a stochastic capacity augmentation bound of $b = 2$.*

To prove Theorem 46, we simply prove if the BASIC admits a task set, then FAIR does as well; since BASIC admits any task set that satisfies Conditions (6.1) and (6.2) of Definition 5 for $b = 2$, FAIR also admits them. Since FAIR always provides bounded tardiness to task sets it admits, this establishes Theorem 46.

First, we show that the minimum core assignment $\hat{n}_i(p = 0.5)$ to each high-utilization task by the FAIR algorithm is at most the number of cores n_i that the BASIC algorithm assigns.

Lemma 47 *If $n_i = \left\lceil \frac{\mathrm{E}[\mathcal{C}_i]-\mathrm{E}[\mathcal{L}_i]-\alpha_i}{D_i-\mathrm{E}[\mathcal{L}_i]-\alpha_i} \right\rceil$ for $\mathrm{E}\left[\mathcal{U}_i\right] > 1$ and $n_1 = 2$ for $\mathrm{E}\left[\mathcal{U}_i\right] = 1$; and $\hat{n}_i(p = 0.5) = \left\lfloor \frac{\mathcal{C}_i(p)-\mathcal{L}_i(p)}{D_i-\mathcal{L}_i(p)} + 1 \right\rfloor = \left\lfloor \frac{\mathrm{E}[\mathcal{C}_i]-\mathrm{E}[\mathcal{L}_i]}{D_i-\mathrm{E}[\mathcal{L}_i]} + 1 \right\rfloor$; then $\hat{n}_i \leq n_i$ for all $\mathrm{E}\left[\mathcal{U}_i\right] \geq 1$.*

Proof. To make the proof straightforward, we use the two cases from our definition of $\hat{n}_i$ in Section 6.3.4.

For $\mathrm{E}\left[\mathcal{U}_i\right] > 1$, obviously $\frac{\mathrm{E}[\mathcal{C}_i]-\mathrm{E}[\mathcal{L}_i]-\alpha_i}{D_i-\mathrm{E}[\mathcal{L}_i]-\alpha_i} > \frac{\mathrm{E}[\mathcal{C}_i]-\mathrm{E}[\mathcal{L}_i]}{D_i-\mathrm{E}[\mathcal{L}_i]} > 1$, since $D_i - \mathrm{E}\left[\mathcal{L}_i\right] > \alpha_i > 0$. So we denote $\epsilon > 0$ and

$$\frac{\mathrm{E}\left[\mathcal{C}_i\right] - \mathrm{E}\left[\mathcal{L}_i\right] - \alpha_i}{D_i - \mathrm{E}\left[\mathcal{L}_i\right] - \alpha_i} = \frac{\mathrm{E}\left[\mathcal{C}_i\right] - \mathrm{E}\left[\mathcal{L}_i\right]}{D_i - \mathrm{E}\left[\mathcal{L}_i\right]} + \epsilon$$

When $\frac{\mathrm{E}[\mathcal{C}_i]-\mathrm{E}[\mathcal{L}_i]}{D_i-\mathrm{E}[\mathcal{L}_i]}$ is not integer,

$$\hat{n}_i(p = 0.5) = \left\lceil \frac{\mathrm{E}\left[\mathcal{C}_i\right] - \mathrm{E}\left[\mathcal{L}_i\right]}{D_i - \mathrm{E}\left[\mathcal{L}_i\right]} \right\rceil \leq \left\lceil \frac{\mathrm{E}\left[\mathcal{C}_i\right] - \mathrm{E}\left[\mathcal{L}_i\right] - \alpha_i}{D_i - \mathrm{E}\left[\mathcal{L}_i\right] - \alpha_i} \right\rceil = n_i$$

When $\frac{\mathrm{E}[\mathcal{C}_i]-\mathrm{E}[\mathcal{L}_i]}{D_i-\mathrm{E}[\mathcal{L}_i]}$ is integer, since $\epsilon > 0$,

$$\begin{aligned} n_i &= \left\lceil \frac{\mathrm{E}\left[\mathcal{C}_i\right] - \mathrm{E}\left[\mathcal{L}_i\right] - \alpha_i}{D_i - \mathrm{E}\left[\mathcal{L}_i\right] - \alpha_i} \right\rceil = \left\lceil \frac{\mathrm{E}\left[\mathcal{C}_i\right] - \mathrm{E}\left[\mathcal{L}_i\right]}{D_i - \mathrm{E}\left[\mathcal{L}_i\right]} + \epsilon \right\rceil \\ &\geq \frac{\mathrm{E}\left[\mathcal{C}_i\right] - \mathrm{E}\left[\mathcal{L}_i\right]}{D_i - \mathrm{E}\left[\mathcal{L}_i\right]} + 1 = \hat{n}_i(p = 0.5) \end{aligned}$$

For $\mathrm{E}\left[\mathcal{U}_i\right] = 1$, $\hat{n}_i = 2 = n_i$. Therefore, for all cases, $\hat{n}_{\text{high}} = \sum_{\text{high}} \hat{n}_i \leq \sum_{\text{high}} n_i = n_{\text{high}}$.

□

FAIR has more cores available for low utilization tasks than BASIC does, since $\hat{n}_{\text{low}}(p = 0.5) = m - \hat{n}_{\text{high}}(p = 0.5) \geq m - n_{\text{high}} = n_{\text{low}}$. It also allows the total utilization of low-utilization tasks to be as high as $\hat{n}_{\text{low}}(p = 0.5)$, while basic only allows it to be n_{low}/b. Therefore, FAIR admits any task set that BASIC admits.

Note that FAIR will only increase $\hat{n}_i$ to $\hat{n}_i(p > 0.5)$ if it admits the task set. Thus, as far as the schedulability and capacity augmentation bound are concerned, this will not affect the proof above. In the most loaded case, $\hat{n}_i(\hat{p}) = \hat{n}_i(p = 0.5)$.

6.5 Numerical Evaluation

To compare the different performances of these schedulability tests for *stochastic task sets*, here, we present a numerical evaluation on randomly generated task sets with probability distributions on execution time and critical-path length.

6.5.1 Task Sets Generation and Experimental Setup

We evaluate the schedulability results on a varying number of cores m: $4, 8, 16, 32$, and 64. For various total task set utilizations U starting from 10% to 80%, we generate task sets, add tasks and load the system to be exactly mU — fully loading a unit speed machine. Results for 4 and 64 cores are similar to the rest, so we omit them for brevity.

For each task, we assume normal distributions of execution time and critical-path length. We uniformly generate the expected execution time $\mathrm{E}[\mathcal{C}_i]$ between 1 and 100. For tasks with small variance, we uniformly generate variance to be from 5% to 10% of $\mathrm{E}[\mathcal{C}_i]$; for tasks with large variance, we let it be from 5% to 500%. We generate the critical-path length following the same rules and ensure the average parallelism $\mathrm{E}[\mathcal{C}_i]/\mathrm{E}[\mathcal{L}_i]$ is 32. To ensure a reasonable amount of high-utilization tasks in a task set on m cores, we uniformly generate the task utilization u_i between 0.4 to $\sqrt{m}$. Since we assume a normal distribution for execution time and critical-path length, with the expected mean and standard deviation, we can calculate the worst-case execution time by calculating the value $\hat{c}_i$ of the distribution when the probability of a longer execution time is less than 0.01. Deadline is calculated by $u_i\mathrm{E}[\mathcal{C}_i]$

Using the task set setups above, we run each setting for 100 task sets. We conduct two sets of experiments:

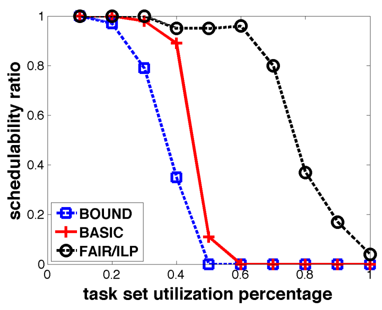
(1) We want to evaluate the performance of the two schedulability tests: BASIC and FAIR. In addition, we use the simple schedulability test from the stochastic capacity augmentation bound as a baseline comparison.

(2) We want to evaluate the different tardiness bounds of each individual task using different federated mapping algorithms. For task sets that are schedulable according the BASIC test, we record the maximum, mean and minimum tardiness of each task set.

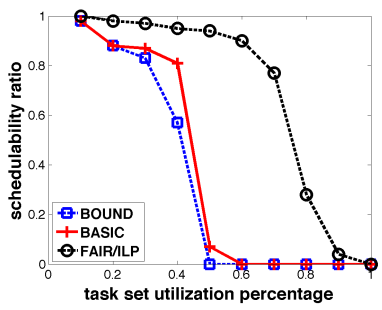
6.5.2 Experiment Results

Schedulability Performance

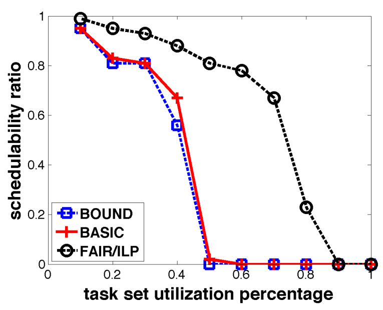
We evaluate the performances of different schedulability tests: BOUND (as a baseline), BASIC and FAIR. Note that, as we have proved, schedulability with the ILP mapping algorithm is exactly the same as with the FAIR mapping algorithm (denoted as FAIR/ILP in the figure). Also, since the exact variance of a task is not needed to run these schedulability



(a) 8 cores

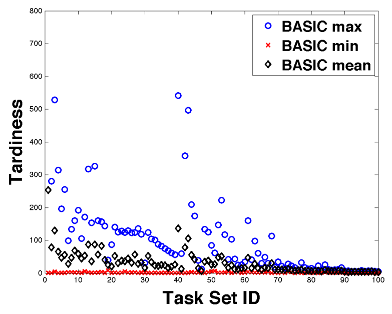


(b) 16 cores

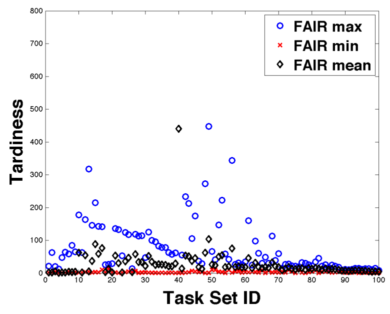


(c) 32 cores

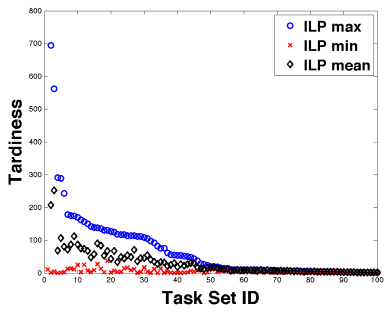
Figure 6.1: Task set utilization vs. schedulability ratio (in percentages) for different number of cores.



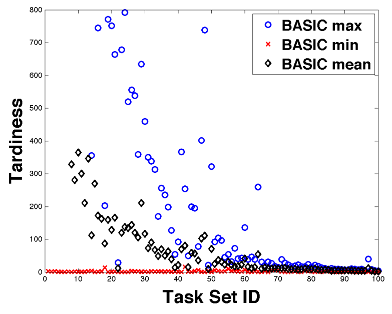
(a) BASIC mapping, small variance



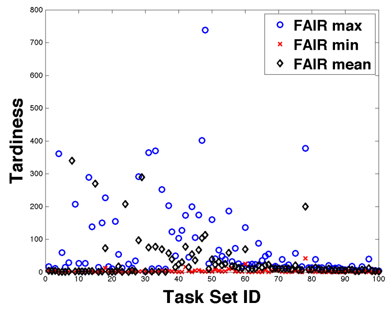
(b) FAIR mapping, small variance



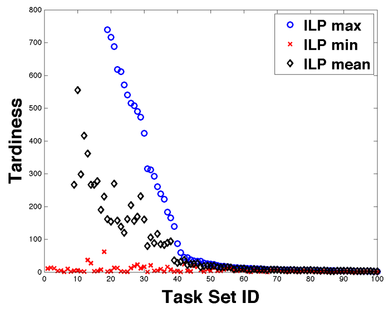
(c) ILP mapping, small variance



(d) BASIC mapping, large variance



(e) FAIR mapping, large variance



(f) ILP mapping, large variance

Figure 6.2: Maximum, mean and minimum tardiness for parameters with small and large variances.

tests, the calculated schedulability of task sets with small variance and large variance is the same. Therefore, we do not include these curves in the figures.

From Figure 6.1, we can see that across different numbers of cores, the FAIR/ILP algorithm performs the best, while BOUND performs the worst. Even though the bound indicates that task sets with total utilization larger than $50\%m$ may not be schedulable in terms of bounded tardiness, the two other linear time schedulability tests can still admit task sets up to around 60% for BASIC and 80% for FAIR.

Also note that some task sets with 10% utilization are deemed unschedulable by BOUND. This is due to the critical-path length requirement for parallel tasks by BOUND. For a few tasks with 100% utilization, the FAIR algorithm still guarantees bounded tardiness, because all tasks in the set are low-utilization tasks, and the GEDF scheduler can ensure bounded tardiness for sequential tasks with no utilization loss.

Tardiness of Tasks with Small and Large Variance

For task sets for which bounded tardiness is guaranteed, we would like to compare the guaranteed expected tardiness. Note that both the LP and ILP optimization in the FAIR and ILP mapping algorithms only try to optimize the maximum tardiness of the entire task sets. Therefore, it would be more interesting to see the different expected tardiness bound for the individual tasks.

Figure 6.2 shows the maximum, mean and minimum expected tardiness calculated from the BASIC, FAIR and ILP mappings for task sets with small and large execution time variations respectively. To make it easy to compare them, we sort all the figures according to the maximum tardiness of the ILP mapping for that corresponding setting (low and high variances).

Not surprisingly, BASIC performs the worst among all three mappings, if we count the number of task sets for which BASIC generates the largest maximum tardiness. In fact, out of all randomly generated task sets, 92% and 85% of the task sets have smaller maximum tardiness with ILP than with BASIC, given small and large variance respectively. Compare FAIR and BASIC, 58% and 76% respectively have lower maximum tardiness under FAIR.

However, we can also see that the maximum tardiness from the BASIC mapping is comparable to (only slightly worse than) that from the FAIR mapping, when variances of execution time and critical-path length are small. It is also comparable to ILP when the variances are large. This is probably because all compared task sets satisfy the requirement of the bound. Therefore, there are enough cores for BASIC mapping to approximate the better core assignment. Hence, when the variations are small, one could use the BASIC mapping to bound the tardiness.

We also find that with large variances, the increase of maximum tardiness with FAIR is not significant, compared to BASIC and ILP. This is not surprising for BASIC result, because it confirms our hypothesis that the BASIC mapping does not take variation into account when allocating cores. However, ILP does try to balance the tardiness of all tasks, considering variance similarly to FAIR.

In fact, comparing FAIR and ILP, we notice that 67% and 58% task sets respectively have smaller maximum tardiness using ILP. ILP results seem much worse with large variances, only because for some task sets, the maximum tardiness comes from low-utilization tasks. Even through ILP can minimize the tardiness for high-utilization tasks, the LP calculation for low-utilization tasks only minimize the part of tardiness (the maximum tardiness of the PS scheduler in Lemma 42) but cannot directly minimize the overall tardiness in Lemma 43. As FAIR inflates the parameters for low-utilization tasks, the LP calculation may result in a better PS rate allocation and hence smaller tardiness.

Chapter 7

Work Stealing for Large Scale Soft Real-time Systems

Parallel real-time scheduling has emerged as a promising scheduling paradigm for computationally intensive real-time applications on multicore systems. Unlike in traditional *multiprocessor scheduling* with only *inter-task parallelism* (where each sequential task can only utilize one core at a time), in *parallel scheduling* each task has *intra-task parallelism* and can run on multiple cores at the same time. As today's real-time applications are trying to provide increasingly complex functionalities and hence have higher computational demands, they ask for larger scale systems in order to provide the same real-time performance. For example, real-time hybrid structural testing for a large building may require executing complex structural models over many cores at the same frequency as the physical structure [70].

Despite recent results in parallel real-time scheduling, however, we still face significant challenges in deploying large-scale real-time applications on microprocessors with increasing numbers of cores. In order to guarantee desired parallel execution of a task to meet its deadline, theoretic analysis often assumes that it is executed by a greedy (work conserving) scheduler, which requires a centralized data structure for scheduling. On the other hand, for general-purpose parallel job scheduling it has been known that centralized scheduling approaches suffer considerable scheduling overhead and performance bottleneck as the number of cores increases. In contrast, a *randomized work stealing* approach is widely used in many parallel runtime systems, such as Cilk, Cilk Plus, TBB, X10, and TPL [32,89,107,134,147]. In work stealing, each core steals work from a randomly chosen core in a decentralized manner, thereby avoiding the overhead and bottleneck of centralized scheduling. However, unlike a centralized scheduler, due to the randomized and distributed scheduling decision making strategy, work stealing may not be suitable for hard real-time tasks.

In this chapter, we explore using randomized work stealing to support large-scale soft real-time applications that have timing constraints but do not require hard guarantees. Despite the unpredictable nature of work stealing, our experiments with benchmark programs found that work stealing (in Cilk Plus) delivers smaller maximum response times than a centralized greedy scheduler (in GNU OpenMP) while exhibiting small variance. To leverage randomized work stealing for scalable real-time computing, we present *Real-Time Work Stealing (RTWS)*, a real-time extension to the widely used Cilk Plus concurrency platform. RTWS employs federated scheduling to decide static core assignment to parallel real-time tasks offline, while using the work stealing scheduler to execute each task on its dedicated cores online. RTWS supports parallel programs written in Cilk Plus with only minimal modifications, namely a single level of indirection of the program's entry point. Furthermore, RTWS requires only task parameters that can be readily measured using existing Cilk Plus tools.

This chapter presents the following contributions:

1. Empirical study of the performance and variability of parallel tasks under randomized work stealing vs. centralized greedy scheduler.

2. Design and implementation of RTWS, which schedules multiple parallel real-time tasks through the integration of federating scheduling and work stealing.

3. Theoretical analysis to adapt federated scheduling to incorporate work stealing overhead.

4. Evaluation of RTWS with benchmark applications on a 32-core testbed that demonstrates the significant advantages of RTWS in terms of deadline miss ratio, relative response time and required resource capacity when comparing with the integration of federated scheduling and centralized scheduler.

7.1 The Case for Randomized Work Stealing for Soft Real-Time Tasks

In this section, we compare the performance of a work stealing scheduler in GNU Cilk Plus with a centralized scheduler in GNU OpenMP for highly scalable parallel programs (see Section 2.3 for a brief introduction). We choose these two implementations because OpenMP and CilkPlus are two of the most widely used parallel languages (and runtime systems) that have been developed by industry and the open source community over more than a decade; they are the only two parallel languages that are supported by both GCC and ICC.

Our goal is to answer two questions: (1) Is it indeed the case that work stealing provides substantially better performance than centralized scheduler for parallel programs? Our experiments indicate that for many programs, including both synthetic tasks and real benchmark programs, work stealing provides much higher scalability. (2) Can work stealing be used for real-time systems? In particular, one might suspect that even if work stealing performs better than centralized scheduler *on average*, the randomization used in work stealing would make its performance too unpredictable to use even in soft real-time systems. Our experiments indicate that this is not the case — in fact, the variation in execution time using Cilk Plus' work-stealing scheduler is small and is comparable to or better than the variation seen in the deterministic centralized scheduler.

7.1.1 Scalability Comparison

We first compare the scalability of the OpenMP centralized scheduler with the Cilk Plus work-stealing scheduler. To do so, we use two types of programs: (1) three synthetic programs that are synchronous tasks; and (2) three real benchmark programs, namely Cholesky

No. cores	Synchronous Task - Type 1		
	OpenMP (med., max, 99th per.)	Cilk Plus (med., max, 99th per.)	Ratio
1	955.13, 958.10, 956.98	948.52, 953.41, 950.94	1.00
6	173.68, 174.31, 174.18	160.77, 161.46, 161.26	0.93
12	256.63, 259.19, 258.89	81.68, 82.56, 81.93	0.32
18	342.20, 365.99, 362.99	55.42, 59.22, 58.96	0.16
24	328.52, 331.11, 329.75	41.23, 45.22, 44.78	0.14
30	311.92, 330.00, 329.00	33.66, 35.02, 34.64	0.11
No. cores	Synchronous Task - Type 2		
	OpenMP (med., max, 99th per.)	Cilk Plus (med., max, 99th per.)	Ratio
1	1243.7, 1247.2, 1246.6	1237.2, 1239.9, 1239.2	0.99
6	210.22, 210.84, 210.74	213.77, 214.30, 214.19	1.02
12	111.58, 111.94, 111.87	107.90, 108.69, 108.11	0.97
18	95.55, 95.96, 95.92	73.45, 73.82, 73.62	0.77
24	85.97, 126.00, 123.01	58.95, 74.80, 69.18	0.59
30	86.74, 119.01, 86.96	45.07, 48.27, 47.33	0.41
No. cores	Synchronous Task - Type 3		
	OpenMP (med., max, 99th per.)	Cilk Plus (med., max, 99th per.)	Ratio
1	948.42, 950.39, 949.97	902.38, 903.29, 903.18	0.95
6	156.47, 156.94, 156.77	155.77, 156.06, 156.00	0.99
12	79.03, 79.34, 79.27	78.80, 79.46, 78.97	1.00
18	53.07, 53.49, 53.28	54.05, 54.41, 54.29	1.02
24	39.99, 69.95, 40.18	40.68, 44.35, 43.62	0.63
30	32.20, 33.18, 32.39	33.40, 37.12, 34.48	1.12

Table 7.1: Median, maximum, and 99th percentile execution times of synchronous tasks for OpenMP and Cilk Plus implementations (in milliseconds) and the ratios of the maximum execution times of Cilk Plus over OpenMP implementations.

factorization, LU decomposition and Heat diffusion — none is synchronous and all have complex DAG dependences (of different types).

We implemented these programs in both Cilk Plus and OpenMP. It is important to note that the entire source code of each program is the same, except that the parallel directives are in either Cilk Plus or OpenMP. Both implementations are compiled by GCC, while linked to either Cilk Plus or OpenMP runtime libraries. Hence, the same program written in Cilk Plus and OpenMP has the same structure and therefore the same theoretical work and span.

Synthetic Synchronous Tasks: The synthetic synchronous tasks have different characteristics to compare the schedulers under different circumstances:

(1) **Type 1** tasks have a large number of nodes per segment, but nodes has small execution times.

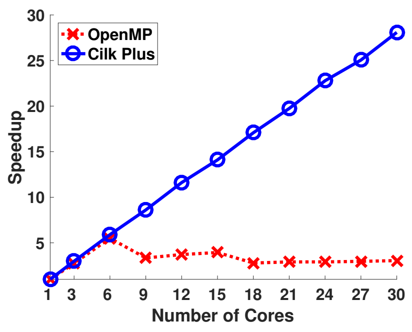
(2) **Type 2** tasks have a moderate number of nodes per segment and moderate work per node.

(3) **Type 3** tasks have a small number of nodes per segment, but nodes have large execution times.

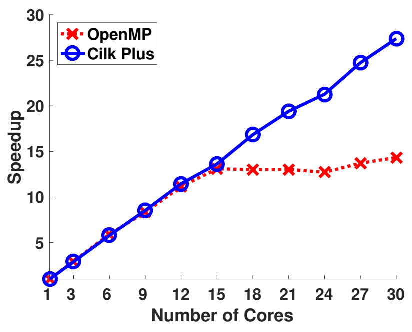
The number of segments for all three types of synchronous tasks are generated from 10 to 20. For synchronous task type 1, we generate the number of nodes for each segment from $100,000$ to $200,000$ and the execution time per node from 5 to 10 nanoseconds; for task type 2, the number of nodes per segment varies from $10,000$ to $20,000$ and the execution time of each node from $2,000$ to $4,500$ nanoseconds; for task type 3, the number of nodes for each segment is from $1,000$ to $2,000$ and each node rans from $20,000$ to $50,000$ nanoseconds. The total work for synchronous tasks of different types was therefore similar. For each synchronous task generated, we ran it on varying numbers of cores with both Cilk Plus and OpenMP and we ran it 1000 times for each setting.

Table 7.1 shows the median, maximum, and 99^{th} percentile execution times of OpenMP and Cilk Plus tasks as well as the ratios of the maximum execution time of Cilk Plus over OpenMP implementations for the three types of synchronous tasks on varying numbers of cores. For most settings, Cilk Plus tasks obtain smaller maximum execution times than OpenMP tasks, as shown in the ratios. We also notice that for type 1 tasks the execution times of the OpenMP tasks even increase when the number of cores is high (e.g, for 18, 24, 30 cores) whereas Cilk Plus tasks keep a steady speedup.

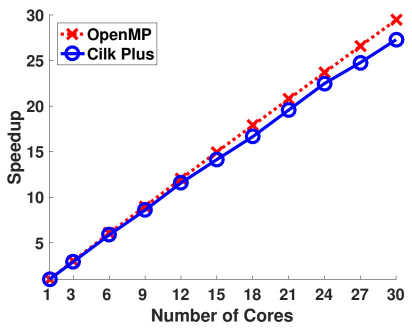
Figure 7.1 shows the speedup of these synchronous tasks. For all three types of tasks, Cilk Plus provides steady and almost linear speedup as we scale up the number of cores. In contrast, for synchronous task type 1 in Figure 7.1(a) where the segment lengths are short



(a) Speedup of task type 1



(b) Speedup of task type 2



(c) Speedup of task type 3

Figure 7.1: Speedup of synchronous tasks in OpenMP and Cilk Plus implementations

and there are many nodes in each segment, OpenMP inevitably suffers high synchronization overhead due to the contention among threads that constantly access the global work queue. This overhead is mitigated when the number of nodes in each segment is smaller and the segment lengths are longer, as in Fig. 7.1(c). In this setting, OpenMP slightly outperforms Cilk Plus, though Cilk Plus still has comparable speedup to OpenMP. Figure 7.1(b) demonstrates the scalability of OpenMP and Cilk Plus with parameters generated in between.

Real DAG Benchmark Programs: To compare the performance between work stealing and centralized scheduler for programs with more complex DAG structures, we use three benchmark programs as described below.

(a) Cholesky factorization (Cholesky): Using divide and conquer, Cholesky program performs Cholesky factorization of a sparse symmetric positive definite matrix into the product of a lower triangular matrix and its transpose. The work and parallelism of Cholesky both increase when the matrix size increases. Note that because Cholesky is parallelized using divide and conquer method, it has lots of spawn and sync operations, forming a complex DAG structure.

(b) LU decomposition (LU): Similar to Cholesky, LU also performs matrix factorization, but the input matrix does not need to be positive definite and the output upper triangular matrix is not necessarily the transpose of the lower triangular matrix. LU also decomposes the matrix using divide and conquer and provides abundant parallelism.

(c) Heat diffusion (Heat): This program uses the Jacobi iterative method to solve an approximation of a partial differential equation that models the heat diffusion problem. The input includes a 2-dimension grid with the numbers of rows and columns, and the number of time steps (or iterations) the computation is performed on that 2D grid. Within each time step, the computation is carried out in a divide and conquer manner.

The Cholesky program was run for a matrix of size 3000×3000. The LU program was run for a matrix of size 2048×2048. For both of them, the base case matrix had size of 32×32. The Heat program was run with a 2-dimensional input of size 4096×1024 and 800 time steps. For each setting, we ran the program 100 times.

For each program, we first compare its execution times under work stealing and centralized scheduler on varying numbers of cores, as shown in Table 7.2. For all three benchmarks, we notice that the execution times are tight which means both scheduling strategies have a decent predictability. However, Cilk Plus implementations have smaller maximum execution times which means that Cilk Plus tasks have higher chance of finishing by their deadlines.

Figure 7.2 shows the speedups of these programs in the same experiments. For matrix computation programs like Cholesky and LU, where there is abundant parallelism, OpenMP obtains good speedups but Cilk Plus obtaines even better speedups. The difference is more notable in the Heat diffusion program, where there is less parallelism to exploit. For this program, Cilk Plus still has reasonable speedup, while the speedup of OpenMP starts to degrade when the number of cores is more than 21.

We also notice that for Cholesky and LU programs, the performances of OpenMP are quite sensitive to the base case sizes whereas Cilk Plus performed equally well regardless of the base case sizes. For demonstration, Figure 7.2(b) shows the experiment results of Cholesky with a base case matrix of size 4×4. Notably, no speedup was observed for OpenMP when the number of cores increases. Thus, one has to tune the base case size for OpenMP in order to get comparable performance with their Cilk Plus counterparts. This is

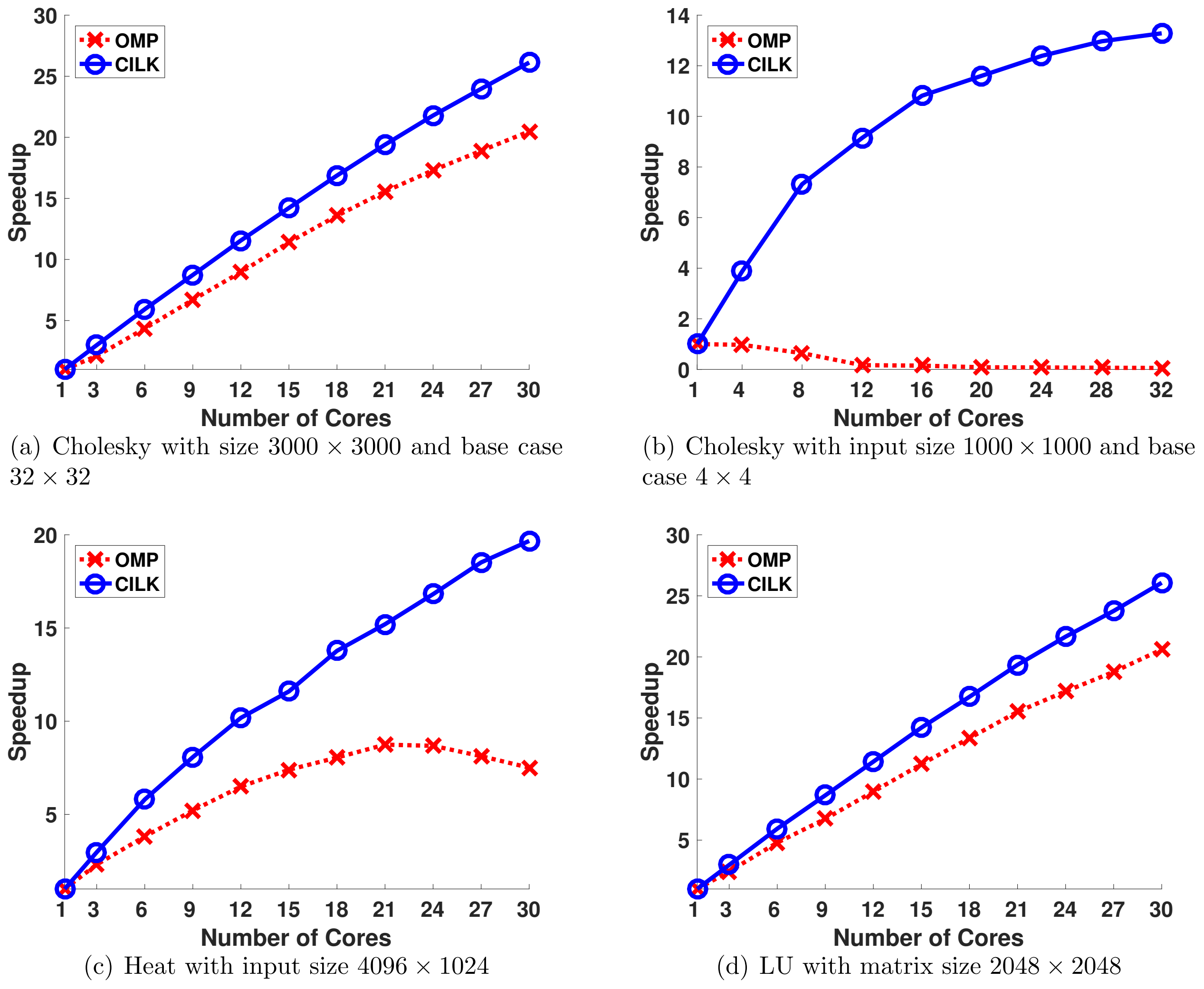
No. cores	Cholesky Factorization		
	OpenMP (med., max, 99^{th} per.)	Cilk Plus (med., max, 99^{th} per.)	Ratio
1	32.12, 32.18, 32.17	32.31, 32.36, 32.35	1.01
6	7.39, 7.62, 7.61	5.44, 5.47, 5.47	0.72
12	3.58, 3.72, 3.71	2.79, 2.89, 2.87	0.78
18	2.36, 2.43, 2.43	1.91, 1.96, 1.95	0.81
24	1.85, 1.92, 1.92	1.48, 1.52, 1.51	0.79
30	1.56, 1.62, 1.61	1.23, 1.28, 1.28	0.79
No. cores	LU Decomposition		
	OpenMP (med., max, 99^{th} per.)	Cilk Plus (med., max, 99^{th} per.)	Ratio
1	16.98, 17.09, 17.07	16.76, 16.82, 16.82	0.98
6	3.53, 3.79, 3.79	2.82, 2.84, 2.83	0.75
12	1.89, 1.97, 1.97	1.44, 1.87, 1.78	0.95
18	1.27, 1.37, 1.35	0.99, 1.07, 1.06	0.78
24	0.99, 1.06, 1.05	0.76, 0.84, 0.83	0.79
30	0.82, 0.86, 0.86	0.64, 0.71, 0.69	0.82
No. cores	Heat Diffusion		
	OpenMP (med., max, 99^{th} per.)	Cilk Plus (med., max, 99^{th} per.)	Ratio
1	51.57, 52.04, 52.04	51.70, 52.11, 52.11	1.00
6	13.50, 13.83, 13.81	8.80, 9.28, 9.26	0.67
12	7.93, 8.41, 8.31	5.06, 5.82, 5.70	0.69
18	6.40, 6.73, 6.69	3.73, 3.96, 3.95	0.59
24	5.94, 6.10, 6.10	3.06, 4.06, 3.67	0.67
30	6.87, 7.20, 7.17	2.62, 2.73, 2.73	0.38

Table 7.2: Median, maximum, and 99^{th} percentile execution times of Cholesky, LU, and Heat for OpenMP and Cilk Plus implementations (in seconds) and the ratio of the maximum execution times of Cilk Plus over OpenMP implementations.

again caused by the fact that the overhead of centralized scheduler adds up and outweighs the performance gain by running program in parallel, when the base case is small.

7.1.2 Tightness of Randomized Work Stealing in Practice

One might expect that even though a work-stealing scheduler may perform well on average due to low overheads, would not be suitable for real-time platforms due to high variability in its execution times due to randomness. However, this intuition turns out to be inaccurate.



(a) Cholesky with size 3000 × 3000 and base case 32 × 32

(b) Cholesky with input size 1000 × 1000 and base case 4 × 4

(c) Heat with input size 4096 × 1024

(d) LU with matrix size 2048 × 2048

Figure 7.2: Speedup of benchmark programs in OpenMP and Cilk Plus implementations

Theoretically, strong high probability bounds have been proven for the execution times for work stealing [33, 148]. Our experiments also suggest that the variation in execution time is small in practice. In our experiments, the difference between the mean execution time and the 99^{th} percentile execution time is less than 5% most of the times and the variation between the mean and the maximum execution time is also small.

More importantly, the variation shown by work stealing is never worse than (and is generally better than) that shown by the deterministic scheduler used by OpenMP. This indicates that work-stealing schedulers show promise for use in real-time systems, especially soft real-time systems which can tolerate some deadline misses, since they can potentially provide much better resource utilization than centralized schedulers for parallel tasks.

Discussion: One might wander whether a different centralized scheduler that builds on better synchronization primitives can outperform Cilk Plus's work-stealing scheduler. Our experiments in Figure 7.1(a), 7.1(b) and 7.2(b) indicate that the higher overhead of centralized scheduler mostly comes from the larger number of synchronization operations on the centralized global queue compared to lower contention on the distributed local queues. Therefore, even if synchronization primitives of the centralized scheduler is further optimized to reduce overheads, it is still unlikely to negate the inherent scalability advantages of randomized work-stealing, especially with increasing number of cores and workload complexity.

7.2 Adaptation to Federated Scheduling using Work Stealing

As demonstrated in Section 7.1, a centralized greedy scheduler incurs high overheads for parallel tasks and is less scalable compared to a randomized work-stealing scheduler. Therefore, when the task set allows occational deadline misses, using randomized work stealing can be more resource efficient and scalable. In order to leverage work stealing, while providing soft real-time performance to parallel task sets, we adapt federated scheduling to incorporate work stealing overhead. In this section, we first briefly introduce federated scheduling and then present how we can adapt it to incorporate work stealing overhead.

7.2.1 Federated Scheduling for Parallel Real-Time Tasks

The federated scheduling in Chapter 4 is an scheduling paradigm for parallel real-time tasks. Given a task set τ, federated scheduling either *admits* a task set and outputs a **core assignment** for each task; or declares the task set to be unschedulable. In the core assignment, each high-utilization task (utilization > 1) is allocated n_i dedicated cores, where $n_i = \left\lceil \frac{C_i - L_i}{D_i - L_i} \right\rceil$. During runtime, a *greedy* scheduler is required to execute each high-utilization task on its dedicated cores. All the low-utilization tasks are forced to execute sequentially on the remaining cores scheduled by a multiprocessor scheduling algorithm. Since low-utilization tasks do not need parallelism to meet deadlines, in this work we focus only on high-utilization tasks.

Federated scheduling has been proved to have a **capacity augmentation bound** of 2, meaning that given m cores federated scheduling can guarantee schedulability to any task set τ as long as it satisfies: (1) the total utilization of the task set is no more than half of the total available cores – $\sum u_i \leq m/2$; (2) for each task, the critical-path length is less than half of its deadline – $L_i < D_i/2$.

Discussion: Why do we choose to integrate work stealing into federated scheduling instead of other real-time schedulers, such as global EDF? Firstly, as discussed in related work, federated scheduling has the best capacity augmentation bound of 2, so it can schedule task sets with higher load than other scheduler strategies. More importantly, it has the benefit that the parallel scheduler used for executing parallel task does not need to be deadline- or priority-aware, since the task is assigned with dedicated cores. Otherwise, worker threads of the parallel scheduler have to be able to quickly switch to jobs with shorter deadlines or higher priorities when they arrive. However, recall that the advantage of work stealing is that a worker thread works off its own deque most of the time, which is against the requirement of fast switching between jobs. Because of this, implementing other parallel real-time scheduling strategies using work stealing can be difficult and involve high overheads.

7.2.2 Incorporating Work Stealing Overhead into Federated Scheduling

When each parallel task is executed by a work-stealing scheduler on its dedicated cores, the core assignment of federated scheduling needs to incorporate work stealing overheads when calculating core assignment. In work stealing, there are two types of overheads that we need to consider: stealing overhead and randomization overhead.

Stealing overheads includes the explicit costs of bookkeeping parallel nodes and the implicit cost of cache misses due to the migration of stolen nodes. Since stealing can only occur at spawn and sync points in the DAG, given a specific DAG one can estimate the overheads due to scheduling events by counting these quantities. Based on this insight, *burdened DAG* is introduced to estimate stealing overheads for DAG task [85]. Using profiling tools, such as Cilk View and Cilkprof [139], the *burdened critical-path length* $\hat{L}_i$, with stealing overheads incorporated, can be measured. When calculating core assignment for tasks, we use the burdened critical-path length $\hat{L}_i$ to replace critical-path length L_i, so that stealing overhead is included.

Compared to a greedy scheduler, the randomness of work stealing introduces additional overhead. In particular, even if there is available work on another core, a core may still take some time to find the available work, because the random stealing may fail to find the available work. Thus, the execution time of a task under work stealing is a random variable. By extending the result of stochastic federated scheduling in Chapter 6, we incorporate the randomization overhead into core assignment and also analyze the expected tardiness bound for federated scheduling using work stealing.

To analyze the randomness overhead, we first state known results on work-stealing response time γ_i for task τ_i with total execution time C_i and critical path-length L_i [148].

Lemma 48 [Tchi.13] *A work-stealing scheduler guarantees completion time γ_i on n_i dedicated cores where*

$$\mathrm{E}\left[\gamma_i\right] \leq \tfrac{C_i}{n_i} + \delta L_i + 1 \tag{7.1}$$

$$\mathbb{P}\left\{\gamma_i \leq \tfrac{C_i}{n_i} + \delta(L_i + \log_2 \tfrac{1}{\epsilon}) + 1\right\} \geq 1 - \epsilon \tag{7.2}$$

Note that the δ in the above formula is the critical-path length coefficient. Theoretically it has been proven to be at most 3.65, while empirically it is set to 1.7 for measurement using Cilk View [85] and is set to 1.5 when using Cilkprof.

Consider a random variable X with CDF function

$$\mathbb{F}(x) = \mathbb{P}\{X \leq x\} = 1 - e^{-\lambda(x-\mu)}$$

where $\mu = \frac{C_i}{n_i} + \delta L_i + 1$ and $\lambda = \frac{\ln 2}{\delta}$. This is the CDF of a shifted exponential distribution with mean value $\mathrm{E}[X] = \mu + \frac{1}{\lambda} = \frac{C_i}{n_i} + \delta(L_i + \frac{1}{\ln 2}) + 1$ and variance λ^{-2}.

If we set $x = \mu + \delta \log_2 \frac{1}{\epsilon}$, then we get $\epsilon = e^{-\lambda(x-\mu)}$. Using Inequality (7.2), the above CDF can be rewritten as

$$\begin{aligned}\mathbb{F}(x) =& \mathbb{P}\left\{X \leq \frac{C_i}{n_i} + \delta(L_i + \log_2 \frac{1}{\epsilon}) + 1\right\} = 1 - \epsilon \\ \geq& \mathbb{P}\left\{\gamma_i \leq \frac{C_i}{n_i} + \delta(L_i + \log_2 \frac{1}{\epsilon}) + 1\right\}\end{aligned}$$

Therefore, the CDF of random variable X is the upper bound of the CDF of completion time γ_i. Every instance j drawn from the distribution of γ_i can be mapped to an instance in the distribution of X that is no smaller than j. In other words, X's probability density function of $f(x) = \lambda e^{-\lambda(x-\mu)}$ is the worst-case distribution of completion time γ_i of task τ_i under work stealing.

Now we can use a lemma from queueing theory [98] to calculate core assignment and bound the response time for federated scheduling incorporated with work stealing.

Inspired by the stochastic analyses in [122] and Section 6.3.3, Lemma 38 in Section 6.3.3 can be interpreted as follows: parallel jobs are customers; implicit deadline is the inter-arrival time $Y = D_i$; and the completion time on n_i dedicated cores using work stealing is the service time $\mathcal{X} = \gamma_i$. As discussed above, $f(x)$ is the worst-case distribution of γ_i with mean value $\frac{C_i}{n_i} + \delta(L_i + \frac{1}{\ln 2}) + 1$. Thus, Lemma 38 guarantees bounded response time for

$n_i > \frac{C_i}{D_i - \delta(L_i + \frac{1}{\ln 2}) - 1}$, since

$$\mathrm{E}\left[\mathcal{X}\right] = \mathrm{E}\left[\gamma_i\right] \leq \frac{C_i}{n_i} + \delta(L_i + \frac{1}{\ln 2}) + 1 < D_i = Y$$

Therefore, after incorporating the stealing overhead and randomness overhead into federated scheduling, the number of cores assigned to a task is adapted as

$$n_i = \left\lfloor \frac{C_i + D_i - \delta\hat{L}_i}{D_i - \delta\hat{L}_i} \right\rfloor \tag{7.3}$$

Note that we omitted the terms $\frac{1}{\ln 2}$ and 1, because they are in unit time step, which is negligible compared with C_i and L_i in actual time. If $D_i \leq \delta\hat{L}_i$, the task is deem unschedulable.

From Lemma 38, we can also calculate the bound on task expected response time R_i. Again as $f(x)$ is the worst-case distribution of γ_i with variance $(\frac{\delta}{\ln 2})^2$, given n_i cores the expected response time of task τ_i is bounded by

$$\mathrm{E}\left[R_i\right] \leq \mathrm{E}\left[\gamma_i\right] + \frac{\delta^2_{\gamma_i}}{2(D_i - \mathrm{E}\left[\gamma_i\right])} \leq \frac{C_i}{n_i} + \delta\hat{L}_i + \frac{(\frac{\delta}{\ln 2})^2}{2(D_i - \frac{C_i}{n_i} - \delta\hat{L}_i)}$$

7.3 RTWS Platform

In this section, we describe the design of the **RTWS** platform, which provides federated scheduling service for parallel real-time tasks. which

Note that RTWS and RTCG have the same API and similar design. Because each parallel task is executed on dedicated cores, and no other tasks can introduce CPU interference with them. Therefore, the parallel scheduler does not need to be deadline- or priority-aware. This design allows us to use existing Cilk Plus runtime systems to schedule parallel tasks.

Here we point out the major differences in the RTWS platform from RTCG. The RT-scheduler in RTWS calculates a core assignment using the formula (7.3) in Section 7.2

during offline, which has incorporated work stealing overheads into federated scheduling. During execution, the PL-dispatcher enforces the periodic invocation of each task and calls an individual GNU Cilk Plus runtime system to provide parallel execution of each task. To isolate multiple concurrent parallel runtime systems from each other, we modified its runtime system, so that each Cilk Plus runtime only creates n_i workers, each of which is pinned to one of the n_i assigned cores.

Profiling Tool: Since the work and critical-path length of each task must be specified to the platform (in the configuration file), we also provide a simple profiling utility to automatically measure these quantities for each task. The work of a task can be measured by running the profiling program on a single core. Measuring the critical-path length is more difficult. We adopt a profiling tool Cilkprof [139], which can automatically measure the work and the burdened critical-path length of a single job. In particular, Cilkprof uses compiler instrumentation to gather the execution time of every *call site* (i.e., a node in the DAG) and calculate the critical-path length in nanosecond. To be consistent with GNU Cilk Plus (and GNU OpenMP), we use a version of Cilkprof that instrumented the GCC compiler and incorporated the burdened DAG into the measurement. Intel provides another tool Cilkview [85] that can measure the number of instructions of burdened critical-path length using dynamic binary instrumentation.

Discussion: As shown in Section 7.1, work stealing has better parallel performance than the centralized scheduler. Thus, RTWS using work stealing is a better candidate for parallel tasks with soft real-time constraints, as confirmed via the empirical comparison in Section 7.4. However, it may not be the best approach for other scenarios. First and foremost, the execution time of a parallel task using work stealing can be as slow as its sequential execution time in the worst case, even though the probability of the worst case happening can be extremely low in practice. Therefore, it can never be applied to hard real-time systems without modifying the work stealing protocol to provide some form of progress guarantee.

In addition, for special purposed system where the structure of parallel task is static and well measured, a static scheduler that decides how to execute the parallel task prior to execution can effectively reduce scheduling overheads and may perform better than work stealing.

7.4 Platform Evaluation

In this section, we evaluate the soft real-time performance provided by RTWS using a randomized work-stealing scheduler (**RTWS**) compared to the alternative implementation of federated scheduling using a centralized greedy scheduler (**RTCG**). We use three DAG applications written in both Cilk Plus and OpenMP (discussed in Section 7.1) to randomly generate task sets for emperical experiments. To the best of our knowledge, RTWS is the first real-time platform that supports general DAG tasks, such as these benchmark programs. Since other existing real-time systems do not support parallel DAG tasks, we do not compare against them.

Experiments were conducted on a 32-core machine composed of four Intel Xeon processors (each with 8 cores). When running experiments, we reserved two cores for operating system services, leaving 30 experimental cores. Linux with CONFIG_PREEMPT_RT patch version r14 applied was the underlying RTOS.

7.4.1 Benchmark Task Sets Generation

We now describe how we generate task sets composed of the three benchmark programs (Cholesky, Heat and LU) with the general DAG structures. We generate 4 sets of task sets and evaluate their performances. The first 3 sets are composed with tasks running the same application, denoted as **Cholesky**, **Head** and **LU task sets**. The last set comprises a mix of all benchmarks, denoted as **Mixed task sets**.

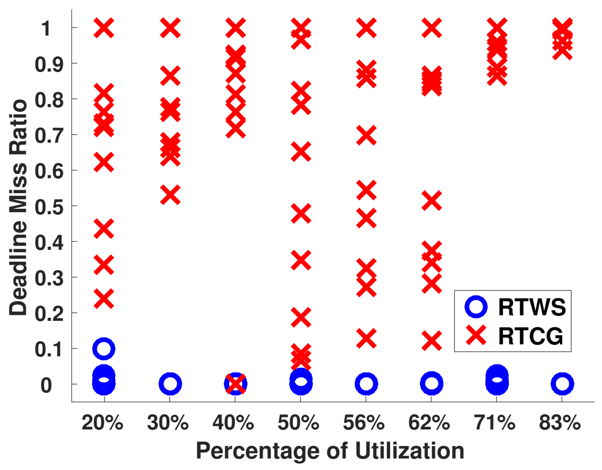
We profile Cholesky, Heat and LU programs using 14, 6 and 3 different input sizes, respectively. For each program with each input size, we measure its work and burdened critical-path length using Cilkprof. Then we generate different tasks (from one benchmark with one input size) and assign it with a randomly generated utilization. To see the effect of scalability of large parallel tasks (i.e., spanning many cores), we intentionally create 5 types of tasks: tasks with mean utilization from {1, 3, 6, 12, 15}. When assigning utilization to a task, we always try to pick the largest mean utilization that does not make the task set utilization exceed the total utilization that we desire. After deciding a mean utilization, we will then randomly generate the utilization of the task using the mean value. A task's period is calculated using its work over utilization. We keep adding tasks into the task set, until it reaches the desired total utilization. For each setting, we randomly generate 10 task sets.

7.4.2 Evaluation Results

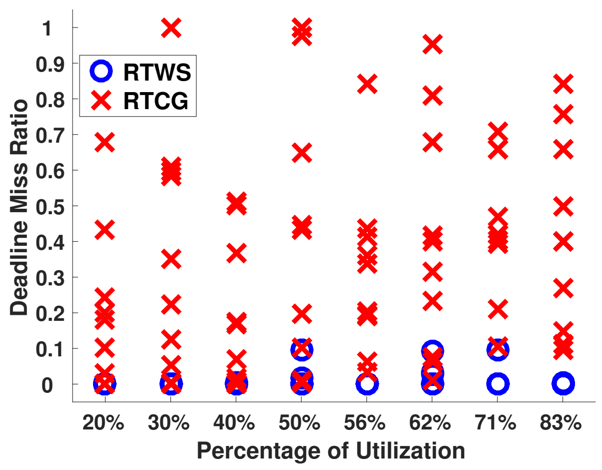
For each DAG task set, we record the **deadline miss ratio**, which is calculated using the total number of deadline misses divided by the total number of jobs in the task set. We also record the response time of each individual job during the execution to calculate a **relative response time**, which is the job's response time over its deadline. We then calculate the average relative response time for each task set.

In the first two comparisons between RTWS and RTCG, we'd like to see how the integration of federated scheduling and randomized work stealing performs compared with federated scheduling using a centralized greedy scheduler *given the same resource capacity* for soft real-time task sets. Therefore, for these experiments we use the *same core assignment* as described in Section 7.2, which incorporates work stealing overheads into federated scheduling.

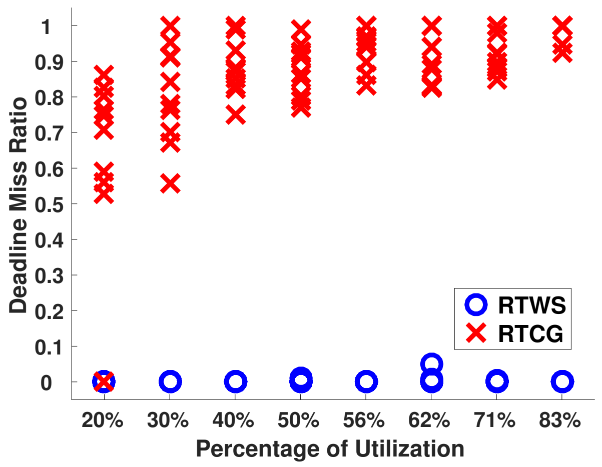
Since the centralized scheduler generally has larger overheads and takes longer to execute as shown in Section 7.1, it is not surprising to see that RTCG performs worse than RTWS given the same resource capacity. To further analyze the performance difference between the two approaches, in the last experiment we increase the resource capacity for RTCG. We'd like to see how much more resource capacity RTCG requires in order to *schedule the same task sets* compared with RTWS.



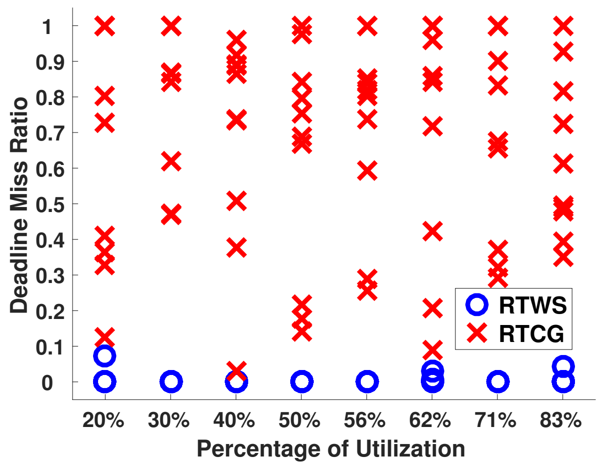
(a) Deadline miss ratio of Cholesky task sets



(c) Deadline miss ratio of LU task sets



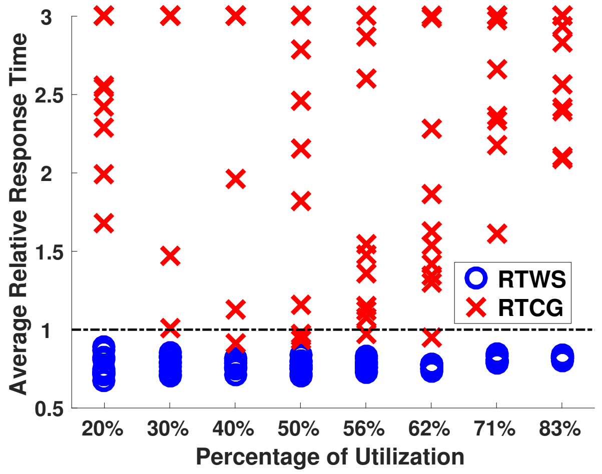
(b) Deadline miss ratio of Heat task sets



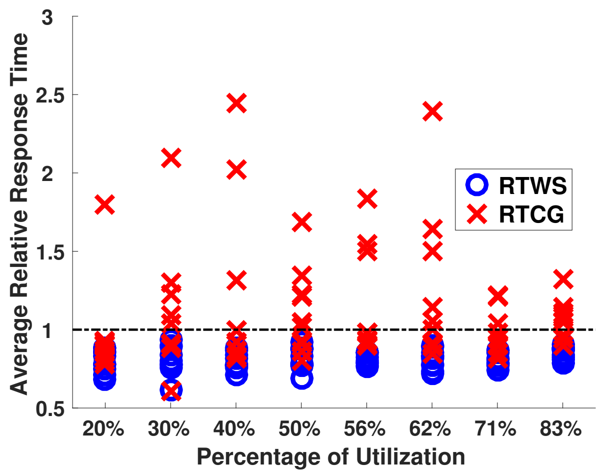
(d) Deadline miss ratio of Mixed task sets

Figure 7.3: Deadline miss ratio of different task sets (Cholesky, Heat, LU and Mixed task sets) with increasing total utilization under RTWS (providing federated scheduling service integrated with a randomized work-stealing scheduler in GNU Cilk Plus) and RTCG (providing federated scheduling service integrated with a centralized greedy scheduler in GNU OpenMP). In these experiments, RTWS and RTCG use the same core assignment.

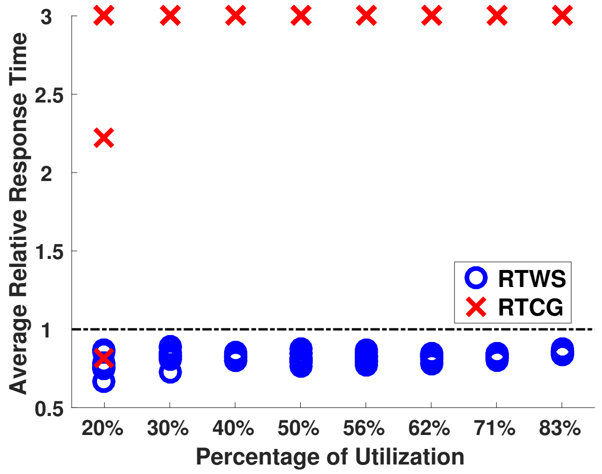
(1) Deadline miss ratio comparison: We first compare the deadline miss ratio in Figure 7.3(a),7.3(b),7.3(c) and 7.3(d) for Cholesky, Heat, LU and Mixed task sets, respectively. Notably, most of the task sets under RTWS has no deadline misses and all of the task sets have a deadline miss ratio no more than 10%. In fact, from all the experiments we run, there are only 2.25% tasks (28 out of 1243 tasks) having deadline misses. In contrast, given the same core assignment RTCG misses substantially more deadlines, especially for Heat task sets where many tasks miss all of their deadlines.



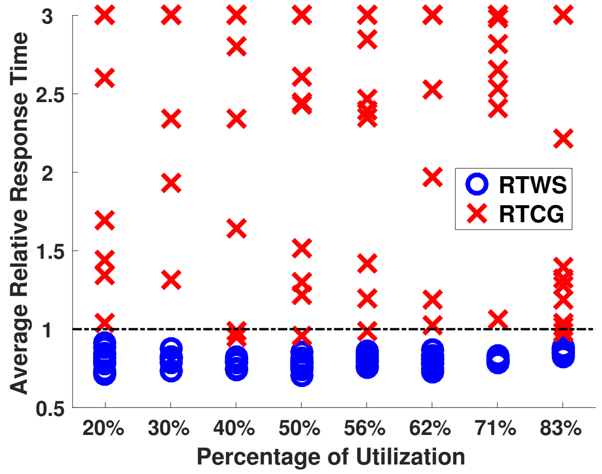
(a) Relative Response time of Cholesky task sets



(c) Relative Response time of LU task sets



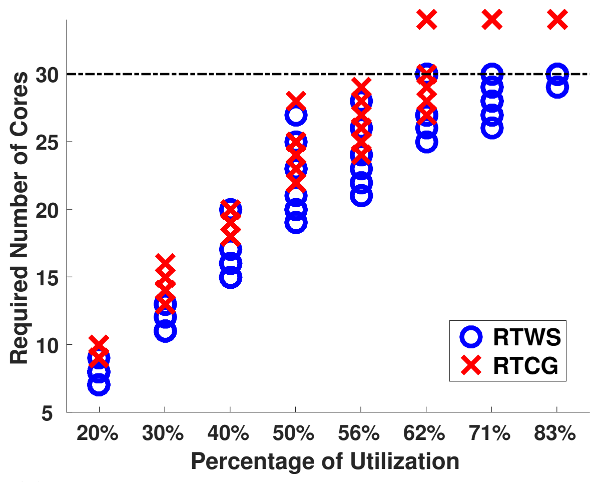
(b) Relative Response time of Heat task sets



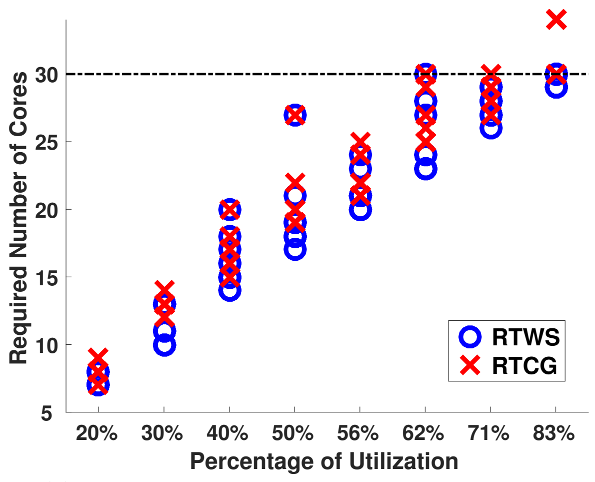
(d) Relative Response time of Mixed task sets

Figure 7.4: Average relative response time of different task sets (Cholesky, Heat, LU and Mixed task sets) with increasing total utilization under RTWS and RTCG. In these experiments, RTWS and RTCG use the same core assignment.

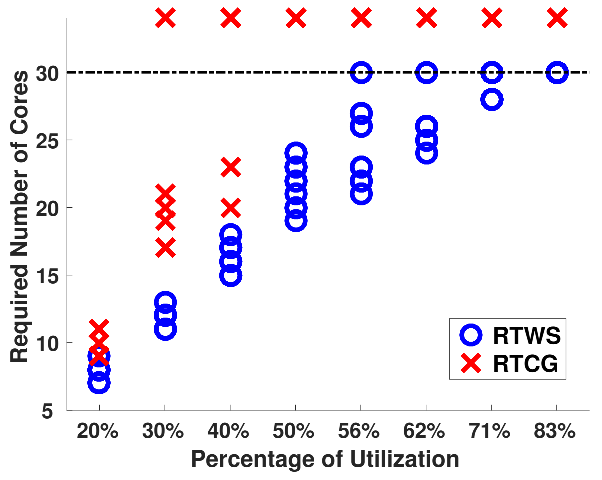
(2) Relative response time comparison: In Figure 7.4(a),7.4(b),7.4(c) and 7.4(d), we observe that RTCG has much higher average relative response time than RTWS, given the same resource capacity. For all task sets, the average relative response time of RTWS is less than 1, while some tasks under RTCG even have relative response times larger than a hundred. In order to clearly see the relative response times smaller than 1, when plotting the figures we mark all the relative response times that are larger than 3 as 3.



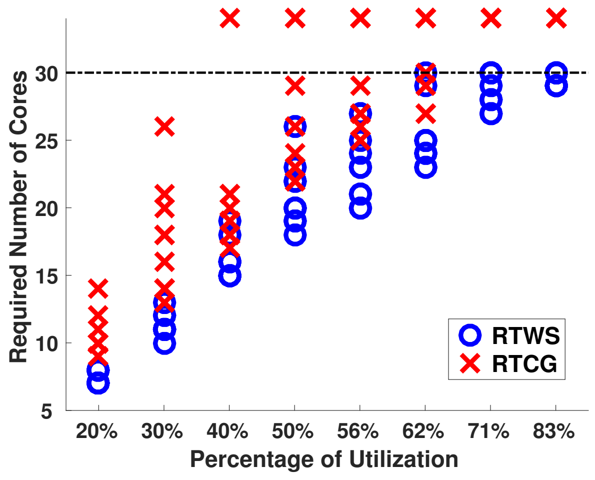
(a) Required number of cores of Cholesky task sets



(c) Required number of cores of LU task sets



(b) Required number of cores of Heat task sets



(d) Required number of cores of Mixed task sets

Figure 7.5: Required number of cores of different task sets (Cholesky, Heat, LU and Mixed task sets) with increasing total utilization under RTWS and RTCG. In these experiments, we increase the number of cores for each task under RTCG until it misses no more than 60% of deadlines.

(3) Required resource capacity: From the first two comparisons, we can clearly see that RTCG requires more cores (i.e., resource capacity) in order to provide the same real-time performance as RTWS. Thus, in Figure 7.5(a),7.5(b),7.5(c) and 7.5(d) we keep increasing the number of cores assigned to tasks under RTCG that have more than 25% of deadline misses. Note that all tasks under RTWS meet at least 80% of deadlines. We compare the required number of cores of RTCG and RTWS for the same task sets to meet most of their deadlines. If a task set misses most deadlines when allocated with all the 30 available cores, then we mark the number of required cores as 34. For Cholesky and LU task sets, RTCG requires about 1 to 3 additional cores. For some Heat task sets, even doubling the number of cores for RTCG is still not sufficient.

Part II

Online Systems with Parallel Latency-Critical Jobs

Introduction

This part of the thesis focus on the online scheduling problems for parallel latency-critical jobs. In these systems, jobs arrive over time and the scheduler does not know the existence of jobs until they arrive. The goal of the scheduler is both to use the resources efficiently and to provide a good quality of service to jobs.

Chapter 8 focuses on the performance metric of minimizing the number of jobs that miss a target latency. It shows how to generalize work-stealing, which is traditionally used to minimize the makespan of a single parallel job, to optimize for a target latency in interactive services with multiple parallel jobs. Our experiments indicate that our prototype work-stealing scheduler using TBB is very effective for highly parallel interactive workloads. Although no current interactive service uses work stealing, as far as we are aware, the work stealing framework is appealing because it supports the most general models of both static and dynamic parallelism. To use our framework, services need only to express the parallelism in individual requests. The benefits of the proposed scheduling strategy include improved user experience with more consistent responsiveness and increased system capacity for interactive services.

Chapter 9 focuses on scheduling parallel jobs online to maximize the throughput or profit of the execution. Specifically, in an execution a set of n jobs arrive online and each job J_i has an associated function $p_i(t)$, the profit obtained for finishing job J_i at time t. Each job has its own arbitrary non-increasing profit function. This chapter gives the first non-trivial results for the profit scheduling problem for DAG jobs.

Chapter 10 studies the problem of scheduling a set of parallel jobs with the objective of minimizing the maximum latency experienced by any job. In this setting, jobs arrive online and the scheduler has no information about the arrival rate, arrival time or work distribution of the jobs. The scheduling goal is to minimize the maximum amount of time between the arrival of a job and its completion — this goal is referred to in scheduling literature as maximum flow time. In addition, this chapter also considers the case where jobs have weights (typically representing priorities) and the objective is minimizing the maximum weighted flow time.

Chapter 11 studies the problem of scheduling parallel jobs online with an objective of minimizing average flow time. For this objective, we present a scalable algorithm which is $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^3})$-competitive for any $\epsilon > 0$. We further introduce the first greedy algorithm for scheduling parallel jobs — our algorithm is a generalization of the shortest jobs first algorithm. Greedy algorithms are among the most useful in practice due to their simplicity. We show that this algorithm is $(2+\epsilon)$-speed $O(\frac{1}{\epsilon^4})$-competitive for any $\epsilon > 0$.

Chapter 12 also considers the problem of minimizing average flow time (latency) of parallel jobs online on multicore machines. Although the algorithms proposed in Chapter 11 have good theoretical performance, they are impractical to implement. Therefore, this chapter proposed a new distributed scheduling algorithm that has the same theoretical performance as the greedy algorithm in Chapter 11. This distributed algorithm has several practical advantages: (1) it is a non-clairvoyant algorithm — it does not know the execution characteristics of jobs. (2) The scheduling algorithm consists of a simple protocol where all processors make scheduling decisions in a distributed manner requiring minimal synchronization. (3) It requires a small bounded number of preemptions.

Contents of this part of the thesis have appeared in the following publications:

- Kunal Agrawal, Jing Li, Kefu Lu, and Benjamin Moseley. Scheduling Parallel DAG Jobs Online to Minimize Average Flow Time, ACM-SIAM Symposium on Discrete Algorithms (SODA'16), January 2016.

- Jing Li, Kunal Agrawal, Sameh Elnikety, Yuxiong He, I-Ting Angelina Lee, Chenyang Lu, and Kathryn S. McKinley. 2016. Work stealing for interactive services to meet target latency. In Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'16), March 2016.

- Kunal Agrawal, Jing Li, Kefu Lu, and Benjamin Moseley. 2016. Scheduling Parallelizable Jobs Online to Minimize the Maximum Flow Time. In Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'16), July 2016.

- Kunal Agrawal, Jing Li, Kefu Lu, and Benjamin Moseley. 2017. Brief Announcement: Scheduling Parallelizable Jobs Online to Maximize Throughput. In Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'17), July 2017.

Chapter 8

Scheduling Parallel Jobs Online to Meet Target Latency

Delivering consistent interactive latencies is the key performance metric for interactive cloud services, such as web search, stock trading, ads, and online gaming. The services with the most fluid and seamless responsiveness incur a substantial competitive advantage in attracting and captivating users over less responsive systems [56, 57, 83, 155]. Many such services are deployed on cloud systems that span hundreds or thousands of servers, where achieving interactive latencies is even more challenging [91,133,135,154,156]. The seemingly infrequent occurrence of high latency responses at one server is significantly amplified because services aggregate responses from large number of servers. Therefore, these servers are designed to minimize tail latency, i.e., the latency of requests in the high percentiles, such as 99th percentile latency. This chapter studies a closely related problem, optimizing for a **target latency** — given a target latency, the system minimizes the number of requests whose latency exceeds this target.

This chapter explores interactive services on a multicore server with multiple parallel requests. Interactive service workloads tend to be computationally intensive with highly variable work demand [56, 82, 83, 91, 94, 156]. A web search engine, for example, represents, partitions, and replicates billions of documents on thousands of servers to meet responsiveness requirements. Request work is unknown when it arrives at a server and prediction is unappealing because it is never perfect or free [90, 97, 118]. The amount of work per request varies widely. The work of the 99th percentile requests is often larger than the median by orders of magnitude, ranging from 10 to over 100 times larger. Moreover, the workload is highly parallelizable — it has **inter-request parallelism** (multiple distinct requests execute simultaneously) and fine-grain **intra-request parallelism** [82, 91, 156]. For instance, search could process every document independently. More generally, the parallelism of these requests may change as the request executes.

Many modern concurrency platforms support scheduling a single parallel program, including Cilk Plus [89], Java Fork/Join [104], OpenMP [125], Task Parallel Library [107], Threading Building Blocks (TBB) [134], and X10 [147]. Most of these platforms use a **work-stealing scheduler**, which is efficient both in theory and in practice [8,32,33]. It is a distributed scheduling strategy with low scheduling overheads. The goal of traditional work stealing is to deliver the smallest possible makespan for one job. In contrast, interactive services have multiple requests (jobs) and the goal is to meet a target latency. To exploit the benefits of work-stealing strategy for interactive services, in this chapter, we modify work stealing so it can optimize for target latency.

Designing such a strategy faces several challenges. (1) When a request arrives in the system, the request's work is unknown. (2) Which requests to sacrifice depends on the request's progress, the target latency, and current load. Serializing large request can limit their impact on queuing delay of waiting small requests and enabling more requests to meet the target latency. However, serialized large requests will almost certainly miss the target latency and thus we must not unnecessarily serialize large requests. (3) In interactive services, we know the expected average system load and workload distribution, but the instantaneous load can vary substantially due to unpredictable request arrival patterns. Whether a particular request should be serialized depends on its work and the instantaneous load. If the system is heavily loaded, even a moderately large request may increase the waiting time of many small requests. Therefore, we must be more aggressive about serializing requests. When the system is lightly loaded, we should be less aggressive and execute even very large requests in parallel so they can also meet the target latency. In short, there is no fixed threshold for classifying a request as large and serializing it; rather, the runtime should adapt the threshold based on instantaneous system load.

Based on these intuitions, this chapter introduces the **tail-control** work-stealing scheduling strategy for interactive workloads with multiple simultaneous requests. Tail-control continuously monitors the system load, estimated by the number of active requests. It aggressively parallelizes requests when the system is lightly loaded. When the system is momentarily overloaded, it identifies and serializes large requests to limit their impact on other waiting requests. Specifically, the contributions of this chapter are as follows:

(1) We design an offline **threshold-calculation algorithm** that takes as input the target latency, the work demand distribution, and the number of cores on the server, and calculates a *large request threshold* for every value of the instantaneous load. It generates offline a table indexed by system load that the online scheduler uses to decide which requests to serialize.

(2) We extend work stealing to handle multiple requests simultaneously and incorporate the tail-control strategy into the stealing protocol. We perform bookkeeping to track a) the instantaneous system load; and b) the total work done by each request thus far. Most bookkeeping is done distributedly, which amortizes its overhead with steal attempts. Our modified work-stealing scheduler add no constraints on the programming model and thus can schedule any dynamic multithreaded program.

(3) We implement a tail-control work-stealing server in the Intel Thread Building Block (TBB) runtime library [134] and evaluate the system with several interactive server workloads: a Bing search workload, a finance server workload [83], and synthetic workloads with long-tail distributions. We compare tail-control with three baseline work-stealing schedulers, *steal-first*, *admit-first*, and default TBB. The empirical results show that tail-control significantly outperforms them, achieving up to a 58% reduction in the number of requests that exceed the target latency.

Section 8.1 provides background information about parallelism in interactive service requests and simple work stealing extensions to online multiple jobs scheduling. Based upon the intuitions of minimizing tail latency presented in Section 8.2, we provide the specific threshold calculation in Section 8.3.1 and present our algorithm design in Section 8.3.2. Sections 8.4 presents experimental evaluations using synthetic workload based on real search request workload.

8.1 Background and Terminology

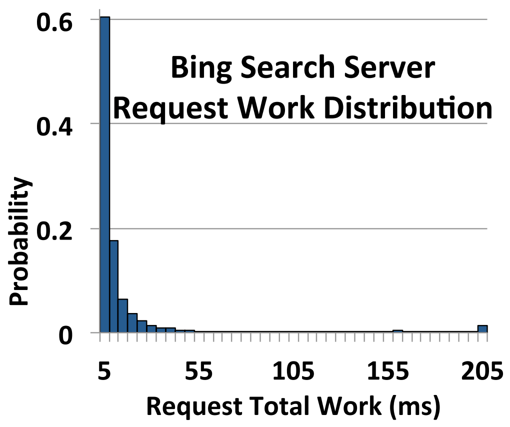
This section defines terminology used throughout this chapter and characterizes interactive services and available intra-request parallelism.

8.1.1 Terminology

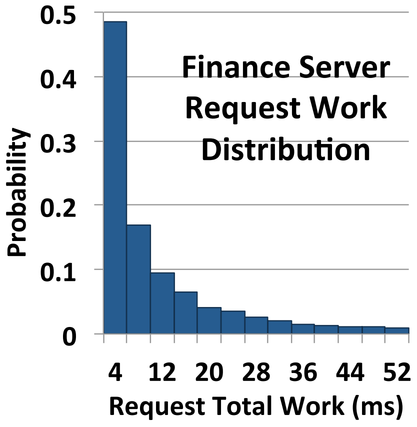
The **response time (latency)** of an interactive service request is the time elapsed between the time when the request **arrives** in the system and the time when the request completes. Once a request arrives, it is **active** until its completion. A request is **admitted** once the system starts working on it. An active request is thus either executing or waiting. Since the service may perform some or all of the request in parallel to reduce its execution time, we define request **work** as the total amount of computation time that all workers spend on it. A request is **large** or **small** according to its relative total work. A request **misses** a specified target latency if its latency is larger than the target.

8.1.2 Characteristics of Interactive Services

Three important characteristics of interactive services inform our scheduler design. First, many interactive services, such as web search, stock trading, online gaming, and video streaming are *computationally intensive* [56,83,91,94]. For instance, banks and fund management companies evaluate thousands of financial derivatives every day, submitting requests that value derivatives and then making immediate trading decisions — many of these methods use computationally intensive Monte Carlo methods [38, 53]. Second, the work demand per request can be highly variable. Figure 8.1 shows representative request work distributions for Bing and a finance server. Many requests are small and the smallest are more than 40 times smaller than the largest for Bing and 13 for the finance server. Third, the requests often have internal parallelism and each request can potentially utilize multiple cores.



(a) Bing search



(b) Finance server

Figure 8.1: Work distribution of two interactive services: Bing search server [97] and an option pricing finance server [135]. Note that in Bing search server the probabilities of requests with work between 55ms to 200ms are small but larger than zero and total probability of these requests is around 3.5%.

8.2 Intuitions for Tail-Control

This section describes intuitions for reducing the number of requests that miss a target latency. We first review results from scheduling sequential requests to derive inspiration and then show how to adapt them to work stealing for parallel requests.

Theoretical Results on Minimizing Tail Latency.

The theoretical results for online scheduling of *sequential* requests on multiple processors indicate that no single scheduling strategy is optimal for minimizing tail latency [35, 152]. When a system is lightly loaded or when all requests have similar execution times, a **first-come-first-serve (FCFS)** strategy is asymptotically optimal [152]. The intuition is that under these circumstances, the scheduler should minimize the execution time of each request, and FCFS does exactly that. However, under heavily loaded systems and when requests have high variability in work, FCFS is not optimal, since under high loads, large requests can delay many small requests by monopolizing processing resources. Under these circumstances, schedulers that prevent large requests from monopolizing processing resources perform better. Examples include a **processor sharing** scheduler [99] that divides up the processing resources equally among all requests and an **SRPT-like** (shortest remaining processing time) scheduler [140] that gives priority to small requests.

In interactive services, the request work is often highly variable, and the instantaneous load varies as the system executes. When the instantaneous load is low, an approximation of FCFS will work well. However, when the instantaneous load is high, the system should prevent large requests from delaying small requests. We now see how we can apply these intuitions to a work-stealing scheduler.

Baseline Variations of Work Stealing for Interactive Services.

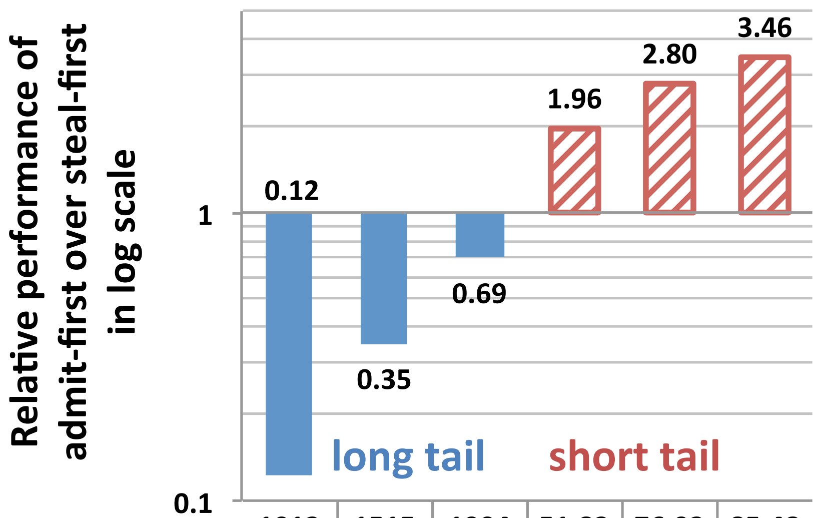
Since traditional work-stealing does not handle multiple requests, we first make a simple modification to work stealing to handle dynamically arriving multiple requests. We add a global FIFO queue that keeps all the requests that have arrived, but have not started executing on any worker. A request is admitted when some worker removes it from the global queue and starts executing it.

Now consider how to achieve FCFS in work stealing for multiple parallel requests. To minimize execution time, we want to exploit all the parallelism — which leads to the default **steal-first** work-stealing policy. We modify steal first for the server setting as follows. When a worker runs out of work on its deque, it still randomly steals work from victims, but if there is no stealable work, then it admits a new request. Intuitively, steal-first minimizes the execution time of a request, because it executes requests with as much parallelism as the application and hardware support and only admits new requests when fine-grain parallelism of already admitted requests is exhausted. However, we find it has the same weakness as FCFS — when the system is heavily loaded, a large request can delay many small requests. In fact, this effect is amplified since in steal first for the server setting because a large request with ample parallelism may monopolize *all* of the processors.

A simple and appealing strategy for high load is **admit-first**, where workers preferentially admit new requests and only steal if the global queue is empty. Intuitively, admit first minimizes queuing delay, but each request may take longer, since the system does not fully exploit available fine-grain parallelism.

Motivating Results.

We perform a simple experiment that confirms these intuitions. We create two workloads with only small and large requests, both with ample software parallelism (more inter-request parallelism than cores). The first workload, named "short tail", contains two kinds of requests: 98% are small requests and 2% medium requests with 5 times more work than the small ones. The second workload, named "long tail", contains 98% small requests and 2% large requests with 100 times more work than the small ones. Both systems have an average utilization of 65% and the request arrival rate follows the Poisson distribution. Therefore, even though the average utilization is modest, on occasion, the system will be under- and over-subscribed due to the variability in arrival patterns. We run both workloads 10,000 requests on a 16-core machine (see Section 8.4 for hardware details).



Target latency (ms)	1013	1515	1904	51.83	76.93	85.48
AF: Admit-First miss ratio	5.0%	1.0%	0.5%	9.8%	2.8%	1.7%
SF: Steal-First miss ratio	40.7%	2.9%	0.7%	5.0%	1.0%	0.5%
Relative performance = AF / SF	0.12	0.35	0.69	1.96	2.80	3.46

Figure 8.2: Neither steal-first nor admit-first always performs best. Each bar plots a target latency and the ratio (on a log scale) of requests that miss the target latency under admit-first over those that miss the target latency under steal-first. When the bar is above 1, steal-first is better. Below 1, admit-first is better.

Figure 8.2 shows the results. When all requests are about the same length, the short-tail workload shown in hashed red bars, steal first works better since it approximates FCFS and processes requests as quickly as it can. On the other hand with the long-tail workload, steal-first performs poorly due to the relative difference in large and small requests — large requests delay many small requests.

Tail-Control Algorithm Overview.

Since the workloads for interactive services have a range of time varying demand, we design a scheduler that adapts dynamically based on the instantaneous load. Under low load, we use steal-first to maximize parallelism, executing each request as fast as possible. Under high load, we wish to minimize the impact of large requests on the waiting time of small requests. Therefore, we execute large requests sequentially (thereby giving up on them, making them unlikely to achieve the target latency) and use steal-first for the other requests. At any moment, say k requests of work greater than l are serialized. Given m workers, the remaining $m-k$ workers execute all the remaining requests of work smaller than l in parallel. Thus, the policy effectively converts a high-load, high-variable-work instance on m workers into a low-variable-work instance on $m-k$ workers. Since steal-first is a good policy for low-variable-work instances, we get good performance for the requests that were not serialized.

Note that we have been playing fast and loose with the term large request. There is no fixed threshold which determines if a request is large or not — this threshold is a function of system load.

> *A key insight of this chapter is that the higher the load, the more aggressively we want to serialize requests by lowering the large request threshold.*

The challenges for implementing such a policy include (a) identifying large requests dynamically, because the individual request work is unknown at arrival time and hard to predict; (b) determining how much and when requests are imposing queuing delay on other requests — thereby determining large requests as a function of current load; and (c) dynamically adjusting the policy.

Since we cannot determine the work of a request when it arrives, tail-control conservatively defaults to a steal-first policy that parallelizes all requests. We note that the large requests reveal themselves. Tail-control keeps track of the total work done so far by all active requests. Therefore only later, when a request reveals itself by executing for some threshold amount of work *and* when system load is high, do we force a request to execute sequentially. As mentioned earlier, the higher the load, the smaller the threshold at which we serialize requests. This approach also has the advantage that the higher the load, the earlier we serialize large requests, thereby reducing the amount of parallel processing time we spend on hopeless requests.

The next section presents our offline threshold-calculation that calculates the large request threshold for each value of system load based on the probabilities in a given work distribution. The higher the load, the lower the large request threshold calculated by this algorithm. It also shows how tail-control uses this information, load, and request progress to dynamically control parallelism.

8.3 Tail-Control Scheduler

This section describes the tail-control scheduler which seeks to reduce the number of requests that exceed a target latency. At runtime, the scheduler uses the number of active requests to estimate system load and continuously calculates each request's work progress. Tail-control serializes large requests to limit their impact on the waiting time of other requests. The scheduler identifies large requests based on when their work exceeds a **large request threshold**, which is a value that varies based on the instantaneous system load and the target latency. The higher the load, the lower the threshold and the more large requests tail-control serializes, such that more small requests make the target latency.

The tail-control scheduler has two components: (1) an offline threshold-calculation algorithm that computes a table containing large request thresholds indexed by the number of active requests, and (2) an online runtime that uses steal-first, but as needed, serializes large requests based on the large request threshold table.

8.3.1 The Threshold-Calculation Algorithm

We first explain how we compute the large request threshold table offline. The table is a set of tuples $\{q_t : 1 \text{ to } q_{max} \mid (q_t, l_{q_t})\}$ indicating the large request threshold l_{q_t} indexed by the number of active requests q_t. Figure 8.3 presents the pseudo code for calculating the table. For each q_t, the algorithm iterates through the set of candidate large request thresholds l and calculates the expected number of requests that will miss the target latency if the threshold is set as l. It sets l_{q_t} as the l that has the minimum expected requests whose latency will exceed the target.

```
1  for each number of active request $q_t$ from 1 to $q_{max}$
2      for each large request threshold candidate $l \in L$
3          calculate the length of pileup phase $T$
4          calculate the number of large request exceeding target $miss_l$
5          calculate the number of small request exceeding target $miss_s$
6      $l_{q_t}$ is the $l$ minimizing total misses $miss_l + miss_s$ for given $q_t$
7      Add $(q_t, l_{q_t})$ to large request threshold table
```

Figure 8.3: Algorithm for calculating large request threshold table

The algorithm takes as input: (a) a target latency, *target*; (b) requests per second, *rps*; (c) the number of cores in the server m; and (d) the work distribution, such as the examples shown in Figure 8.1. We derive the maximum number of active requests q_{max} from profiling or a heuristic such as twice the average number of active requests based on *rps* using queuing theory.

The work distribution is a probability distribution of the work per request, which service providers already compute to provision their servers. Here we use it for scheduling. We represent it as a set of non-overlapping bins: $\{\text{bin } i : 1 \text{ to } n \mid (p_i, w_i)\}$, where each bin has two values: the probability of a request falling into this bin p_i and the maximum work of the requests in this bin w_i. Note that the sum of the probability of all bins is one, i.e. $\sum_{\text{bin } i} p_i = 1$. For each bin in the work distribution, we only know the maximum work of requests falling in this bin. Therefore, we only need to consider these discrete values for large request thresholds l, since we pessimistically assume any request that exceeds w_i will definitely execute until w_{i+1}.Therefore, formally the set of large request threshold candidate set is $L = \{i : 1 \text{ to } n \mid w_i\}$.

We define an **instantaneous pileup** as a short period of time when the system experiences a high load. To reduce the number of requests that miss the tail latency constraint

Symbol	**Definition**
$target$	Target Latency
rps	Request per second of a server
m	Number of cores of a server machine
p_i	Probability of a request falling in bin i
w_i	Work of a request falling in bin i
q_t	Instantaneous number of active requests at time t
l	Large request threshold
L	Set of potential large request thresholds
$\bar{w}$	Average work per request
p_l	Probability of a request being a large request
$\bar{w_s}$	Expected work per small request
$\bar{w_f}$	Expected superfluous work per large request
$\bar{w_e}$	Expected essential work per request

Table 8.1: Notation Table in Chapter 8

during a pileup, our scheduler limits the processing capacity spent on large requests, to prevent them from blocking other requests.

The key of this algorithm is thus to calculate the expected number of requests that will miss the target latency during a pileup if the system load is q_t and the large request threshold is l, for every candidate $l \in L$ given the other parameters. We first calculate the expected work of requests, expected parallel work, and expected sequential work due to requests that exceed l. We then use these quantities to calculate the expected length of a pileup and the approximate number of requests whose latency will exceed the target. Table 8.1 defines the notations for this calculation.

Basic Work Calculation.

To start, consider the simple expected work calculations that depend only on the work distribution and a large request threshold candidate l. First, we calculate the expected work of a request $\bar{w} = \sum_{\text{bin } i} p_i w_i$. Next, we can calculate the probability that a particular request is large: $p_l = \sum_{w_i > l} p_i$; and the probability that a request is small: $p_s = 1 - p_l$. Additionally, the expected work per small request can be derived as $\bar{w_s} = (\sum_{w_i \le l} p_i w_i)/(1 - p_l)$.

Now consider the **expected essential work** per request $\bar{w_e}$ — the work that tail-control may execute in parallel. In tail-control, the entire work of small requests may execute in parallel. In addition, because a large request is only detected and serialized after being processed for l amount of work, hence every large request's initial l amount of work will execute in parallel given sufficient resources.

$$\bar{w_e} = \sum_{w_i \leq l} p_i w_i + \sum_{w_i > l} p_i l$$

We call this essential work since processing this work quickly allows us to meet target latency and detect large requests.

The remaining work of a large request exceeding l is serialized. We deem this work superfluous since these requests are very likely to miss the target latency and therefore this work will not contribute to the scheduling goal. We calculate the expected amount of work that is serialized per large request, formally denoted as the **expected superfluous work** per large request $\bar{w_f}$:

$$\bar{w_f} = \frac{\sum_{w_i > l} p_i (w_i - l)}{\sum_{w_i > l} p_i}$$

Pileup Phase Length.

Now we calculate the length of a pileup when the number of active requests is q_t and the large request threshold is l. We define the pileup start time as the time when a large request is first detected in the non-pileup state. The pileup ends when the large request that caused the pileup is completed, all the work that has accumulated in the server due to this overload is also completed, and the system reaches a steady state again.

We can approximately calculate the amount of accumulated work at the start of a pileup. First, let us look at the first large request that was detected. This request has the essential work of l and the remaining expected superfluous work of $\bar{w_f}$; it has a total of $\bar{w_f} + l$ work in expectation. In addition, each of the remaining $q_t - 1$ active requests has an expected work of $\bar{w}$. Thus, the total accumulated work at the start of a pileup phase is estimated by $\bar{w_f} + l + (q_t - 1)\bar{w}$.

We also need to account for the work that arrives during the pileup. We define the average utilization of the server as $U = \bar{w} \times rps$. We assume that $U < m$ since otherwise the latency of the requests will increase unboundedly as the system gets more and more loaded over time. In expectation, the new requests that arrive during the pileup have a utilization of U, which means that we need U cores to process them. Therefore, tail-control has $m - U$ remaining cores with which to execute the already accumulated work. Thus, to entirely finish the accumulated work, it will take about $\frac{(\bar{w_f}+l)+(q_t-1)\bar{w}}{m-U}$ time.

Now consider the first large request in a pileup: it executes in parallel for l amount of work and then executes sequentially for $\bar{w_f}$ time. Its minimum completion time would be $l/m + \bar{w_f}$, which assumes that it runs on all the cores before being detected and serialized. Thus, the expected length of the pileup phase T is the maximum of the time to entirely finish the accumulated work and the time to finish the first large request:

$$T = \max\left\{\frac{(\bar{w_f} + l) + (q_t - 1)\bar{w}}{m - U},\ l/m + \bar{w_f}\right\}$$

Large Request Target Misses.

Once we know the length of the pileup phase, we can trivially derive the number of requests that are expected to arrive during this window, which is $rps \times T$. Now we can calculate the expected number of large requests during the pileup phase. Since the probability of one request being large is p_l, we expect $p_l(rps \times T + q_t - 1) + 1$ large requests in total, including the first large request and the other potential large requests that become active or arrive in the interim. In the calculation, we pessimistically assume that large requests always exceed the target latency, as we will serialize them. This assumption enforces the serialization of requests to be conservative. Hence, the number of large requests exceeding the target latency

$miss_l$ is

$$miss_l = p_l(rps \times T + q_t - 1) + 1$$

Small Request Target Misses.

We now calculate the number of small requests, $miss_s$, that will miss the target latency, given a particular value of active requests q_t and large request threshold l. We optimistically assume that small requests always run with full parallelism. Therefore, if a small request starts executing as soon as it arrives, it will always meet the target latency. Thus, we first calculate how long a small request must wait to miss the target latency. Then we calculate x, which is how many requests must be ahead of this request in order to have this long of a wait. Finally, we calculate how many requests could have x requests ahead of them during a pileup phase.

We first calculate the average number of cores that are spent on executing the superfluous work of large requests. There are $miss_l$ large requests, each in expectation has $\bar{w_f}$ superfluous work. Therefore, the total amount of superfluous work in the pileup is $miss_l \times \bar{w_f}$. Hence, on average, $miss_l \times \bar{w_f}/T$ cores are wasted on working on the superfluous work since the pileup lasts for time T. Note that this quantity is very likely to be less than m assuming the system utilization $U < m$.[7] Thus, the remaining $m_s = m - (miss_l \times \bar{w_f}/T)$ cores can work on the essential work of both small and large requests. Since we now only consider the essential work, we can think of it as scheduling on a new system with m_s cores, the same rps as the original system, but the expected work per request is now $\bar{w_e}$.

For a small request with total expected work $\bar{w_s}$, its minimum execution time on the remaining m_s cores is $\bar{w_s}/m_s$. Given the target latency $target$, it will exceed the target if its waiting time is more than $target - \bar{w_s}/m_s$. Given x requests ahead of this request in

[7]If $miss_l \times \bar{w_f}/T$ is greater than m, this configuration of l would lead to system overrun, which means that this particular l is not a good candidate for l_{q_t}. Thus, we simply set $miss_s$ to be ∞.

addition to the first large request that triggered the pileup, then the essential work that needs to execute on these m_s cores before we get to the small request in consideration is $l + \bar{w_e}x$. Therefore, its waiting time can be estimated as $(l + \bar{w_e}x)/m_s$. A request misses the target latency if there are at least x requests ahead of it where $x = (target \times m_s - \bar{w_s} - l)/\bar{w_e}$.

Among the q_t requests that are active when we detect the large request, there are $y_1 = \max(q_t - 1 - x, 0)$ requests that have at least x requests in addition to the first large request that are ahead of it. These y_1 requests (where y_1 can be 0) will likely overrun the target latency. In addition, we must account for the additional requests that arrive while the queue is still longer than x. Assuming optimistically that requests run one by one in full parallelism, then requests leave at the rate of $1/(\bar{w_e}/m_s) = m_s/\bar{w_e}$. On the other hand, the system has a request arrival rate of rps. Hence the number of requests in the system decreases with a rate of $m_s/\bar{w_e} - rps$.[8] Thus, it takes $y_1/(m_s/\bar{w_e} - rps)$ amount of time for the number of active requests to decrease from q_t to $x + 1$. During this time, the number of newly arrived requests is then $y_2 = y_1/(m_s/\bar{w_e} - rps) \times rps$.

Therefore in total, we have $y_1 + y_2$ requests that will wait for more than x requests. Since a request is small with probability $(1 - p_l)$, the expected number of small requests that miss the target latency is

$$\begin{aligned} miss_s &= (y_1 + y_2)(1 - p_l) \\ &= \max(q_t - 1 - x, 0) \times \frac{m_s/\bar{w_e}}{(m_s/\bar{w_e} - rps)} \times (1 - p_l) \end{aligned}$$

Thus, we get the expected total number of requests exceeding the target latency for q_t number of active requests and l large request threshold is $miss = miss_s + miss_l$.

Discussion: The complexity of the offline algorithm is $O(q_{max} \times n^2)$, where n is the number of bins in the request work distribution. Note that we are conservative in serializing large

[8]Note that this rate is positive in order for the l in consideration to be a good candidate for l_{q_t}. Thus if the rate is not positive, we again set $miss_s$ to be ∞

requests. We estimate the execution time for a small request to be $\bar{w_s}/m_s$, which is minimum by assuming that it can occupy all available cores m_s and execute with linear speedup. However, some requests (especially small requests) may not have enough parallelism to use all m cores. The system is thus conservative in serializing large requests, because this calculation may overestimate the waiting time before the small requests miss the target latency. If the system profiler also profiles the average parallelism of requests in all bins (or the average span[9] of requests in all bins — these are equivalent), we can incorporate this information into the calculation easily to perform more accurate calculations, potentially leading to more aggressive parallelization of large requests.

Example Output.

Figures 8.4 shows an example output from threshold-calculation with input of the same work distribution and *rps* but three different target latencies. Let's first examine a single curve, say the one with 14.77ms target latency. As the number of active requests increases (i.e., higher instantaneous system load), the large request threshold decreases, indicating that the system serializes requests more aggressively. When examining all three curves collectively, Figure 8.4 illustrates how the relationship between the threshold and active number of requests varies according to the target latency. As the target latency increases, the curves shift towards the right, showing that when given the same number of active request, the system can increase the large request threshold because of the longer target latency. In other words, given the same instantaneous load, the system is less aggressive about serializing requests when the target latency is longer.

[9]The span of a request is the longest dependency chain in the request that must execute sequentially.

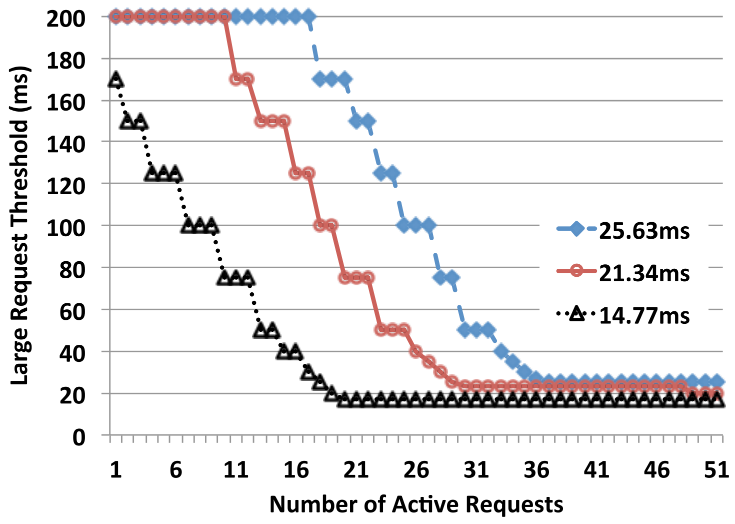


Figure 8.4: Example large request threshold tables output by threshold-calculation with input of the same work distribution and *rps*, but three different target latencies. The x-axis is the number of active requests, and the y-axis is the large request threshold. Each curve plots the output threshold table for a given latency.

8.3.2 Extending Work-Stealing with Tail-Control

This section describes how to use the large request table to implement the tail-control strategy in a work-stealing runtime. While we implemented our system in TBB [134], this approach is easy to add to other work-stealing runtimes.

Scheduler Overview.

As with the basic work-stealing scheduler, tail-control is a distributed scheduler, and each worker maintains its own work deque. A shared global FIFO queue contains requests that have arrived but not yet being admitted. Tail-control behaves like steal-first by default. Only when the system identifies a large request does its behavior diverge from steal-first with respect to the identified large request — it serializes the large request's remaining execution. Specifically, tail-control performs the following additional actions: 1) it tracks the number of active requests in the system to estimate system load; 2) it tracks processing time spent thus far on each executing request; 3) it identifies a large request based on processing time and serializes its remaining execution.

```
void scheduling_loop(Worker *w) {
  Task *task = NULL, *last_task = NULL;
  while (more_work_to_do()) {
    if (task != NULL) {
      do {
        execute(task);
        last_task = task;
        task = pop_deque(w);
      } while(task != NULL);
      if (last_task->parent() != NULL) {
        task = last_task->parent();
        if (task->ref_count() == 0) continue;
      } else if (last_task->is_root() &&
                 last_task->is_done()) {
          global_queue->dec_active_request_count();
        }
      }
      // about to switch request; update bookkeeping info
      long long proc_time = get_time() - w->start_time;
      w->last_request = w->curr_request;
      w->curr_request = NULL;
      w->last_request->accum_process_time(proc_time);
    } //end of if (task != NULL)
    check_pileup_phase();
    task = try_random_steal();
    if (task == NULL)
      task = global_queue->admit_request();
    if (task != NULL) {
      w->start_time = get_time();
      w->curr_request = task->get_request();
    }
  } // end of while (more_work_to_do())
} // end of scheudling_loop
```

Figure 8.5: The pseudo code for the main loop of tail-control in a work-stealing runtime system. Tail-control adds only the bold lines and the function `check_pileup_phase` **to steal-first.**

Tail-control performs actions 1) and 2) in a distributed fashion. Each worker accumulates processing time for its active request. Whenever a worker finishes a request completely, it decrements a shared global counter that maintains the number of executing requests in the system. Overhead for action 1) is minimal since it occurs infrequently. Overhead for action 2) is amortized against the cost of steals, since it needs to be done only between steals.

Action 3) requires more coordination. To identify a large request, it is necessary to collect processing time scattered across all the workers that are executing a request, which can incur overhead. Ideally, we would like to check each request's processing time frequently so that a request does not execute any longer once it exceeds the large request threshold. On the other hand, we do not want to burden executing workers with the overhead of frequent checks.

```
Task * try_random_steal() {
   Task *task = NULL;
   while (has_stealable_victims()) {
      Worker *vic = choose_random_victim();
      task = try_steal_deque_top(vic);
   }
   return task;
}

void check_pileup_phase() {
   int active = global_queue->get_active_request_count();
   int req_thresh = large_request_table[active];
   long long added_time, new_proc_time,
   long long curr_time = get_time();
   hashtable req_map;

   // update the processing time for each executing request
   for (Worker *w : worker_list) {
      Request *req = w->curr_request;
      if (req == NULL) continue;
      added_time = curr_time - w->start_time;
      // find returns 0 if req is not found
      if (req_map.find(req) == 0)
          new_proc_time = req->get_proc_time() + added_time;
      else
          new_proc_time = req_map.find(req) + added_time;
      req_map.insert(req, new_proc_time);
      // mark a request that exceeds threshold
      if (new_proc_time > req_thresh) {
         if (req.is_valid()) req->set_stealable(false);
      }
   }
}
```

Figure 8.6: The pseudo code for the helper routines of tail-control in a work-stealing runtime system. Tail-control adds only the bold lines and the function `check_pileup_phase` **to steal-first.**

Thus, we piggyback the execution of action 3) on a thief looking for work to do. Whenever a worker runs out of work, before it steals again, it first computes processing time of executing requests on all other workers, computes the accumulated processing time for each request, and marks any requests that exceed the threshold. Once a request is marked as large, it needs to be serialized. We modify the stealing protocol to mark large requests lazily. If a thief tries to steal from a deque with tasks that belong to a large request, the thief simply gives up and tries to steal somewhere else. Again, the overhead is amortized against the cost of steals.

Implementation of Tail-Control.

Figure 8.5 shows the pseudo code for the top-level scheduling loop that a worker executes in tail-control. The bold lines mark the code that is only necessary for tail-control but not steal-first. During execution, a worker always first tries to pop tasks off its local deque as long as there is more work in the deque (lines 5–8). When a worker's deque becomes empty, it tries to resume the parent of its last executed task if it is ready to be resumed (lines 9–12). In short, a worker always executes tasks corresponding to a single request for as long as work can be found locally (lines 5–12). As part of the bookkeeping in tail-control, if the last executed task has no parent (i.e., root), the worker checks for the completion of a request and decrements the active-request counter if appropriate (lines 13–16).

Once the control reaches line 18, a worker has exhausted all its tasks locally, and it is ready to find more work. Before performing a random steal, a worker performs the necessary bookkeeping to accumulate the processing time it spent on last request lines 18–21). It updates its `curr_request` field to reflect the fact it is not working on a request. This field keeps tracks of the worker's current request, and is read by a thief in `check_pileup_phase` when it accumulates processing time of an executing request.

Then the worker calls `check_pileup_phase` (line 23) to identify large requests and mark them as **not stealable**. It then becomes a thief and tries to steal randomly (line 24). If no currently executing requests are stealable, `try_random_steal` returns `NULL`, and tail-control admits a new request (line 26) and assigns it to the worker. Regardless of how the worker obtains new work, it updates its `curr_request` and the start time for this request (lines 27–30). When it completes this work, it loops back to the beginning of the scheduling loop to perform book keeping and find more work. In a server system, this scheduling loop executes continuously.

The tail-control implementation has the same high-level control flow as in steal-first, since except for the pileup phase, it follows the steal-first policy. Moreover, the difference between steal-first and admit-first is simply in the order in which a thief finds new work. By switching the sequence of stealing and admitting, i.e., switching line 24 and line 26, one trivially obtains admit-first.

Figure 8.6 shows the pseudo code for subroutines invoked by the main scheduling loop. Again, the differences between tail-control and steal-first are few and are highlighted in bold. Specifically, in tail-control, a thief gives up on the steal if the task on top of the victim's deque is a not-stealable large request. The `try_random_steal` (line 5) and the `has_stealable_victims` (line 3) implement this difference. This implementation causes the marked large request to serialize lazily; the parallelism dissolves when all workers working on that request exhaust their local tasks. We choose not to enforce the marked request to serialize immediately to avoid additional overhead.

The `check_pileup_phase` function implements the detection and marking of large requests. When a thief executes this function, it first evaluates the current load by getting the active-request count, and uses it to index the large request threshold table (lines 10–11). With the given threshold, it then examines all workers and accumulates the processing time of all executing requests into a local hashtable (lines 15–27). The hashtable uses requests as keys and stores their processing time as values. A hashtable is needed because there could be multiple workers working on the same request. The processing time of an executing request is essentially the time accumulated on the request thus far, and the additional processing time elapsed on an executing worker since the last time it updated its start time. Lines 16–23 does exactly that calculation.

Note the pseudo code simply shows the high-level control flow, abstracting many operations, including synchronization. Shared fields in a request object and the global queue are protected by locks. The locks are acquired only when concurrent writes are possible. One notable feature is that when a thief executes `check_pileup_phase`, it does not acquire locks when reading shared fields, such as `w->start_time`, `w->curr_request`, and `req->get_proc_time`. We intentionally avoid this locking overhead at the cost of some loss in accuracy in calculating the requests' processing time due to potentially stale values of these fields the thief may read. In this way, the processing time calculation happens entirely in a distributed manner and a thief will never interfere workers who are busy executing requests.

8.4 Experimental Evaluation

We now present an empirical evaluation of the tail-control strategy as implemented in the TBB work-stealing scheduler. We compare tail-control to steal-first and admit-first, and show that tail-control can improve over all baselines. Our primary performance metric is the number of requests that exceed the latency target.

Experimental Setup.

Experiments were conducted on a server with dual eight-core Intel Xeon 2.4Ghz processors with 64GB memory and Linux version 3.13.0. When running experiments, we disable processor throttling, processor sleeping, and hyper-threading. The Tail-control scheduler is implemented in the Intel Thread Building Block (TBB) library [134], version 4.3.

We evaluate our strategy on several different workloads that vary along three dimensions: (1) different work distributions (two real-world workloads and several synthetic workloads), (2) different inter-arrival time distributions (Poisson distribution with different means and long-tail distributions with varying means and standard deviations), and (3) different request parallelism degrees (embarrassingly parallel requests and requests with parallelism less than the number of cores).

First, we evaluate requests with different work distributions. The two real-world work distributions are shown in Figure 8.1. Henceforth, we shall refer to them as the **Bing workload** and the **finance workload**, respectively. In addition, we also evaluate synthetic workload with log-normal distributions, referred as **log-normal workload**. A log-normal distribution generates random variables whose logarithm is normally distributed. Thus, it has a longer tail than a normal distribution and represents the characteristics of many real-world workloads. For all the workloads, we use a simple program to generate work — the program performs a financial calculation which estimates the price of European-style options with Black-Scholes Partial Differential Equation. Each request is parallelized using parallel-for loops. Note that the particular computation performed by the workload is not important for the evaluation of our strategy; any computation that provides the same workload distribution should provide similar results.

Second, we evaluate requests with different arrival distributions. In particular, to randomly generate the inter-arrival time between requests, we use two distributions: a Poisson distribution with a mean that is equal to $1/rps$, and a log-normal distributions with a mean equal to $1/rps$ and varying standard deviations. In other words, for all the distributions, the requests are generated in an open loop at the rate of rps (queries per second). We use $100,000$ requests to obtain single point in the experiments.

In addition, we evaluate the effect of requests with different parallelism degrees. In addition to the embarrassingly parallel requests generated by parallel-for loops of Black-Scholes, we also intentionally insert sequential segments to make requests less parallel. For each request, we add sequential segments with total length of 10% of its work. By doing so, the parallelism of requests is less than 10, which is smaller than the 16 available cores.

Finally, we explore some additional properties of tail-control. We test its performance when the input work distribution is inaccurate and differs from the actual work distribution. We also present the improvement of tail-control in terms of system capacity. We conduct comparison with two additional algorithms to position the performance of steal-first, admit-first, and tail-control. Lastly, we present a result trace to unveil the inner workings of tail-control.

8.4.1 Different Work Distributions

We first compare the performance of tail-control to admit-first and steal-first, the two baseline work-stealing policies described in Section 8.2, with various loads and target latencies. For all the experiments in this subsection, requests arrive according to Poisson distribution with varying mean inter-arrival times. Figures 8.7, 8.8 and 8.9 show the results on work distribution for Bing,finance, and log-normal work distributions respectively, where each bar graph in a figure shows the result for one load setting (light, medium or heavy when going from left to right). Within each graph, we compare the policies for five latency targets. As we go from left to right, the target latency increases,[10] and thus all policies improve in performance from left to right. Now we can look at the specific results for each workload.

Bing Workload.

From Figure 8.7, we can make three observations. First, for the Bing workload, admit-first performs better than steal-first in most cases. The reason is, as seen in Figure 8.1(a), the Bing workload has high variability between the work of the largest vs. the smallest requests. As discussed in Section 8.2, steal-first is likely to perform worse in this situation since it allows very large requests to monopolize the processing resources, potentially delaying a high number of small and medium requests. Second, as target latency increases, steal-first's performance in comparison to admit-first improves, finally overtaking it slightly for the longest latency target. As the target latency increases, the impact on waiting time due to executing large requests reduces and steal-first starts reaping the benefits of exploiting intra-request parallelism; therefore, it starts performing better than admit-first. This observation reflects the trade-off between steal-first and admit-first and confirms that they cannot perform well in all settings. Finally, tail-control provides consistently fewer missed requests across the three load settings and target latencies — it has a relative improvement of 35% to 58% over

[10]The target latencies are chosen as the 97.5%, 98.5%, 99%, 99.5% and 99.75% tail latencies of steal-first in order to provide evidence that tail-control performs well under varying conditions.

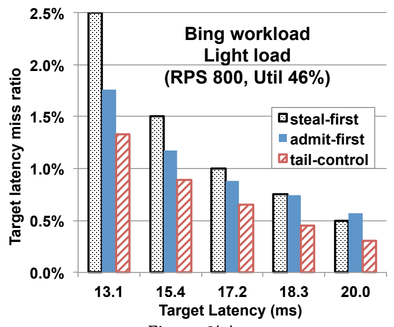


Figure 6(a)

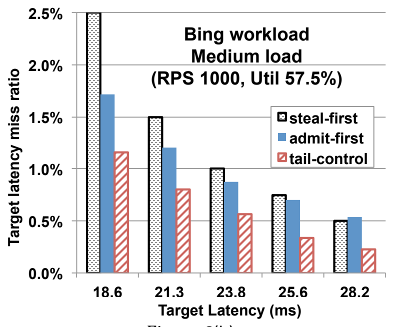


Figure 6(b)

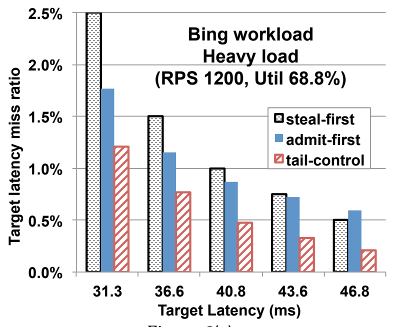


Figure 6(c)

Target (ms)	13.1	15.4	17.2	18.3	20.0
Imp. over SF	47%	41%	35%	40%	39%
Imp. over AF	24%	25%	26%	39%	47%

Table 6(a)

Target (ms)	18.6	21.3	23.8	25.6	28.2
Imp. over SF	54%	46%	43%	55%	54%
Imp. over AF	33%	33%	35%	52%	57%

Table 6(b)

Target (ms)	31.3	36.6	40.8	43.6	46.8
Imp. over SF	52%	49%	52%	56%	58%
Imp. over AF	31%	34%	45%	54%	65%

Table 6(c)

Figure 8.7: Results for the Bing workload with three different load settings and Poisson arrival. The x-axis shows different target latencies from shorter to longer from left to right. The y-axis shows the target latency miss ratio. The table below each figure shows tail-control's relative improvement over steal-first and admit-first for a given latency.

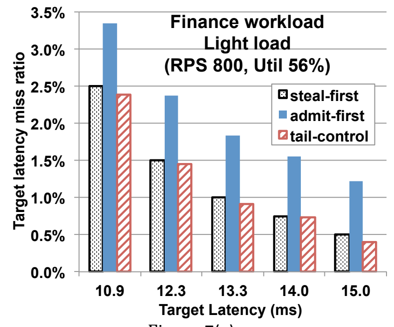


Figure 7(a)

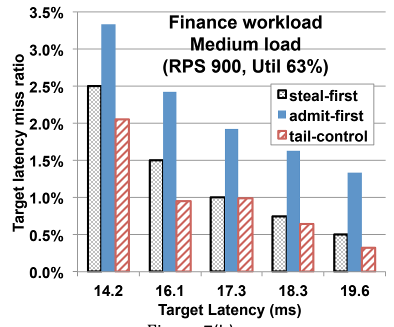


Figure 7(b)

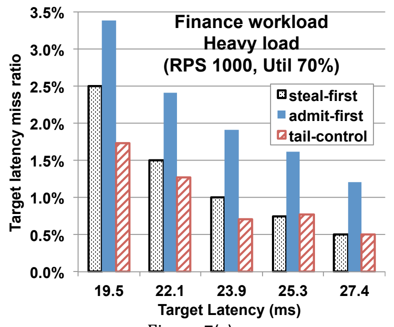


Figure 7(c)

Target (ms)	10.9	12.3	13.3	14.0	15.0
Imp. over SF	5%	3%	9%	1%	19%
Imp. over AF	29%	39%	51%	53%	67%

Table 7(a)

Target (ms)	14.2	16.1	17.3	18.3	19.6
Imp. over SF	18%	36%	2%	14%	36%
Imp. over AF	39%	61%	49%	60%	76%

Table 7(b)

Target (ms)	19.5	22.1	23.9	25.3	27.4
Imp. over SF	31%	15%	29%	-3%	-1%
Imp. over AF	49%	47%	63%	52%	58%

Table 7(c)

Figure 8.8: The finance workload results with the same figure and table configuration as in Figure 8.7.

steal-first and of 24% to 65% over admit-first in all settings. In particular, tail-control has higher improvement at the harsher setting — when the system load is heavier and the target latency is shorter. It limits the impact of large requests by serializing the large requests. However, it still reaps the benefit of intra-request parallelism since it parallelizes short and medium requests and processes them quickly.

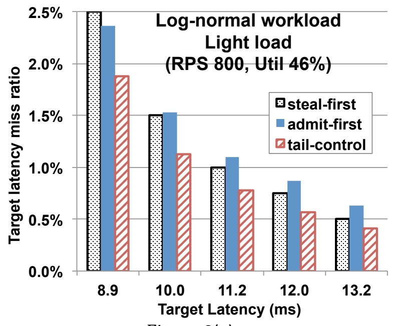


Figure 8(a)

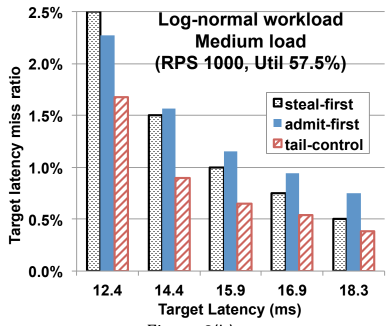


Figure 8(b)

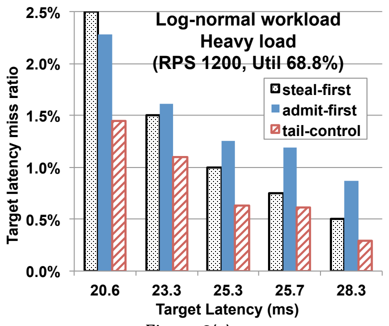


Figure 8(c)

Target (ms)	8.9	10.0	11.2	12.0	13.2
Imp. over SF	25%	25%	23%	24%	17%
Imp. over AF	20%	26%	29%	35%	34%

Table 8(a)

Target (ms)	12.4	14.4	15.9	16.9	18.3
Imp. over SF	33%	40%	35%	28%	22%
Imp. over AF	26%	43%	44%	43%	49%

Table 8(b)

Target (ms)	20.6	23.3	25.3	25.7	28.3
Imp. over SF	42%	27%	37%	18%	41%
Imp. over AF	37%	32%	50%	49%	66%

Table 8(c)

Figure 8.9: The log-normal workload results with the same figure and table configuration as in Figure 8.7.

Finance Workload.

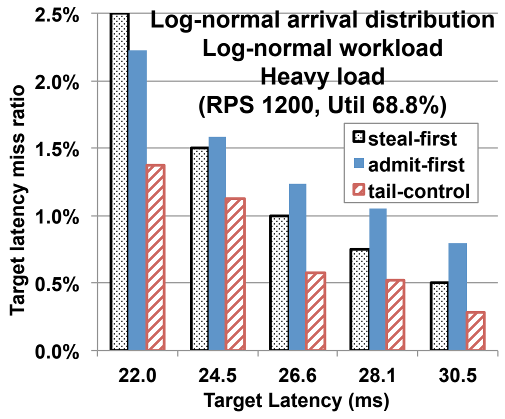
Figure 8.8 shows the results for the finance workload, and the results reinforce the observations made above. The finance workload has less variability in its work distribution compared to the Bing workload; therefore, as our intuition indicates, steal-first performs better than admit-first. Since large requests are not that much larger than other requests, they do not impact the other requests as significantly. Therefore, steal-first is a good strategy for this workload. In the settings under the heavy load and the two longest target latencies, tail-control provides exactly the same schedule as steal-first, since the calculated thresholds from threshold-calculation are high resulting in no request being serialized. This result indicates that when steal-first is the best strategy, tail-control essentially defaults to it and does not gratuitously increase target latency misses by serializing requests unnecessarily. In addition, even though tail-control has a higher overhead due to bookkeeping, the overhead is lightweight and not significant enough to affect performance. Moreover, even on this workload where steal-first is already a good strategy, tail-control still significantly improves the miss ratio in some cases. For instance, under medium load with the longest target latency, tail-control provides a 36% improvement. These results indicate that tail-control performs well under different types of workload characteristics.

Log-normal Workload.

In Figure 8.9, we generate the work of a request using a log-normal distribution with mean of 10ms and standard deviation of 13ms. We selectively choose the mean of 10ms in order to directly compare the results with that of Bing workload, as the mean work of Bing workload is also 10ms. The standard deviation is chosen, such that the log-normal work distribution has a slightly shorter tail than the Bing workload, but a longer tail than finance workload. For a log-normal workload, steal-first performs better than admit-first when target latency is long and slightly worse when the target latency is short. In this setting, tail-control consistently outperforms steal-first and admit-first with improvement from 17% up to 66%.

8.4.2 Different Arrival Distributions

Note that in all the previous experiments, requests arrive according to a Poisson arrival process. In these experiments, we vary the inter-arrival time distribution and select the log-normal distribution, which has larger variance than the Poisson distribution. Figure 8.10 shows experimental results for a log-normal workload and a log-normal arrival distribution with mean inter-arrival time of 0.83ms (*rps* of 1200) and standard deviation of 1.09ms. Note that this experiment is constructed such that it has almost the same parameters as that of Figure 8(c), except that the latter's inter-arrival time has a smaller standard deviation of 0.91ms. By comparing the two experiments, we can see that the relative trend remains the same. In particular, steal-first is better than admit-first, while tail-control outperforms the best of them by 25% to 44%. The comparison between Poisson and log-normal arrival distribution with Bing workload are similar too. The results indicate that request arrival distribution does not impact the relative performance much.

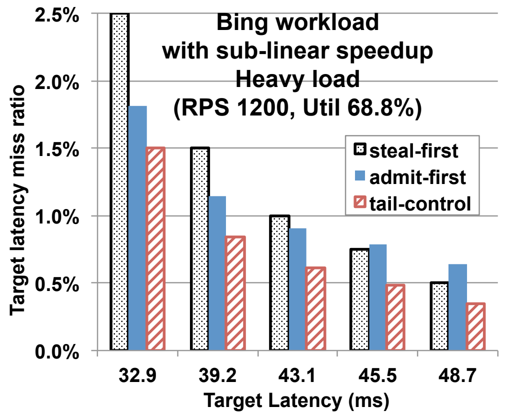


Target (ms)	22.0	24.5	26.6	28.1	30.5
Imp. over SF	45%	25%	42%	31%	44%
Imp. over AF	38%	29%	53%	51%	65%

Figure 8.10: Results for the log-normal workload and a log-normal arrival distribution with 1200 *rps*; Figure and table configurations are similar as in Figure 8.9.

8.4.3 Request with Sub-Linear Speedup

For the rest of the section, we focus on the Bing and log-normal workloads at the heavy load of 1200 rps. In all the previous experiments, requests are embarrassingly parallel with near linear speedup. Here we evaluate how well tail-control performs, when increasing the span and decreasing the parallelism degree of requests. In particular, in Figure 8.11 we intentionally add sequential segments with a total length of 10% work into each request, resulting a parallelism degree of less than 10 and smaller than the total 16 cores. As discussed in Section 8.3.1, we incorporate the span into the tail-control threshold calculation. Note that this experiment has almost the same parameters as Figure 6(c), except for a smaller parallelism degree. By comparing the two, we can see that the relative trend among different algorithms remains the same. However, tail-control has less improvement over steal-first and admit-first from 17% to 46%. The results for log-normal workload are similar to that of Bing workload. Tail-control improves less in this setting, because small requests are not able to utilize all the available cores, even when large requests are serialized. Moreover, the large request does not monopolize the entire system due to its long sequential segments. Note that in the extreme case where all requests are sequential, all the three algorithms will be



Target (ms)	32.9	39.2	43.1	45.5	48.7
Imp. over SF	40%	44%	39%	35%	31%
Imp. over AF	17%	27%	33%	38%	46%

Figure 8.11: Results for the Bing workload and a Poisson arrival distribution with 1200 *rps* for requests with sub-linear speedup; Figure and table configurations are similar as in Figure 8.7.

the same. The improvement that tail-control provides depends on the parallelism of requests and the target latency. As the degree of request parallelism increases, tail-control provides more benefit.

8.4.4 Inaccurate Input Work Distribution

Note that tail-control calculates large request threshold using request work distribution as input. Hence, it is natural to ask how tail-control performs when the work distribution differs from the input work distribution for the threshold-calculation algorithm. We experimentally evaluate how brittle tail-control is when provided with a somewhat inaccurate input work distribution. The experiment in Figure 8.12 has the same setting as Figure 8(c) with target latency of 28ms. In addition to the tail-control with the correct input, we also run tail-control using inaccurate input. In particular, we slightly alter the input work distribution by changing the standard deviation while keeping the same mean work. As the input distribution is inaccurate, the calculated thresholds are also inaccurate.

From Figure 8.12, we can see that when the input inaccuracies are small, for example standard deviation of 10ms and 17ms instead of the true 13ms, tail-control still has comparable performance. However, the improvement of tail-control decreases when the error increases. Moreover, tail-control is less sensitive to a larger inaccurate standard deviation than a smaller one. When the standard deviation of the profiling workload is small, tail-control less aggressively serializes requests and it performs similarly to steal-first. In contrast, when the standard deviation of the profiling workload is large, tail-control is slightly more aggressive than it should be. In this case, it unnecessarily serializes only a few large requests. Since the probability of large requests is small, it affects the performance of tail-control only slightly. However, when the input is significantly different from the actual work distribution (for example, when the mean work is significantly inaccurate), tail-control could have worse performance than steal-first and admit-first. This case causes tail-control to very aggressively serialize requests, even when the number of requests is less than the number of cores.

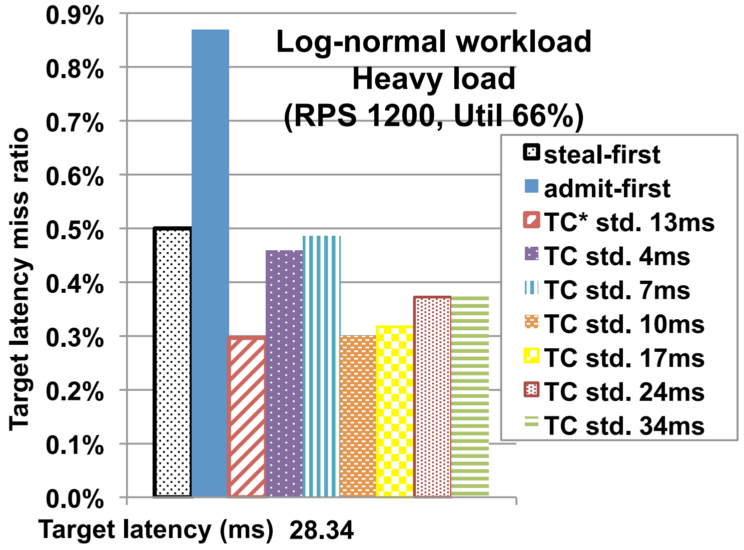


Figure 8.12: Results for the log-normal workload a and Poisson arrival distribution with 1200 *rps* and a target latency of 28ms. We compare tail-control when using inaccurate input distributions with smaller to larger standard deviation from left to right.

In summary, tail-control does require a relatively accurate work distribution, but it does not need to be exact. If the system load changes significantly over time, the system operator

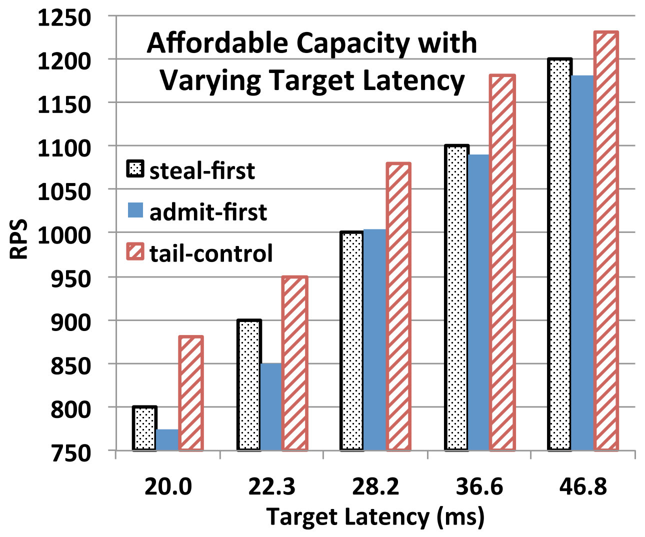


Figure 8.13: The Bing workload results. Tail-control increases system capacity for different target latencies compared to steal-first and admit-first.

or an online profiling mechanism should profile the new work distribution and *rps*. Although we presented the threshold calculation (Section 8.3.1) as an offline algorithm, it is fast; thus, an interactive service could sample its workload and recalculate the large request thresholds on-line and then adapt the tail-control scheduler in real-time.

8.4.5 Increased System Capacity

The benefits of tail-control can be used to increase server capacity, thereby reducing the number of servers needed to run a particular aggregate workload as each server can run a higher number of requests per second. For instance, say we have m servers and for a particular workload, steal-first enables 99.5% of the requests to meet the target latency. Here, tail-control can provide the same guarantee (99.5% requests meet the same latency target) with a higher system load. Figure 8.13 provides evidence for this claim. It shows the maximum load for several target latencies at the 99.5-percentile. For instance, at target latency 20.00ms, tail-control sustains 880 *rps* compared to 800 for steal-first and 775 for admit-first, showing 10% capacity increase over the best of the two.

8.4.6 Comparison with Additional Algorithms

Now we provide a comparison with two additional algorithms. The first one is denoted as **default-TBB**, as it is the algorithm of the default TBB implementation. Note that in both steal-first and admit-first, requests are first submitted to a global FIFO queue and workers explicitly decide whether and when to admit requests. In contrast, default-TBB implicitly admit requests by submitting requests to the end of a random worker's deque. After doing so, workers can act as if there is a single request and they only need to randomly steal when running out of work. This strategy may have smaller overhead than the global FIFO queue. However, now requests are admitted randomly instead of in the FIFO order, so it may cause a waiting request to starve in the worst case.

We measure the performance of default-TBB for all three workloads with Poisson arrival and observe that default-TBB has comparable or worse performance than admit-first in most settings. Default-TBB acts similarly to admit-first, but it admits requests in a random order. A request that has waited for a very long time could potentially be admitted later than a request that just arrived, causing significantly increases in the latency of some requests when the system is busy and consequently increasing the request target miss ratio. On the other hand, without any global queue default-TBB has smaller overheads. In this case, the system processes requests slightly faster and hence reduces the length of pileup phase. Thus, default-TBB performs slightly better than admit-first in these cases. However, tail-control still outperforms default-TBB by at least 31% in Figure 8.14 and 32% in Figure 8.15, respectively.

The second baseline algorithm marked as **TC-Clairvoyant** in Figure 8.15 and Figure 8.14 shows the lower bound of tail-control, because this scheduler utilizes the exact work of each request to perform tail-control strategy. Specifically, we modified TBB and the application server to mark each request with its actual work when it is submitted to

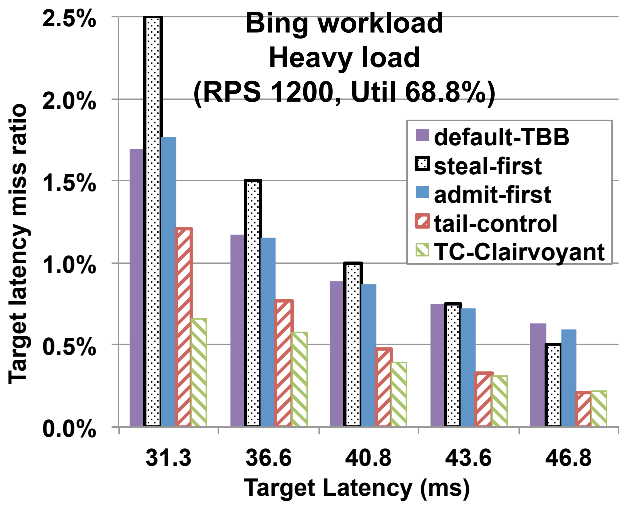


Figure 8.14: The Bing workload results. The figure is the same as Figure 8.7(c), except it adds default-TBB and TC-Clairvoyant.

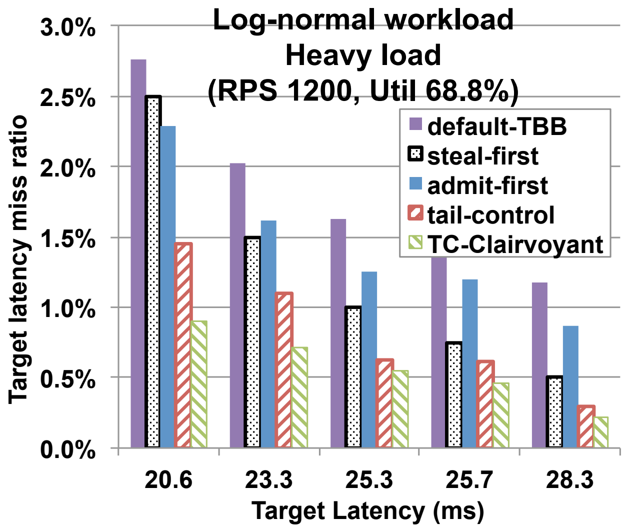


Figure 8.15: The log-normal workload results. The figure is the same as Figure 8.9(c), except it adds default-TBB and TC-Clairvoyant.

the global FIFO queue. For threshold calculation, we adjust the threshold calculation algorithm described in Section 4 for TC-Clairvoyant to account for the fact that a serialized large request under TC-Clairvoyant is never executed on more than one processor. During online execution, the TC-Clairvoyant scheduler knows the exact work of each request even before its execution (thus clairvoyant). Hence, unlike tail-control, TC-Clairvoyant knows whether a request is a large request and can directly serialize it without any parallel execution. Thus, TC-Clairvoyant serves as a lower bound of non-clairvoyant tail-control, as it can more effectively limit the impact of large request if the actual work of each request is known ahead-of-time to the scheduler. Of course, this this knowledge is not typically available.

From Figure 8.15 and Figure 8.14, we can see that TC-Clairvoyant and tail-control have comparable performance when the target latency is long, because to minimize for long target latency, only the very large requests need to be serialized. When the work distribution has a relatively long tail, we can more easily distinguish large requests from other medium or small requests. TC-Clairvoyant improves much more when the target latency is short because it on occasion will serialize some medium requests as soon as they begin executing. In the case of tail-control, medium requests will execute for most of their work before tail-control can distinguish them from small requests. The TC-Clairvoyant results shows that our tail-control strategy would improve performance even more if there was an accurate and efficient mechanism to predict work.

8.4.7 The Inner Workings of Tail-Control

We now look a little deeper into the inner workings of tail-control. Figure 8.16 shows the trace of the execution of the same workload under both steal-first and tail-control. The x-axis is time t as the trace executes. The lower y-axis is the queue length at time t. Recall that tail-control works in the same way as steal-first, except that under highly loaded conditions, it serializes large requests so as to reduce the waiting time of small requests in the queue. This figure shows that tail-control succeeds in this stated goal — when the instantaneously system load is high (for instance, at time 0.2s and 0.6s), the queue length is shorter under tail-control than under steal-first, but otherwise they follow each other closely.

The top part of the figure is even more interesting. For the same experiment, the upper y-axis shows the latency of each request released at time t along the x-axis. We can see

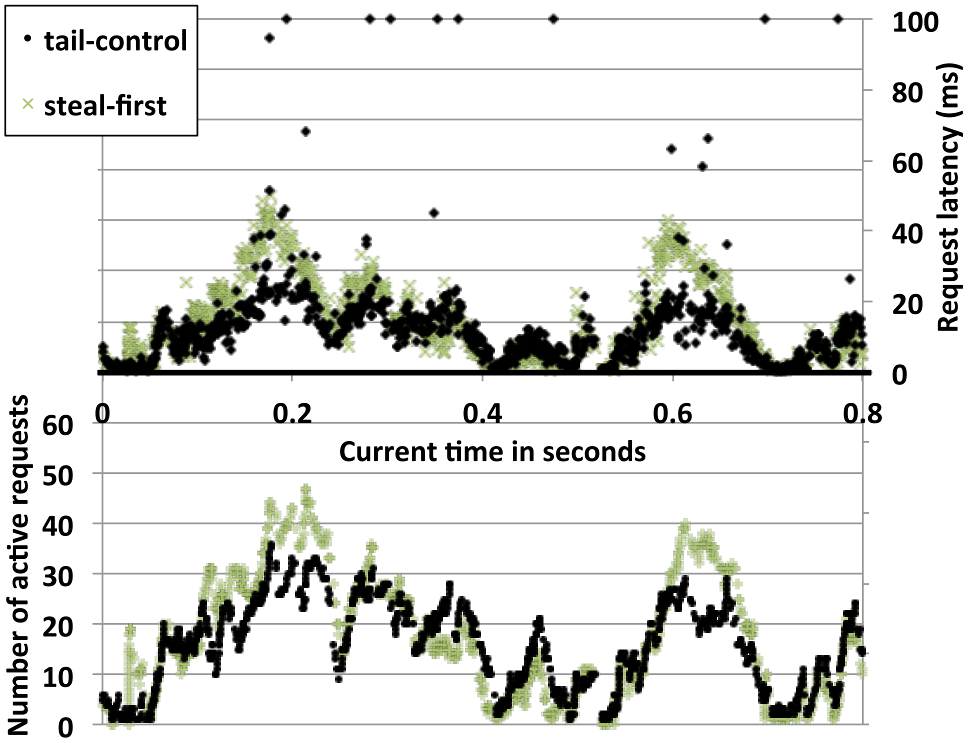


Figure 8.16: Number of active requests (lower part) and request latency (upper part) traces of the same workload under steal-first and tail-control in a 0.8 **second window (Bing workload).**

clearly that tail-control sacrifices a few requests (e.g., the ones with over 60ms latency), increasing their latency by a large amount in order to reduce the latency of the remaining requests. For the high instantaneous load at time $0.2s$ for instance, tail-control has a few requests that have very large latencies, but the remaining requests have smaller latencies than steal first. In particular, under steal-first, 183 requests have latencies that exceed the target latency of 25ms, while only 42 requests exceed the target under tail-control.

Chapter 9

Scheduling Parallel Jobs Online to Maximize Profit

Scheduling preemptive jobs online to meet deadlines is a fundamental problem and, consequently, this area has been extensively studied. In a typical setting, there are n jobs that arrive over time. Each job J_i arrives at time r_i, has a deadline d_i, relative deadline $D_i = d_i - r_i$ and a profit or weight p_i that is obtained if the job is completed by its deadline. The **throughput** of a schedule is the total profit of the jobs completed by their deadlines, and a popular scheduling objective is to maximize the throughput of the schedule.

In a generalization of the throughput problem, each job J_i is associated with a function $p_i(t)$, which specifies the profit obtained for finishing job J_i at time $r_i + t$. It is assumed that p_i can be different for each job J_i and the functions are arbitrary non-increasing functions. The goal is for the scheduler to maximize $\sum_{i \in [n]} p_i(t_i)$. We call this problem the **general profit** problem.

In this work, we consider the throughput and general profit scheduling problems in the preemptive online setting for parallel jobs. In this setting, the *online* scheduler is only aware of the job at the time it arrives in the system, and a job is *preemptive* if it can be started, stopped, and resumed from the previous position later. We model each parallel job as a **directed acyclic graph (DAG)**. Each node in the DAG is a sequence of instructions to be executed; the edges in the DAG represent dependencies. A node can be executed if and only if all of its predecessors have been completed. Therefore, two nodes can potentially run in parallel if neither precedes the other in the DAG. In this setting, each job J_i arrives as a single independent DAG and a profit of p_i is obtained if *all* nodes of the DAG are completed by job J_i's deadline. The DAG model can represent parallel programs written in many widely used parallel languages and libraries, such as OpenMP [125], Cilk Plus [89], Intel TBB [134] and Microsoft Parallel Programming Library [41].

Both the throughput and general profit scheduling problem have been studied extensively for sequential jobs. In the simplest setting, each job J_i has work or processing time W_i to be processed on a single machine. It is known that there exists a deterministic algorithm which is $O(\delta)$-competitive, where δ is the ratio of the maximum to minimum density of a job [24, 25, 101, 153]. The *density* of job J_i is $\frac{p_i}{W_i}$ (the ratio of its profit to its work). In addition, this is the best possible result for any deterministic online algorithm even in the case where all jobs have unit profit and the goal is to complete as many jobs as possible by their deadlines. If the algorithm can be randomized, $\Theta(\min\{\log \delta, \log \Delta\})$ is the optimal competitive ratio [92, 100]. Δ is the ratio of the maximum to minimum job processing time.

These strong lower bounds on the competitive ratio of any online algorithm makes it difficult to differentiate between algorithms and to discover the key algorithmic ideas that work well in practice. To overcome this challenge, the now standard form of analysis in scheduling theory is a *resource augmentation* analysis [93, 141]. In a resource augmentation analysis, the algorithm is given extra resources over the adversary and the competitive ratio is bounded. A s-speed c-competitive algorithm is given a processor that is s times faster than the optimal solution and achieves a competitive ratio of c. The seminal scheduling paper [93] considered the throughput scheduling problem and gave the best possible $(1+\epsilon)$-speed $O(\frac{1}{\epsilon})$-competitive algorithm for any fixed $\epsilon > 0$.

Since this work, there has been an effort to understand and develop algorithms for more general scheduling environments and objectives. In the identical machine setting where the jobs can be scheduled on m identical parallel machines, a $(1+\epsilon)$-speed $O(1)$-competitive algorithm is known for fixed $\epsilon > 0$ [11]. This has been extended to the case where the machines have speed scalable processors and the scheduler is energy aware [132]. In the related machines and unrelated machines settings, similar results have been obtained as well [87]. In [119] similar results were obtained in a distributed model. None of the prior work considers parallel jobs.

Results: We give the *first* non-trivial results for scheduling parallelizable DAG jobs online to maximize throughput and then we generalize these results to the general profit problem. Two important parameters in the DAG setting are the **critical-path length** L_i of job J_i (its execution time on an infinite number of processors) and its total **work** W_i (its uninterrupted execution time on a single processor). The value of $\max\{L_i, W_i/m\}$ is a lower bound on the amount of time any 1-speed scheduler takes to complete job J_i on m cores. We will focus on schedulers that are aware of the values of L_i and W_i when the job arrives, but are unaware of the internal structure of the job's DAG. That is, besides L_i and W_i, the only other information a scheduler has on a job's DAG is which nodes are currently available to execute. We call such an algorithm *semi-non-clairvoyant* — for DAG jobs. This is a reasonable model for the real world programs since the DAG generally unfolds dynamically as the program executes. We first state a simple theorem about these schedulers.

Theorem 49 *There exists inputs where any semi-non-clairvoyant scheduler requires a speed augmentation of* $2 - 1/m$ *to be* $O(1)$*-competitive for maximizing throughput.*

Scheduling even a single DAG job in time smaller than $\frac{W_i - L_i}{m} + L_i$ is a hard problem even in the offline setting where the entire job structure is known in advance. This is captured by the classic problem of scheduling a precedence constrained jobs to minimize the makespan. For this problem, there is no $2 - \epsilon$ approximation assuming a variant of the unique games conjecture [146]. We can construct a DAG where any semi-non-clairvoyant scheduler will take roughly $\frac{W_i - L_i}{m} + L_i$ time to complete, while a fully clairvoyant scheduler can finish in time W_i/m. By setting the relative deadline to be $D_i = W_i/m = L_i$, every semi-clairvoyant scheduler will require a speed augmentation of $2 - 1/m$ to have bounded competitiveness.

With the previous theorem in place, we cannot hope for a $(1+\epsilon)$-speed $O(1)$-competitive algorithm. To circumvent this hurdle, one could hope to show $O(1)$-competitiveness by either using more resource augmentation or by making an assumption on the input. Intuitively, the hardness comes from having a relative deadline D_i close to $\max\{L_i, W_i/m\}$. In practice, this is unlikely to be the case. We show that so long as $D_i \geq (1+\epsilon)(\frac{W_i - L_i}{m} + L_i)$ then there is a $O(\frac{1}{\epsilon^6})$-competitive algorithm.

Theorem 50 *If $(1+\epsilon)(\frac{W_i - L_i}{m} + L_i) \leq D_i$ for every job J_i, there is a $O(\frac{1}{\epsilon^6})$-competitive algorithm for maximizing throughput.*

We note that this immediately implies the following corollaries. One with no assumptions on the input and one for "reasonable jobs."

Corollary 51 *There is a $(2+\epsilon)$-speed $O(\frac{1}{\epsilon^6})$-competitive algorithm for maximizing throughput.*

Proof. No schedule can finish a job J_i if its relative deadline is smaller than $\max\{L_i, \frac{W_i}{m}\}$ and we may assume that no such job exists. Using this, we have that $(\frac{W_i}{m} + L_i) \leq 2D_i$. Consider transforming the problem instance giving the algorithm *and* the optimal solution together $2+\epsilon$ speed. In this case, the condition of Theorem 50 is met since we can view this as scaling the work in each node of the jobs by $2+\epsilon$. This scales the work and critical-path length by $2+\epsilon$. The corollary follows by observing that in this case we are comparing to an optimal solution with $2+\epsilon$ speed which is only stronger than comparing to an optimal solution with 1 speed. □

We note that the theorem also immediately implies the following corollary for "reasonable jobs."

Corollary 52 *There is a $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^6})$-competitive for maximizing throughput if $(W_i - L_i)/m + L_i \leq D_i$ for all jobs J_i.*

The assumption on the job deadline is reasonable as there exists inputs for which even the optimal semi-non-clairvoyant scheduler has unbounded performance if the deadline is any smaller.

For the general profit scheduling problem, we can make the following assumption. For all jobs J_i its general profit function satisfies $p_i(d) = p_i(x_i^*)$, where $0 < d \leq x_i^*$ for some $x_i^* \geq (1+\epsilon)(\frac{W_i - L_i}{m} + L_i)$. This assumption states that there is no additional benefit for completing a job J_i before time x_i^*; this is the natural generalization of the assumption in the throughput case. Using this, we show the following.

Theorem 53 *If for every job J_i it is the case that $p_i(d) = p_i(x_i^*)$, where $0 < d \leq x_i^*$ for some value of $x_i^* \geq (1+\epsilon)(\frac{W_i - L_i}{m} + L_i)$, there is a $O(\frac{1}{\epsilon^6})$-competitive algorithm for the general profit objective.*

Corollary 54 *There is a $(2+\epsilon)$-speed $O(\frac{1}{\epsilon^6})$-competitive algorithm for maximizing general profit.*

9.1 Preliminaries

In the problem considered, there is a set $\mathcal{J}$ of n jobs $\{J_1, J_2, ..., J_n\}$ which arrive online. The jobs are to be scheduled on m identical processors. Job J_i arrives at time r_i. Let $p_i(t)$ be an arbitrary non-negative non-increasing function for job J_i. The value of $p_i(t)$ is the profit obtained by completing job i at time $r_i + t$. Under some schedule, let t_i be the time it takes to complete job i after its arrival (that is, the job completes at time $r_i + t_i$). The goal is for the scheduler to maximize $\sum_{i \in [n]} p_i(t_i)$.

A special case of this problem is scheduling jobs with deadlines. In this problem, each job J_i has a deadline d_i and obtains a profit of p_i if it is completed by this time. In this case, we let $D_i = d_i - r_i$ be the relative deadline of the job. To make the underlying ideas of our approach clear, we will first focus on proving this case and the more general problem can be found in the Section 9.3.

Each individual job is represented by a Directed-Acyclic-Graph (DAG). A node in the DAG is *ready* to execute if all its predecessors have completed. A job is *completed* only when *all* nodes in the job's DAG have been processed. We assume the scheduler knows the ready nodes for a job at any point in time, but does not know the entire DAG structure a priori. Any set of ready nodes can be processed at once, but each processor can only execute one node at a time.

A DAG job has two important parameters. The total *work* W_i is the sum of the processing time of the nodes in job i's DAG. The *span* or *critical-path-length* L_i is the length of the longest path in job i's DAG, where the length of the path is the sum of the processing time of nodes on the path. The notations used throughout this chapter is summarized in Tables 9.1, 9.2 and 9.3.

Notation	**Definition**
OPT	optimal schedule and also optimal objective
m	the number of processors
W_i	the total work of job J_i
L_i	the span of job J_i
D_i	relative deadline of job J_i
r_i	the arrival time of J_i
d_i	the absolute deadline of J_i (that is, $r_i + D_i$)
$A(T, v_1, v_2)$	all jobs in T with density within the range $[v_1, v_2)$
$N(T, v_1, v_2)$	$= \sum_{J_i \in A(T,v_1,v_2)} n_i$, the total number of processors required by $A(T, v_1, v_2)$
v-dense	if Job J_i has density $v_i \geq v$
δ	$< \epsilon/2$
c	$\geq 1 + \frac{1}{\epsilon\delta}$
b	$= (\frac{1+2\delta}{1+\epsilon})^{1/2} < 1$
a	$= 1 + \frac{1+2\delta}{\epsilon - 2\delta}$

Table 9.1: Notations and definitions throughout Chapter 9

9.2 Maximizing Profit of Jobs with Deadlines

In this section, we give an algorithm and analysis proving Theorem 50 when jobs have deadlines and profits. Throughout the proof, we let C^O denote the jobs that the optimal solution completes by their deadline and let $\|C^O\|$ denote the total profit obtained by the

Notation	Definition
p_i	the profit of job J_i
n_i	$= \frac{(W_i - L_i)}{\frac{D_i}{1+2\delta} - L_i}$, the number of processors allocated to J_i
x_i	$= \frac{W_i - L_i}{n_i} + L_i$, the maximum execution time of J_i
v_i	$= \frac{p_i}{x_i n_i}$ the density of J_i
δ-good	job J_i has $D_i \geq (1+2\delta)x_i$
δ-fresh	at time t, job J_i has $d_i - t \geq (1+\delta)x_i$
R	the set of jobs started by S
C	the set of jobs completed by S
U	unfinished jobs by S (that is, $R \setminus C$)
C^O	the set of jobs completed by OPT
$\mathcal{J}$	the set of all jobs
$T_O(v, \mathcal{E})$	the total work processed by the optimal schedule for the jobs in $\mathcal{E}$ that are v-dense
$T_S(v, \mathcal{E})$	the total number of processors steps S used for executing jobs in $\mathcal{E}$ that are v-dense

Table 9.2: Notations and definitions specific to jobs with deadlines

Notation	Definition
$p_i(t)$	the profit of job J_i if the job with arrival time r_i completes by $r_i + t$
n_i	$= \frac{(W_i - L_i)}{\frac{x_i^*}{1+2\delta} - L_i}$, the number of processors allocated to J_i
x_i	$= \frac{W_i - L_i}{n_i} + L_i$, the maximum execution time of J_i
v_i	$= \frac{p_i(D_i)}{x_i n_i}$ the density of J_i

Table 9.3: Notations and definitions specific to jobs with general profit functions

optimal solution. Our goal is to design a scheduler that achieves profit close to $\left\|C^O\right\|$. Throughout the proof, it will be useful to discuss the aggregate number of processors assigned to a job over all time.

9.2.1 Scheduler S for Maximizing Profit of Jobs with Deadlines

Here we present an algorithm S for jobs with deadlines and profits for which Theorem 50 holds.

On every time step, the algorithm S must decide which jobs to schedule and which ready nodes of each job to schedule. When a job J_i arrives, S calculates a value, n_i —

the number of processors "allocated" to J_i. On any time step when S decides to run J_i, it will always allocate n_i processors to J_i. In addition, since S is semi-non-clairvoyant, it is unable to distinguish between ready nodes of J_i; when it decides to allocate n_i nodes to J_i, it arbitrarily picks n_i ready nodes to execute if more than n_i nodes are ready.

In the assumption of Theorem 50, each job follows the condition that $(1+\epsilon)(\frac{W_i-L_i}{m}+L_i) \leq D_i$ for some positive constant ϵ.

We define the following **constants**. Let $\delta < \epsilon/2$, $c \geq 1 + \frac{1}{\delta\epsilon}$ and $b = (\frac{1+2\delta}{1+\epsilon})^{1/2} < 1$ be fixed constants. For each job J_i, the algorithm S calculates $n_i = \frac{(W_i-L_i)}{\frac{D_i}{1+2\delta}-L_i}$, where n_i is the number of processors S will give to J_i if it decides to execute J_i on a time step.

Let $x_i = \frac{W_i-L_i}{n_i} + L_i$, which is the number of time steps to complete J_i on n_i dedicated processors (regardless of the order the nodes are executed in). Therefore, job J_i can meet its deadline if it is given n_i dedicated processors for x_i time steps during $[r_i, d_i]$.

We define a **processor step** as a unit of time on a single processor and the **density** of a job as $v_i = \frac{p_i}{x_i n_i}$. Note that this is a non-standard definition of density. We define the density as $\frac{p_i}{x_i n_i}$ instead of $\frac{p_i}{W_i}$, because we think of J_i requiring $x_i n_i$ processor steps to complete by S. Thus, this definition of density indicates the potential profit per processor step that S can obtain by executing job J_i.

The scheduler S maintains jobs that have arrived but are unfinished in two priority queues. Queue Q stores all the jobs that have been *started* by S. Queue P stores all the jobs that have arrived but have not been started. In both queues, the jobs are sorted according to the density from high to low.

Job Execution: At each time step t, S picks a set of jobs in Q to execute in order from highest to lowest density. If a job J_i has been completed or if its absolute deadline d_i has passed ($d_i > t$), S removes the job from Q. When considering job J_i, if the number of unallocated processors is at least n_i the scheduler assigns n_i processors to J_i for execution.

Otherwise, it continues on to the next job. S stops this procedure when either all jobs have been considered or when there are no remaining processors to allocate.

A job J_i is δ-**good** if $D_i \geq (1+2\delta)x_i$. A job is δ-**fresh** at time t if $d_i - t \geq (1+\delta)x_i$. For any set T of jobs, let the set $A(T, v_1, v_2)$ contains all jobs in T with density within the range $[v_1, v_2)$. We define $N(T, v_1, v_2) = \sum_{J_i \in A(T,v_1,v_2)} n_i$. This is the total number of processors that S allocates to jobs in $A(T, v_1, v_2)$. We say that the set of job $A(T, v_1, v_2)$ *requires* $N(T, v_1, v_2)$ processors.

Adding Jobs: There are two types of events that may cause S to add a job to Q. Either a job arrives or S completes a job. When a job J_i arrives, S adds it to queue Q if it satisfies the following:

(1) J_i is δ-good;

(2) For all job $J_j \in Q \cup J_i$ it is the case that $N(Q \cup J_i, v_j, cv_j) \leq bm$. In words, the total number of processors required by jobs in $Q \cup J_i$ with density in the range $[v_j, cv_j)$ is no more than bm.

If these conditions are met, then J_i is inserted into queue Q; otherwise, job J_i is inserted into queue P. When a job is added to Q, we say that the job is **started** by S.

When a job completes, S considers the jobs in P from highest to lowest density but first removes all jobs with absolute deadlines that have passed. Then S checks if a job J_i in P can be moved to queue Q by checking if job J_i is δ-fresh and meets condition (2) from above. If both are true, then J_i is moved from queue P to Q.

Remark: Note that the Scheduler S pre-computes a fixed number of processors n_i assigned to each job; this may seem strange. We chose this design because n_i is approximately the minimum number of dedicated cores job J_i requires to complete by $\frac{D_i}{1+2\delta} \rightarrow D_i$, without knowing J_i's the DAG structure.

Outline of the Analysis of S: Our goal is to bound the total profit that S obtains. We first discuss some basic properties of S in Section 9.2.2. In Section 9.2.3 be bound the total profit of all the jobs S starts by the total profit of jobs that S completes. Then in Section 9.2.4 we bound the total profit of the jobs the optimal solution completes by the total profit of jobs that S starts. Putting these two together, we are able to bound the performance of S.

9.2.2 Properties of the Scheduler S

We begin by stating two observations regarding the parallel jobs.

Observation 55 *If a job J_i has all of its r ready nodes being executed by a schedule with speed s on m processors, where $r \leq m$, then the remaining critical-path length of J_i decreases at a rate of s.*

Observation 56 *Job J_i can meet its deadline if it is given n_i dedicated processors for x_i time steps in the interval $[r_i, d_i]$.*

We now show some structural properties for S that we will leverage in the proof. We first bound the number of processors n_i that S will allocate to job J_i.

Lemma 57 *For every job J_i we have that $n_i \leq b^2 m$.*

Proof. By assumption we know that

$$D_i \geq (1+\epsilon)(\tfrac{W_i - L_i}{m} + L_i)$$

The definition of n_i gives $n_i = \frac{W_i - L_i}{\frac{D_i}{1+2\delta} - L_i} \leq \frac{W_i - L_i}{\frac{1+\epsilon}{1+2\delta}(\frac{W_i - L_i}{m} + L_i) - L_i} \leq \frac{1+2\delta}{1+\epsilon} m = b^2 m.$ □

We now show that every job is δ-good.

Lemma 58 *Every job J_i is δ-good, i.e. $x_i(1+2\delta) \leq D_i$.*

Proof. Note that $L_i \leq \frac{1}{1+\epsilon} D_i$ by definition. Since $n_i = \frac{W_i - L_i}{\frac{D}{1+2\delta} - L_i}$, we have $x_i(1+2\delta) = (\frac{W_i - L_i}{n_i} + L_i)(1+2\delta) = (\frac{D_i}{1+2\delta} - L_i + L_i)(1+2\delta) \leq D_i$. □

The next lemma bounds the total number of processor steps occupied by a job.

Lemma 59 *$x_i n_i \le aW_i$, where a is $1 + \frac{1+2\delta}{\epsilon - 2\delta}$.*

Proof. By definition we have

$$\begin{aligned} x_i n_i = W_i - L_i + n_i L_i &\le W_i + \frac{W_i - L_i}{\frac{D_i}{1+2\delta} - L_i} L_i \le W_i + \frac{W_i - L_i}{\frac{D_i}{1+2\delta} - \frac{D_i}{1+\epsilon}} \Big(\frac{D_i}{1+\epsilon}\Big) \\ &\le W_i + \frac{(W_i - L_i) D_i (1+2\delta)}{D_i(\epsilon - 2\delta)} \le W_i + \frac{W_i(1+2\delta)}{\epsilon - 2\delta} \le W_i \Big(1 + \frac{1+2\delta}{\epsilon - 2\delta}\Big) \end{aligned}$$

□

Observation 60 *At any time and for any $v > 0$, the total number of processors required by all the jobs J_i that are in queue Q and have density $v \le v_i < cv$ is no more than bm, i.e. $N(Q, v_i, cv_i) \le bm$.*

Proof. Jobs are only added to queue Q when a new job arrives or a job completes. According to algorithm S, at both times, a job is only added to Q when this condition is satisfied. □

9.2.3 Bounding the Profit of Jobs Completed by S

In this section, we bound the profit of jobs completed by S compared to the profit of all jobs it ever starts (adds to Q). Let R denote the set of jobs S starts (that is, the set of jobs added to queue Q). Among the jobs in R, let C be the set of jobs it completes and U be the set of jobs that are unfinished. We say job J_i (and its assigned processors) is **v-dense**, if its density $v_i \ge v$. For any set A of jobs, define $\|A\|$ as $\sum_{i \in A} p_i$, the sum of the profits of jobs in the set.

Lemma 61 *For a job $J_i \in U = R \setminus C$ that was added to queue Q but does not complete by its deadline, S must have run cv_i-dense jobs for at least δx_i time steps where J_i is in Q using at least $(1-b)m$ processors at each such time.*

Proof. Since J_i is at least δ-fresh when added to Q and it does not complete by its deadline, there are at least δx_i time steps where S is not executing J_i by Observation 56. In each of these the time steps, all the m processors are executing v_i-dense jobs.

By Observation 60, jobs in Q with density in range $[v_i, cv_i)$ require at most $N(Q, v_i, cv_i) \leq bm$ processors to execute. Therefore, for each of the δx_i time steps, there are at least $(1-b)m$ processors executing cv_i-dense jobs. So the total number processor steps where cv_i-dense jobs are executing is at least $\delta x_i(1-b)m$. □

We now bound the profit of the jobs completed by their deadline under S by those started.

Lemma 62 $\|C\| \geq (\epsilon - \frac{1}{(c-1)\delta}) \|R\|$.

Proof. We use a charging scheme with credit transfers between the jobs. We give each job $J_i \in R$ a bank account B_i. Initially, all completed jobs (in C) are given p_i credits and other jobs (in U) have 0 credit. We will transfer credits between jobs in C and jobs in U. We want to show that after the credit transfer, every job J_i in R will have $B_i \geq (\epsilon - \frac{1}{(c-1)\delta})p_i$. This implies $\|C\| \geq (\epsilon - \frac{1}{(c-1)\delta}) \|R\|$.

Now we explain how credits are transferred. For each time step, a processor executing J_i will transfer $\frac{v_j n_j}{\delta bm}$ credits from B_i to every job J_j in queue Q that has density $v_j \leq \frac{v_i}{c}$.

For every job $J_j \in U$, Lemma 61 implies that there are at least δx_j time steps where at least $(1-b)m$ processors are executing cv_j-dense jobs. By our credit transfer strategy J_j will receive at least $\frac{v_j n_j}{\delta bm}$ credits from each processor in a time step. Therefore, the total credits J_j receives is at least

$$\delta x_j(1-b)m(\frac{v_j n_j}{\delta bm}) = v_j x_j n_j(\frac{1-b}{b}) = p_i(\frac{1-b}{b}).$$

This bounds the total amount of credit each job receives. We now show that not too much credit is transferred out of each job's account. We bound this on a job by job basis. Fix a job $J_i \in R$ and consider how many credits it transfers to other jobs during its execution.

By Observation 56, we know that J_i can execute for at most x_i time steps on n_i dedicated processors before its completion.

The job J_i will transfer credit to all jobs in Q with density less than $\frac{v_i}{c}$ at any point in time where J_i is being processed. These are the jobs in $A(Q, 0, \frac{v_i}{c})$. Fix an integer $l \geq 1$ and consider the set of jobs $A(Q, \frac{v_i}{c^{l+1}}, \frac{v_i}{c^l})$ in Q that have density within the range $[\frac{v_i}{c^{l+1}}, \frac{v_i}{c^l})$. Note that the total number of processors required by them is $N(Q, \frac{v_i}{c^{l+1}}, \frac{v_i}{c^l}) \leq bm$ by Observation 60. Knowing that a job J_j in $A(Q, \frac{v_i}{c^{l+1}}, \frac{v_i}{c^l})$ has density $v_j \leq \frac{v_i}{c^l}$ by definition it is the case that the total credits that J_i gives to jobs in $A(Q, \frac{v_i}{c^{l+1}}, \frac{v_i}{c^l})$ per processor assigned to J_i during any time step is at most

$$\begin{aligned}\sum_{J_j \in A(Q, \frac{v_i}{c^{l+1}}, \frac{v_i}{c^l})} \frac{v_j n_j}{\delta bm} \leq \sum_{J_j \in A(Q, \frac{v_i}{c^{l+1}}, \frac{v_i}{c^l})} \frac{\frac{v_i}{c^l} n_j}{\delta bm} = \frac{v_i}{\delta bmc^l} \sum_{J_j \in A(Q, \frac{v_i}{c^{l+1}}, \frac{v_i}{c^l})} n_j \\ = \frac{v_i}{\delta bmc^l} N(Q, \frac{v_i}{c^{l+1}}, \frac{v_i}{c^l}) \leq \frac{v_i}{\delta bmc^l} bm = \frac{v_i}{\delta c^l}.\end{aligned}$$

This bounds the total credit transferred to jobs in $A(Q, \frac{v_i}{c^{l+1}}, \frac{v_i}{c^l})$ during a time step for each processor assigned to J_i. We sum this quantity over all $l \geq 1$ and all n_i processors assigned to i to bound the total credit transferred from job J_i during a time step. Recall that $c > 1$ by definition.

$$\frac{n_i v_i}{\delta} \sum_{l=1}^{\infty} \frac{1}{c^l} = \left(\frac{n_i v_i}{\delta}\right) \frac{\frac{1}{c}}{1 - \frac{1}{c}} = \left(\frac{n_i v_i}{\delta}\right) \frac{1}{c-1}$$

Therefore, the total credits J_i transfers to all the jobs in $A(Q, 0, \frac{v_i}{c})$ over all times it is executed is at most $(\frac{x_i n_i v_i}{\delta})\frac{1}{c-1} = \frac{p_i}{(c-1)\delta}$ due to the fact that a job will be executed for at most x_i time steps in S's schedule.

Now we put these two observations together. Each job receives at least $p_i \frac{1-b}{b}$ credit and pays at most $\frac{p_i}{(c-1)\delta}$. After the credit transfer, the credits that a job J_i has is at least

$$p_i \frac{1-b}{b} - \frac{p_i}{(c-1)\delta} = p_i(\epsilon - \frac{1}{(c-1)\delta})$$

By our setting of c, this quantity is always positive. Therefore, we conclude that $\|C\| \geq (\epsilon - \frac{1}{(c-1)\delta}) \|R\|$. $\square$

9.2.4 Bounding the Profit of Jobs Completed by OPT

In this section, we bound the profit of the jobs OPT completes by all of the jobs that S starts. Our high level goal is to first bound the total amount of time OPT spends processing jobs that S does not complete by the time S spends processing jobs. Then using this and properties of S we will be able to bound the total profit of jobs OPT completes. At a high level, this follows since S focuses on processing high density jobs and OPT and S spend a similar amount of time processing jobs.

We begin by showing show that if not too many processors are executing $\frac{v_i}{c}$-dense jobs then all such jobs must be currently executing.

Lemma 63 *For any density v_i and time, if there are less than $b(1-b)m$ processors executing $\frac{v_i}{c}$-dense jobs, then all $\frac{v_i}{c}$-dense jobs in queue Q are executing and $N(Q, \frac{v_i}{c}, \infty) < b(1-b)m$.*

Proof. By definition, there are at least $m - b(1-b)m > bm - b(1-b)m = b^2m$ processors executing jobs with density less than $\frac{v_i}{c}$. For the sake of contradiction, suppose there is a $\frac{v_i}{c}$-dense job J_j that is not executing by S. By Lemma 57 we know that $n_j \leq b^2m$. Therefore, J_j would have been executed by S on the b^2m processors that are executing lower density jobs, a contradiction.

Now we know all all $\frac{v_i}{c}$-dense jobs in queue Q are executing. By assumption they are using less than $b(1-b)m$ processors and the lemma follows. $\square$

In the next lemma, we show that if not too many processors are running $\frac{v_i}{c}$-dense jobs then when a job arrives or completes, the schedule S will start processing a v_i-dense job that is δ-fresh, for any density v_i (if such a job exists). In particular, the job J_j will pass condition (2) of for adding jobs to Q in the definition of S.

Lemma 64 *Fix a density v_i. At a time where a new job arrives or a job completes if there are less than $b(1-b)m$ processors executing $\frac{v_i}{c}$-dense jobs, then a δ-fresh v_i-dense job J_j (arriving or in queue P) will be added to Q by S assuming such a job J_j exists.*

Proof. By Lemma 63, we know that all $\frac{v_i}{c}$-dense jobs in queue Q are executing on less than $b(1-b)m$ processors. By Lemma 57, we know that $n_j \leq b^2 m$. Therefore,

$$N(Q \cup J_j, \frac{v_i}{c}, \infty) < b(1-b)m + b^2 m = bm$$

Consider any δ-fresh job J_j that is also v_i-dense. Consider any job J_k where $J_j \in A(Q \cup J_i, v_k, cv_k)$. By definition of J_j being v_i-dense it must be the case that $A(Q \cup J_i, v_k, cv_k) \subseteq A(Q \cup J_j, \frac{v_i}{c}, \infty)$. The above implies that $N(Q \cup J_i, v_k, cv_k) \leq N(Q \cup J_j, \frac{v_i}{c}, \infty) \leq bm$. Thus, the condition (2) in our algorithm is satisfied. □

For an arbitrary set of jobs $\mathcal{E}$ and any $v \geq 0$ let $T_O(v, \mathcal{E})$ denote the total work processed by the optimal schedule for the jobs in $\mathcal{E}$ that are v-dense. We similarly let $T_S(v, \mathcal{E})$ be the total number of processors steps S used for executing jobs in $\mathcal{E}$ that are v-dense over all time. Now we are ready to bound the time that OPT spends on jobs S never adds to Q.

Lemma 65 *Consider the jobs in $\mathcal{J} \setminus R$, the jobs that are never added to Q. For all $v > 0$, $T_O(v, \mathcal{J} \setminus R) \leq \frac{1+2\delta}{\delta b(1-b)} T_S(\frac{v}{c}, \mathcal{J})$.*

Proof. Let $\{I_k = [s_k, e_k]\}$ be the set of maximal time intervals where at least $b(1-b)m$ processors are running $\frac{v}{c}$-dense jobs in S's schedule. Notice that by definition $\sum_{k=1}^{\infty}(e_k - s_k)b(1-b)m \leq T_S(\frac{v}{c}, \mathcal{J})$.

Consider a job in $J_i \in \mathcal{J} \setminus R$ that is both δ-good and v-dense and additionally arrives during $[s_k, s_{k+1})$. Note that during the intervals $[e_k, s_{k+1}]$, less than $b(1-b)m$ processors are executing $\frac{v}{c}$-dense jobs. Lemma 64 implies that if J_i arrives during $[e_k, s_{k+1}]$ it will be added to Q. This contradicts the assumption that $J_i \in \mathcal{J} \setminus R$. Therefore, J_i must arrive during $[s_k, e_k)$ and is in queue P at time e_k.

Note that at time e_k, the number of processors executing $\frac{v}{c}$-dense jobs decreases, so there must be a job that completes at time e_k. Again, by Lemma 64 if J_i is δ-fresh at time e_k then it will be added to Q at this time. Again, this contradicts $J_i \in \mathcal{J} \setminus R$. Thus, the only reason that S does not add J_i to Q is because J_i is not δ-fresh at time e_k. Knowing that J_i is δ-good at r_i and is not δ-fresh at e_k, we have

$$e_k - s_k \geq e_k - r_i \geq \delta x_i$$

At time e_k, J_i is not δ-fresh, so

$$d_i - e_k < (1+\delta)x_i < \frac{1+\delta}{\delta}(e_k - s_k)$$

Let K_k be the set of v-dense jobs that arrive during $[s_k, s_{k+1})$ but are not completed by S. Because OPT can only execute all jobs in K_k during $[s_k, d_i]$ on at most m processors, we get

$$T_O(v, K_k) \leq (d_i - s_k)m = ((d_i - e_k) + (e_k - s_k))m \leq \frac{1+2\delta}{\delta}(e_k - s_k)m$$

This completes the proof, as

$$T_O(v, U) = \sum_{k=1}^{\infty} T_O(v, K_k) \leq \sum_{k=1}^{\infty} (\frac{1+2\delta}{\delta}) m(e_k - s_k) \leq \frac{1+2\delta}{\delta} \frac{1}{b(1-b)} T_S(\frac{v}{c}, \mathcal{J})$$

□

Using the previous lemma, we are ready to bound the profit of jobs completed by OPT by the profit of jobs started by S.

Lemma 66

$$\left\|C^O\right\| \leq \left(1 + (1 + \frac{1+2\delta}{\epsilon - 2\delta})(1 + \frac{1}{\epsilon\delta})\frac{1+2\delta}{\delta b(1-b)}\right)\|R\|$$

.

Proof. We may assume WLOG that the adversary completes all jobs it starts. First we partition C^O, the jobs that the adversary completes, into C_R^O and C_S^O where $C_S^O = C^O \cap R$, that is, our algorithm started the job at some point. The remaining jobs are placed in C_R^O. Clearly $\left\|C_S^O\right\| \leq \|R\|$. Now it remains to bound $\left\|C_R^O\right\|$.

Consider every job in C_R^O and let the set of densities of these jobs be $\{\mu_1, \mu_2, \ldots, \mu_m\}$ from high to low and for notational simplicity let $\mu_0 = \infty$ and $\mu_{m+1} = 0$. Recall the adversary completed all jobs it started. Thus for each job with density μ_i, the adversary ran the job for a corresponding W_i processor steps. Let β_i denote the number of processor steps our algorithm takes to run jobs with densities within $(\frac{\mu_{i-1}}{c}, \frac{\mu_i}{c}]$.

We have $T_O(v, \mathcal{J} \setminus R) \leq \frac{1+2\delta}{\delta b(1-b)} T_S(\frac{v}{c}, \mathcal{J})$ from Lemma 65 for all densities v. Equivalently for any given density v:

$$T_O(v, \mathcal{J} \setminus R) = \sum_{i=1}^{v} W_i \leq \frac{1+2\delta}{\delta b(1-b)} \sum_{i=1}^{v} \beta_i = \frac{1+2\delta}{\delta b(1-b)} T_S(\frac{v}{c}, \mathcal{J})$$

We then sum over all densities. The subtraction of densities is necessary to insure that each density is only counted a single time.

$$\sum_{v=1}^{m}\left((\mu_v - \mu_{v+1})\sum_{i=1}^{v} W_i\right) \leq \sum_{v=1}^{m}\left((\mu_v - \mu_{v+1})\frac{1+2\delta}{\delta b(1-b)}\sum_{i=1}^{v}\beta_i\right)$$

The LHS can be simplified:

$$\sum_{v=1}^{m}\left((\mu_v-\mu_{v+1})\sum_{i=1}^{v}W_i\right)=\sum_{i=1}^{m}W_i\sum_{v=i}^{m}(\mu_v-\mu_{v+1})=\sum_{i=1}^{m}W_i(\mu_i-\mu_{m+1})=\sum_{i=1}^{m}W_i\mu_i$$

The RHS similarly simplifies to $\frac{1+2\delta}{\delta b(1-b)}\sum_{i=1}^{m}\beta_i\mu_i$, leading to the inequality that $\sum_{i=1}^{m}W_i\mu_i \le \frac{1+2\delta}{\delta b(1-b)}\sum_{i=1}^{m}\beta_i\mu_i$. Recall that densities such as μ_i are defined by $\mu_i=\frac{p_i}{x_i n_i}$ and $x_i n_i \le aW_i$. Therefore:

$$\sum_{i=1}^{m}W_i\mu_i=\sum_{i=1}^{m}\frac{W_i p_i}{x_i n_i}\ge\sum_{i=1}^{m}\frac{W_i p_i}{aW_i}\ge\sum_{i=1}^{m}\frac{p_i}{(1+\frac{1+2\delta}{\epsilon-2\delta})}=\frac{1}{(1+\frac{1+2\delta}{\epsilon-2\delta})}\left\|C_R^O\right\|$$

And also, by the definition of β_i, we know that $\sum_{i=1}^{m}\beta_i\frac{\mu_i}{c}\le\|R\|$. Combining these results, we get:

$$\begin{aligned}&\frac{1}{(1+\frac{1+2\delta}{\epsilon-2\delta})}\left\|C_R^O\right\|\le\sum_{i=1}^{m}W_i\mu_i\le\frac{1+2\delta}{\delta b(1-b)}\sum_{i=1}^{m}\beta_i\mu_i\le\frac{1+2\delta}{\delta b(1-b)}c\left\|R\right\|\\&\Rightarrow\left\|C_R^O\right\|\le\left(1+\frac{1+2\delta}{\epsilon-2\delta}\right)\left(\frac{1+2\delta}{\delta b(1-b)}\right)c\left\|R\right\|\\&\Rightarrow\left\|C^O\right\|=\left\|C_R^O\right\|+\left\|C_S^O\right\|\le\left(1+(1+\frac{1+2\delta}{\epsilon-2\delta})(1+\frac{1}{\epsilon\delta})\frac{1+2\delta}{\delta b(1-b)}\right)\left\|R\right\|\end{aligned}$$

□

Finally we are ready to complete the proof, bounding the profit OPT obtains by the total profit the algorithm obtains for jobs it completed.

Lemma 67

$$\left\|C^O\right\|\le\frac{\left(1+(1+\frac{1+2\delta}{\epsilon-2\delta})(1+\frac{1}{\epsilon\delta})\frac{1+2\delta}{\delta b(1-b)}\right)}{\epsilon-\frac{1}{(c-1)\delta}}\left\|C\right\|$$

Proof. This is just by combination of Lemma 62 and Lemma 66. □

Therefore, we prove Theorem 50 by showing that scheduler S is $O(\frac{1}{\epsilon^6})$-competitive for jobs with deadlines and profits, when $(1+\epsilon)(\frac{W_i-L_i}{m}+L_i)\le D_i$.

9.3 Maximizing Profit of Jobs with General Profit Functions

In this section, we focus on a more general case. In particular, each job J_i has a non-negative non-increasing profit function $p_i(t)$ indicating its profit if the job with arrival time r_i completes by $r_i + t$. Our goal is to design a scheduler that maximize the profit to make it close to what the optimal solution can obtain, denoted as $\|O\|$.

9.3.1 Scheduler S' for Maximizing General Profit

The algorithm S' for jobs with general profit functions is similar to S. Due to space, we briefly sketch it and point out the differences.

Assigning cores, deadlines and slots to jobs: When a job J_i arrives, S' calculates a relative deadline D_i and a set of time steps I_i with n_i processors. I_i are the only time steps in which J_i is allowed to run. In each time step t in I_i, we say that J_i is assigned to t.

Note that for the general profit problem, a job J_i has no deadline. Thus, S' computes a D_i by searching all the potential deadlines D to find the minimum valid deadline using a complicated process which we omit due to space. The set of time steps I_i is then determined using the chosen deadline D_i.

From the assumption in Theorem 53, for each job J_i the profit function stays the same until $x_i^* \geq (\frac{W_i - L_i}{m} + L_i)(1+\epsilon)$. We set $n_i = \frac{W_i - L_i}{x_i^*/(1+2\delta) - L_i}$, where $\delta < \epsilon/2$. We define its the **density** as $v_i = \frac{p_i(D_i)}{x_i n_i} = \frac{p_i(D_i)}{W_i + (n_i - 1)L_i}$, where $x_i := \frac{W_i - L_i}{n_i} + L_i$.

Executing jobs: This procedure is similar to S, with the only difference that S' only picks jobs to execute that have been assigned to time step t.

Remark: Unlike the scheduler for jobs with deadlines, here we try to complete a job J_i by a calculated deadline D_i that is as close to x_i^* as possible. This is because the obtained profit decreases as the completion time increases but there is no additional benefit for completing a job J_i before time x_i^*. With a carefully designed deadline D_i, we are able to prove the performance bound of the scheduler. Similarly to Section 9.2, we start by stating the basic properties of the scheduler S', followed by bounding the total profit obtained by S'. However, the proofs that bound the profit of jobs that are completed by OPT differ greatly from that for jobs with deadlines. This is because in addition to losing the profit of jobs that do not complete by their assigned deadlines, scheduler S' can also loses profit compared to OPT if the completion time of a job under S' is later than under OPT. By taking into account all these jobs, we are able to bound the performance of S' for jobs with general profit functions.

9.3.2 Properties of the Scheduler S'

We begin by showing some structural properties for S' that we will leverage in the proof and can be obtained directly from the algorithm of scheduler S'. Note that these lemmas are the almost the same as the lemmas shown in Section 9.2.2 if we replace x_i* with D_i. We state them here again for completeness.

Lemma 68 *For every job J_i we have that $n_i \leq b^2 m$, where $b = (\frac{1+2\delta}{1+\epsilon})^{1/2}$.*

Proof. By definition, we know that $x_i^* \geq (1+\epsilon)(\frac{W_i - L_i}{m} + L_i)$. Therefore, we have

$$n_i = \frac{W_i - L_i}{\frac{x_i^*}{1+2\delta} - L_i} \leq \frac{W_i - L_i}{\frac{1+\epsilon}{1+2\delta}(\frac{W_i - L_i}{m} + L_i) - L_i} \leq \frac{1+2\delta}{1+\epsilon} m = b^2 m$$

□

Lemma 69 *Under scheduler S', we have $x_i n_i \leq a W_i$ and $v_i \geq \frac{p_i(D_i)}{aW_i}$, where $a = 1 + \frac{1+2\delta}{\epsilon - 2\delta}$.*

Proof. By definition, $x_i^* > L_i(1+\epsilon)$. Therefore, we have

$$\begin{aligned} x_i n_i = W_i - L_i + n_i L_i = W_i + \frac{W_i - L_i}{\frac{x_i^*}{1+2\delta} - L_i} L_i \leq W_i + \frac{W_i - L_i}{\frac{x_i^*}{1+2\delta} - \frac{x_i^*}{1+\epsilon}} \Big(\frac{x_i^*}{1+\epsilon}\Big) \\ \leq W_i + \frac{(W_i - L_i) x_i^* (1+2\delta)}{x_i^*(\epsilon - 2\delta)} \leq W_i \Big(1 + \frac{1+2\delta}{\epsilon - 2\delta}\Big) \end{aligned}$$

Therefore, we have $v_i = \frac{p_i(D_i)}{x_i n_i} \geq \frac{p_i(D_i)}{aW_i}$. □

Lemma 70 *For every job J_i with the assignment n_i, D_i and I_i, Job J_i can meet its deadline D_i, if it is executed by S' for at least x_i time steps in I_i (on n_i dedicated processors).*

Lemma 71 *For every job J_i, $x_i(1+2\delta) \leq x_i^*$.*

Proof. Note that $L_i \leq \frac{1}{1+\epsilon} D_i$ by requirement of potential assignment. Since $n_i = \frac{W_i - L_i}{\frac{x_i^*}{1+\epsilon} - L_i}$, we have $x_i(1+2\delta) = (\frac{W_i - L_i}{n_i} + L_i)(1+2\delta) \leq (\frac{x_i^*}{1+\epsilon} - L_i + L_i)(1+2\delta) = \frac{x_i^*}{1+\epsilon}(1+2\delta) \leq x_i^*$. □

Lemma 72 *At any time step t during the execution and for any density range $[v, cv)$, the total number of cores required by all the jobs $J_i \in J(t)$ (that have been assigned to t) with density $v \leq v_i < cv$ is no more than bm, i.e. $N(J(t), v_i, cv_i) \leq bm$.*

9.3.3 Bounding the Profit of Jobs Completed by S'

Similar to Section 9.2.3, we bound the profit of jobs completed by scheduler S' compared to the profit of all jobs. Let $\mathcal{J}$ denote the set of jobs arrived during the execution, C denote the set of jobs that actually complete before their deadlines assigned by S', and $U = \mathcal{J} \setminus C$ be the set of jobs that didn't finish by their deadlines assigned by S'. We say job J_i (and its assigned processors during execution) is v-**dense**, if its density $v_i \geq v$. For any set A of jobs, define $\|A\|$ as $\sum_{J_i \in A} p_i(D_i)$, the sum of the profits of jobs in the set under S'.

Lemma 73 *For a job $J_i \in \mathcal{J} \setminus C$ that does not complete by its deadline, the number of time steps in I_i where S' runs cv_i-dense jobs using at least $(1-b)m$ processors is at least δx_i.*

Proof. From Lemma 70, we know that job J_i can complete if it can execute for x_i time steps by S'. Also note that according to the assignment process $(1+\delta)x_i = \|I_i\|$, where $\|I_i\|$ is the number of time steps assigned to J_i during $[r_i, r_i + D_i]$. Since it does not complete by its deadline, there are at least δx_i time steps in I_i where S' does not execute J_i.

Consider each of these time steps t. According to Lemma 72, jobs in $J(t)$ with density in range $[v_i, cv_i)$ require at most $N\left(J(t), v_i, cv_i\right) \leq bm$ processors to execute. Therefore, there must be at least $(1-b)m$ processors executing cv_i-dense jobs. Otherwise, S' would execute all jobs in $A\left(J(t), v_i, cv_i\right)$, which includes job J_i. □

Lemma 74 $\|C\| \geq (\epsilon - \frac{1}{(c-1)\delta}) \|\mathcal{J}\|$.

Proof. Similar to Lemma 62, we use a charging scheme to transfer credit between the jobs. We give each job $J_i \in \mathcal{J}$ an account B_i. In the beginning, all completed jobs (in C) are given $p_i(D_i)$ credits and other jobs (in U) have 0 credit. We will transfer credits between jobs in C and jobs in U. We want to show that after the credit transfer, every job J_i will have $B_i \geq (\epsilon - \frac{1}{(c-1)\delta}) \times p_i(D_i)$. This implies $\|C\| \geq (\epsilon - \frac{1}{(c-1)\delta}) \|\mathcal{J}\|$.

Now we explain how credits are transferred. For each time step t, a processor executing J_i will transfer $\frac{v_j n_j}{\delta bm}$ credits from B_i to every job $J_j \in J(t)$ that has density $v_j \leq \frac{v_i}{c}$.

For every job $J_j \in U$, Lemma 73 implies that there are at least δx_j time steps in I_j where at least $(1-b)m$ processors are executing cv_j-dense jobs. By our credit transfer strategy J_j will receive at least $\frac{v_j n_j}{\delta bm}$ credits from each processor in such a time step. Therefore, the total credits J_j receives is at least

$$\delta x_j (1-b) m \left(\frac{v_j n_j}{\delta bm}\right) = v_j x_j n_j (\frac{1-b}{b}) = p_i(D_i) \times (\frac{1-b}{b})$$

This bounds the total amount of credit each job receives. We now show that not too much credit is transferred out of each job's account. We bound this on a job by job basis. Fix a job $J_i \in \mathcal{J}$ and let's consider how many credits it transfers to other jobs during its execution.

Consider each time step t where J_i is being processed, it will transfer credit to all jobs in $J(t)$ with density less than $\frac{v_i}{c}$. These are the jobs in $A(J(t), 0, \frac{v_i}{c})$. Fix an integer $l \geq 1$ and consider the set of jobs $A(J(t), \frac{v_i}{c^{l+1}}, \frac{v_i}{c^l})$ in $J(t)$ that have density within the range $[\frac{v_i}{c^{l+1}}, \frac{v_i}{c^l})$.

Note that the total number of processors required by them is $N(J(t), \frac{v_i}{c^{l+1}}, \frac{v_i}{c^l}) \leq bm$ by Lemma 72. Knowing that a job J_j in $A(J(t), \frac{v_i}{c^{l+1}}, \frac{v_i}{c^l})$ has density $v_j \leq \frac{v_i}{c^l}$ by definition, it is the case that the total credits that J_i gives to jobs in $A(J(t), \frac{v_i}{c^{l+1}}, \frac{v_i}{c^l})$ per processor assigned to J_i is at most

$$\sum_{J_j \in A(J(t), \frac{v_i}{c^{l+1}}, \frac{v_i}{c^l})} \frac{v_j n_j}{\delta bm} \leq \sum_{J_j \in A(J(t), \frac{v_i}{c^{l+1}}, \frac{v_i}{c^l})} \frac{\frac{v_i}{c^l} n_j}{\delta bm} = \frac{v_i}{\delta bmc^l} \sum_{J_j \in A(J(t), \frac{v_i}{c^{l+1}}, \frac{v_i}{c^l})} n_j$$
$$= \frac{v_i}{\delta bmc^l} N(J(t), \frac{v_i}{c^{l+1}}, \frac{v_i}{c^l}) \leq \frac{v_i}{\delta bmc^l} bm = \frac{v_i}{\delta c^l}.$$

This bounds the credits transferred to jobs in $A(J(t), \frac{v_i}{c^{l+1}}, \frac{v_i}{c^l})$ during time step t for each processor assigned to J_i. We sum this quantity over all $l \geq 1$ and all n_i processors assigned to J_i to bound the total credit transferred from job J_i during a time step where J_i is being processed. Recall that $c > 1$ by definition.

$$\frac{n_i v_i}{\delta} \sum_{l=1}^{\infty} \frac{1}{c^l} = \left(\frac{n_i v_i}{\delta}\right) \frac{\frac{1}{c}}{1 - \frac{1}{c}} = \left(\frac{n_i v_i}{\delta}\right) \frac{1}{c-1}$$

By the definition of x_i, we know that J_i can execute for at most x_i time steps on n_i dedicated processors with before its completion. Therefore, the total credits J_i transfers to all the jobs in $A(J, 0, \frac{v_i}{c})$ over all the time steps where it is executing is at most $(\frac{x_i n_i v_i}{\delta})\frac{1}{c-1} = \frac{p_i(D_i)}{(c-1)\delta}$.

Now we put these two observations together. Each job receives at least $p_i \frac{1-b}{b}$ credit and pays at most $\frac{p_i}{(c-1)\delta}$. After the credit transfer, the credits that a job J_i has is at least

$$p_i(D_i) \times (\frac{1-b}{b}) - \frac{p_i(D_i)}{(c-1)\delta} = p_i(D_i) \times (\epsilon - \frac{1}{(c-1)\delta})$$

We conclude that $\|C\| \geq (\epsilon - \frac{1}{(c-1)\delta}) \|\mathcal{J}\|$. □

9.3.4 Bounding the Profit of Jobs Completed by OPT

Similar to Section 9.2.4, we will now bound the profit of the jobs OPT completes. We are first going to consider the number of processor steps OPT spends on jobs that S' finishes later than OPT. For these jobs, we assume that S' makes no profit since the profit function may become 0 as soon as OPT finishes it. Our high level goal is to first bound the total number of processor steps OPT spends on these jobs, which will allow us to bound OPT's profit. This section of the proof differ greatly from the throughput case.

We begin by showing that if not too many processors are executing $\frac{v_i}{c}$-dense jobs then all such jobs must be currently processed under S'.

Lemma 75 *Consider a job J_i and a time $t^* < D_i$. For any time step $t \in [r_i, r_i + t^*] \setminus I_i$ (that is not added to I_i by S'), the total number of processors required by $\frac{v_i}{c}$-dense jobs in $J(t)$ must be more than $b(1-b)m$, i.e., $N(J(t), \frac{v_i}{c}, \infty) > b(1-b)m$.*

Proof. Because $t \in [r_i, r_i + t^*] \setminus I_i$ and $t^* < D_i$, we know that time step t is before D_i.

Since t is not added to I_i, it must be the case that for some density $v_j \in (\frac{v_i}{c}, v_i]$, the required condition is not true, i.e., $N\left(J(t) \cup J_i, v_j, cv_j\right) > bm$. Note that v_j must be in the range $(\frac{v_i}{c}, v_i]$. This is because without assigning J_i to time step t it is true that $N\left(J(t), v_j, cv_j\right) \leq bm$ according to S', therefore J_i must have a density within the range of $[v_j, cv_j)$ in order to make impact.

By Lemma 68, we know that $n_i \leq b^2 m$. Thus, we have

$$N\left(J(t), v_j, cv_j\right) = N\left(J(t) \cup J_i, v_j, cv_j\right) - n_i > bm - b^2 m = b(1-b)m$$

Therefore, we obtain $N(J(t), \frac{v_i}{c}, \infty) \geq N\left(J(t), v_j, cv_j\right) > b(1-b)m$. □

Let O be the set of jobs completed by OPT. For each job $J_i \in O$, let d be the difference between J_i's completion time and arrival time under OPT; the profit of J_i under OPT is $p_i(d)$. According to the assumption in Theorem 53, we know that if $d \leq x_i^*$, then $p_i(d) =$

$p_i(x_i^*)$ for some $x_i^* \geq (\frac{W_i - L_i}{m} + L_i)(1+\epsilon)$. Therefore, we can assume that OPT assigns a relative deadline D_i^* to J_i, where $D_i^* = \max\{d, x_i^*\}$. Thus, OPT obtains a profit of $p_i(d) = p_i(D_i^*)$.

Lemma 76 *Consider a job J_i such that D_i assigned by scheduler S' is larger than the deadline D_i^* assigned by* OPT*, i.e., $D_i > D_i^*$, the number of time steps during $[r_i, r_i + D_i^*)$ where scheduler S' is actively executing $\frac{v_i}{c}$-dense jobs on at least $b(1-b)m$ cores is at least $\frac{\delta}{1+2\delta} D_i^*$.*

Proof. By definition of D_i^* and Lemma 71, we know that $D_i^* \geq x_i^*$.

Consider the number of time steps in time interval $[r_i, r_i + D_i^*]$ that are added to I_i, it must be less than $(1+\delta)\left(\frac{W_i - L_i}{n_i} + L_i\right) = (1+\delta)x_i$; otherwise, D_i^* would be a valid deadline under scheduler S' with higher profit. Therefore, the number of time steps in $[r_i, r_i + D_i^*] \setminus I_i$ is more than $D_i^* - (1+\delta)x_i \geq D_i^* - \frac{1+\delta}{1+2\delta} x_i^* \geq D_i^* - \frac{1+\delta}{1+2\delta} D_i^* = \frac{\delta}{1+2\delta} D_i^*$.

By Lemma 75, we know that for each time step $t \in [r_i, r_i + D_i^*] \setminus I_i$, the total number of processors required by $\frac{v_i}{c}$-dense jobs in $J(t)$ must be more than $b(1-b)m$. Therefore, there must be at least $b(1-b)m$ cores executing $\frac{v_i}{c}$-dense jobs under scheduler S' at time step t and the number of such steps is at least $\frac{\delta}{1+2\delta} D_i^*$. □

Among the jobs in O, let O_1 be the set of jobs that the deadline D_i assigned by scheduler S' is no larger than that assigned by OPT, i.e., $D_i \leq D_i^* < \infty$. In other words, the obtained profit of these jobs under scheduler S' is no less than that under OPT, i.e., $p_i(D_i) \geq p_i(D_i^*)$, since the profit function $p_i(t)$ is non-increasing. Let O_2 be the remaining jobs $O_2 = O \setminus O_1$.

Let $\|X\|^*$ be the total profit that OPT obtains from jobs in X and $\|X\|$ be the total profit that S' obtains from jobs in X. For jobs in O_1, we have $\|O_1\|^* \leq \|O_1\|$.

For an arbitrary set of jobs $\mathcal{E}$ and any $v \geq 0$ let $T_O(v, \mathcal{E})$ denote the total work processed by the optimal schedule for the jobs in $\mathcal{E}$ that are v-dense. Let β_i denote the total number of time steps where S' is actively processing job J_i. By definition, we have $\beta_i \leq \frac{x_i}{1+\epsilon}$. We similarly let $T_S(v, \mathcal{E})$ be the summation of $\beta_i n_i$ over all jobs i in $\mathcal{E}$ that are v-dense. Note that this counts the total number of processor steps S' executes jobs in $\mathcal{E}$ that are v-dense over all time.

Now we are ready to bound the time that OPT spends on jobs O_2 that scheduler S' obtains less profit than OPT.

Lemma 77 *Consider a job J_i in O_2, the deadline D_i assigned by scheduler S' is longer than deadline D_i^* assigned by* OPT*. For all $v > 0$, $T_O(v, O_2) \leq \frac{2(1+2\delta)}{\delta b(1-b)} T_S(\frac{v}{c}, \mathcal{J})$.*

Proof. For any job $J_i \in O_2$, we denote the lifetime of J_i under OPT as the time interval $[r_i, r_i + D_i^*)$, where D_i^* is the deadline assigned by OPT. For any density $v > 0$, let l be the number of time steps of the union of the lifetimes of all jobs in $A(O_2, v, \infty)$. By definition, $T_O(v, O_2) \leq lm$, since OPT can execute them on at most m processors.

Let $M \subseteq O_2$ be the minimum subset of O_2 that the union of the lifetimes of jobs in M covers the same time intervals of jobs in O_2. By the minimality of M, we know that at any time t, there are at most two jobs in M that cover time t. Therefore, we can further partition M into two sets M_1 and M_2, where for any two jobs in M_1 or any two jobs in M_2, their lifetimes do not overlap. By definition, either M_1 or M_2 has a union lifetime that is at least $l/2$ and we assume WLOG it is M_1.

Consider $J_i \in M_1$ and let k_i be the number of time steps during its lifetime $[r_i, r_i + D_i^*)$ where scheduler S' is actively executing $\frac{v_i}{c}$-dense jobs on at least $b(1-b)m$ cores. By Lemma 76, we know $k \geq \frac{\delta}{1+2\delta} D_i^*$. Therefore, during $[r_i, r_i + D_i^*)$ the number of processor steps where S' is processing $\frac{v_i}{c}$-dense jobs is at least $b(1-b)m\frac{\delta}{1+2\delta} D_i^*$.

Let $K = \sum_{M_1} k_i$, be the total number of processor steps where S' is processing $\frac{v}{c}$-dense jobs (since $v_i \geq v$) during the intervals in M_1. Thus, by definition,

$$K \geq \frac{\delta b(1-b)}{1+2\delta} m \sum_{J_i \in M_1} D_i^* > \frac{\delta b(1-b)}{1+2\delta} m \times \frac{l}{2} \geq \frac{\delta b(1-b)}{2(1+2\delta)} T_O(v, O_2)$$

Clearly, by adding additional intervals that are not in M_1, we have $T_S(\frac{v}{c}, \mathcal{J}) \geq K > \frac{\delta b(1-b)}{2(1+2\delta)} T_O(v, O_2)$, which gives us the bound. □

Lemma 78 *Comparing the total profit of O obtained by* OPT *and $\mathcal{J}$ obtained by S',*

$$\|O\|^* \leq \left(1 + (1 + \frac{1+2\delta}{\epsilon - 2\delta})(1 + \frac{1}{\epsilon\delta})\frac{2(1+2\delta)}{\delta b(1-b)}\right)\|\mathcal{J}\|$$

Proof. First, by the definition of O_1 and O_2, we have $\|O\|^* = \|O_1\|^* + \|O_2\|^*$ and $\|O_1\|^* \leq \|O_1\| \leq \|\mathcal{J}\|$. Now it remains to bound $\|O_2\|$.

We have $T_O(v, O_2) \leq \frac{2(1+2\delta)}{\delta b(1-b)} T_S(\frac{v}{c}, \mathcal{J})$ from Lemma 77 for all densities v. The remaining proof for the lemma is similar to that in Lemma 66, except for a different constant. Therefore, $\|O_2\|^* \leq (1 + \frac{1+2\delta}{\epsilon-2\delta})c\frac{2(1+2\delta)}{\delta b(1-b)}\|\mathcal{J}\|$. Finally, we have

$$\|O\|^* = \|O_1\|^* + \|O_2\|^* \leq \left(1 + \left(1 + \frac{1+2\delta}{\epsilon - 2\delta}\right)c\frac{2(1+2\delta)}{\delta b(1-b)}\right)\|\mathcal{J}\|$$

□

Finally we are ready to complete the proof, bounding the profit OPT obtains by the total profit the algorithm obtains for jobs it completed.

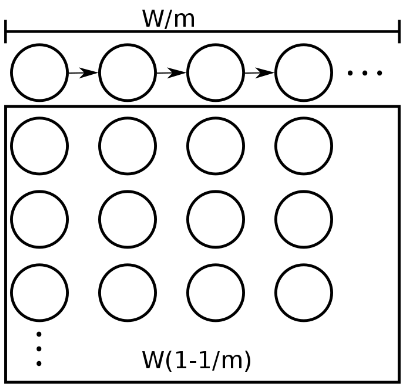
Lemma 79 $\left\|C^O\right\| \leq \frac{1+ac\frac{2(1+2\delta)}{\delta b(1-b)}}{\epsilon - \frac{1}{(c-1)\delta}}\|C\|$.

Proof. This is just by combination of Lemma 74 and Lemma 78. □

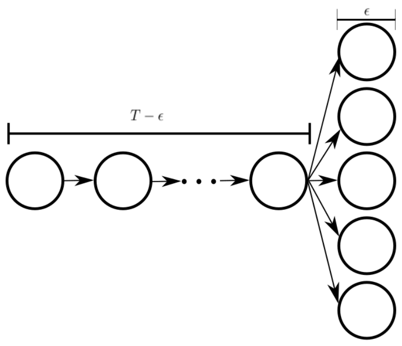
9.4 Lower Bound Examples

In this section, we will give some example DAGs to show why Theorem 50 is close to the best theorem we can hope for using two examples.

The first example, shown in Figure 9.1(a), shows the limitations of semi-non-clairvoyance. In particular, a semi-non-clairvoyant scheduler does not know the structure of the DAG in advance since the DAG unfolds dynamically. At any time step, the scheduler only knows



(a) Example 1



(b) Example 2

the ready nodes available for execution. Given this limitation, consider the DAG shown in Figure 9.1(a). This job has one sequential chain with length $L = \frac{W}{m}$, where W is the total work of the job and m is the number of processors. The remaining $W - W/m$ work are fully parallelizable in a block and can also be done in parallel with the chain. Therefore, L is the span of the jobs. Since a semi-non-clairvoyant scheduler cannot distinguish between ready nodes, it may make unlucky choices and execute the entire block of $W - W/m = W - L$ ready nodes first in $(W - L)/m$ time steps and then execute the chain of L nodes sequentially — leading to a total time of $(W - L)/m + L$. On the other hand, a fully clairvoyant scheduler can execute the entire DAG in W/m time. Therefore, a semi-non-clairvoyant scheduler needs at least $2 - 1/m$ speed augmentation to ensure that it can complete the DAG at the same time as OPT.

We now show another example DAG indicating that it would be reasonable to always set deadlines as $D \geq (W - L)/m + L$ if we do not know the structure of the DAG a priori. Figure 9.1(b) shows an example DAG, which consists of a chain of $L - \epsilon$ nodes followed by $W - L + \epsilon$ nodes that can run in parallel. Each node in the DAG takes ϵ time to run, so the total work of the DAG is W and the span is L. For such a DAG, even a fully clairvoyant scheduler needs $L - \epsilon + \frac{W-L+\epsilon}{m} = \frac{W-L}{m} + L - \epsilon(1 - \frac{1}{m})$, which approaches to $\frac{W-L}{m} + L$ when $\epsilon \to 0$.

Chapter 10

Scheduling Parallel Jobs Online to Minimize the Maximum Flow Time

In today's systems, response time, or latency, is often a very important measure of performance. For interactive services on clouds and servers, the platform scheduler is often interested in minimizing the maximum latency experienced by a job once it has been submitted to be processed. In addition, these services often run on large parallel machines with many processors and it is important to utilize these parallel machines efficiently to process these requests. In this chapter, we consider the problem of minimizing the maximum latency or **maximum flow time**. Formally, given n parallel jobs and a parallel machine with m processors, the scheduling goal is to minimize the amount of time between a job's arrival and its completion, over all jobs. We consider the online version of this problem, where jobs arrive dynamically. The objective of maximum flow time is the natural generalization of the makespan[11] objective to the case where jobs arrive over time. We also assume that the scheduler has no knowledge of the job arrival times or work distribution.

This chapter focuses on parallel directed acyclic graph (DAG) programs. Scheduling a single parallel DAG program has been studied extensively in the parallel computing literature. Parallel runtime systems generally use work-stealing as a scheduler since it is known to be an efficient scheduler for such programs both theoretically and in practice [33, 75]. A single parallel program having W *work* — the running time on 1 processor, and P *critical-path length* — the length of the critical path (the longest path in the program), can be executed in $O(W/m + P)$ (expected) time on m processors (or workers) using a work-stealing scheduler.

[11]The makespan of a schedule is the time that last job completes and this objective is popular when all jobs arrive at the same time.

This running time is asymptotically optimal and guarantees linear speedup for programs with sufficient parallelism.

However, the problem of how to schedule these parallel programs in multiprogrammed environments where a quality of service guarantee must be provided is not well studied. There has been some prior work on how to allocate processors to programs in a fair and efficient manner [2] and some further work on using it to provide mean completion time guarantees [84], but none of the work considers maximum flow time.

On the other hand, the multiprogrammed scheduling problem is well-studied for the case where each job is sequential, i.e. can only use one processor at a time. In particular, it is known that the algorithm First-In-First-Out (FIFO) is $(3/2 - \frac{1}{m})$-competitive [3, 28]. A related problem that has been considered for sequential jobs is when jobs have weights where weight represents some sort of priority of the job (not necessarily correlated with the job's work). In this case, the scheduler is interested in minimizing the maximum weighted flow time. For this setting, it is known that any algorithm is $\Omega(W^{.4})$-competitive where W is the ratio of the maximum weight to minimum weight. This is true even when jobs are sequential and unit sized [50].

Due to this strong lower bound, previous work has considered a *resource augmentation* analysis [93] where the algorithm is given extra speed over the adversary. An ***s*-speed *c*-competitive** algorithm achieves a competitive ratio of c when given processors s times the speed of the optimal schedule. An ideal algorithm is $(1 + \epsilon)$-speed $O(f(\epsilon))$-competitive for any $\epsilon > 0$ where $f(\epsilon)$ is some function that only depends on ϵ. That is, an algorithm which achieves constant competitiveness with the minimum possible resource augmentation. Such an algorithm is referred to as *scalable.* Finding a scalable algorithm is the best positive result one can hope for when strong lower bounds exists without resource augmentation. For jobs with weights, a scalable algorithm is known when jobs are sequential [50].

This chapter presents several theoretical results for minimizing maximum flow time for parallel jobs. These are the first known non-trivial results for maximum flow time in the

DAG model. All of the algorithms considered in this chapter are *non-clairvoyant*, meaning that they have no prior knowledge of the size or structure of the jobs or when they arrive. In particular, our contributions are as follows:

1. Section 10.2 starts with an idealized FIFO scheduler — at each time step, FIFO looks at jobs in the order of arrival and allocates each job as many processors it can use until it runs out of jobs or processors. We prove that FIFO is $(1+\epsilon)$-speed $O(\frac{1}{\epsilon})$-competitive.

2. Section 10.3 then generalizes the result to work stealing. Work stealing is a practical and efficient scheduler that is used in many parallel languages and libraries. In comparison, an implementation of the ideal FIFO scheduler is likely to have high overhead since it is centralized and potentially preempts jobs and re-allocates processors at every time step. For work stealing, we prove that a version of it, called *admit-first*, is scalable for "reasonable jobs". In particular, we show that admit-first with $(1+\epsilon)$-speed has maximum flow time $O(\frac{1}{\epsilon^2} \max\{\mathrm{OPT}, \ln(n)\})$ over n jobs for any fixed $\epsilon > 0$ with high probability. Note that if any job has span $\Omega(\lg n)$ or work $\Omega(m \lg n)$, then $\mathrm{OPT} \geq \ln n$ and admit-first is scalable with $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^2})$-competitive with high probability.

3. We introduce a generalization of admit-first scheduler, called steal-k-first. Our goal in this generalization is to design a work-stealing scheduler that is closest to FIFO since intuitively FIFO is the 'right' scheduling policy for maximum flow time, but is inefficient in implementation. Steal-k-first is parameterized by k. Intuitively as k becomes larger, this algorithm becomes closer to the FIFO scheduler. Theoretically, this scheduler is $(k+1+\epsilon))$-speed $O(\frac{1}{\epsilon^2} \max\{\mathrm{OPT}, \ln(n)\})$-competitive for any $\epsilon > 0$ and $k \geq 0$. It reduces to admit-first when $k = 0$.

4. Section 10.4 provides a lower bound showing that the competitive ratio of work-stealing is $\Omega(\lg n)$ — that is, if all jobs are tiny with work $o(\lg n)$, then work stealing cannot be scalable due to the randomization involved. This shows that our upper bound is close to being tight.

5. We implemented admit-first and steal-k-first in Thread Building Block (TBB) and compare their performance with a simulated optimal scheduler on realistic and synthetic workloads. Evaluation results shows that a work stealing scheduler (especially steal-k-first) have comparable performance to the optimal scheduler (Section 10.5).

6. Section 10.6 considers the case where jobs have weights and show a non-clairvoyant algorithm Biggest-Weight-First (BWF) is $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^2})$-competitive for any $\epsilon > 0$, which is the best positive result that can be shown in the online setting for the weighted case.

10.1 Preliminaries

In the online scheduling problem of multiple jobs, n jobs arrive over time and are scheduled on m identical processors. Each job J_i has an arrival (release) time r_i, which is the first time an online scheduler is aware of the job. Each job could have a weight w_i — this weight is known to the scheduler when the job arrives and may not be correlated to the work of the job. For the unweighted setting, $w_i = 1$ for all jobs.

When analyzing the performance of a scheduling algorithm, we denote c_i as the completion time of job J_i in the algorithm's schedule. We denote $F_i = c_i - r_i$ as the flow time of job J_i in the algorithm's schedule. The goal of the scheduler is to minimize $\max_{i \in [n]} w_i F_i$.

A dynamic multithreaded job J_i can be represented as a **Directed-Acyclic-Graph (DAG)** G_i. Each node (task) v in G_i has an associated processing time p_v and the node must be processed sequentially on a processor for p_v time to be completed. A node in G_i cannot be executed until all of its predecessors in G_i have been executed. We say that a node is *ready* if all of its predecessors have been processed. Multiple ready nodes for the same job can be scheduled simultaneously. A job is completed only once all of the nodes in its DAG have been completely processed. Note that we do not assume that the scheduler knows the DAG in advance; in fact, the DAG unfolds dynamically as the job executes.

Symbol	Definition
c_i	completion time of job J_i in schedule
r_i	arrival time of job J_i
F_i	flowtime of job J_i
P_i	the critical path length of J_i
m	the number of processors
w_i	the weight of job J_i
OPT	optimal schedule and also optimal objective

Table 10.1: Symbols and Definitions in Chapter 10

Dynamic multithreaded jobs can be characterized by two important parameters. The **critical-path length** P_i of job J_i is defined to be the execution time of J_i if it were scheduled continuously on an infinite number of processors. Alternatively, it is defined to be the length of the longest path in G_i, where each node v in the longest path contributes p_v to the length of the path. Note that P_i is a lower bound on the execution time of J_i for any scheduler. The **Work** W_i of job J_i is the execution time on 1 processor; or alternatively, the sum of the processing times of all the nodes in the DAG. We summarize some of the notations in Table 10.1.

The following proposition states that any time a scheduler is working on all the ready nodes for some job J_j, the scheduler must be decreasing the remaining critical path of J_j.

Proposition 80 *If during each time step during a time interval $[t', t]$, a scheduler of speed s is always scheduling all available nodes for a job J_j, then the scheduler reduces the critical path length of J_i by $s(t - t')$.*

Difficulties of Analyzing Algorithms in the DAG Model. For scheduling sequential jobs, previous analyses for maximum flow time follows by showing that the algorithm A under consideration cannot fall behind for more than mp_{max} where p_{max} is the maximum processing time of a job. The is because either the algorithm A has less than m jobs and so there can be at most mp_{max} total work for unsatisfied jobs in its queue; or it has more than

m jobs, but then the algorithm A cannot fall behind much since it will always be using all m processors.

Unfortunately, this argument is no longer straightforward for DAGs. To see this, consider unweighted flow time and let OPT denote the objective of the optimal solution. Note that in the sequential setting, $\text{OPT} \geq p_{max}$. In contrast, in the DAG model, some jobs could have work $\Theta(m\text{OPT})$. Now, even if the algorithm A under consideration has fewer than m unsatisfied jobs, it can have a total work of $\Theta(m^2\text{OPT})$ in its queue — allowing it to fall very far behind OPT. Thus, it is difficult to directly bound the amount of work an algorithm falls behind the OPT just as a function of the number of jobs in the algorithm's schedule. To prove that the algorithm does not fall far behind the optimal solution, a necessary condition for an algorithm to be competitive for maximum flow time, we instead identify times where the algorithm must not be too far behind the optimal solution and then show that the algorithm must not fall behind much further following those times.

On an additional note, most previous work on scheduling parallel jobs is in the arbitrary speedup curves model and uses a potential function argument (see [88] for a tutorial). Unfortunately, there are currently no known potential function proofs for maximum flow time unlike for other objectives.

10.2 Unweighted Maximum Flow Time using FIFO

In this section our goal is to prove the following theorem stating that the algorithm First-In-First-Out (FIFO) is $(1+\epsilon)$-speed $O(\frac{1}{\epsilon})$-competitive for minimizing the maximum unweighted flow time for any $0 < \epsilon < 1$.

Theorem 81 *First-in-First-Out (FIFO) is $(1+\epsilon)$ speed $O(\frac{1}{\epsilon})$ competitive for minimizing the maximum unweighted flow time for any $\epsilon > 0$.*

FIFO is defined as follows. At any time t, FIFO orders the jobs in increasing order by their arrival time, breaking ties arbitrarily. The algorithm then assigns all of the ready nodes for the first job to unique processors, then recursively does the same for the next job in the list. This continues until all processors have been assigned some node or there are no more ready nodes available. The algorithm may have a choice on which ready nodes of a job to schedule if the job has more ready nodes than the number of processors that have not been assigned to a node when the job is considered. In this case, we assume the scheduler chooses an arbitrary set of ready nodes from the job.

The rest of this section is devoted to proving Theorem 81. We assume for the remaining of this section that FIFO is given $(1+\epsilon)$-speed for some constant $0 < \epsilon < 1$ and we will show that FIFO is $\frac{3}{\epsilon}$ competitive. To show this, assume for the sake of contradiction that FIFO is more than $\frac{3}{\epsilon}$-competitive and we consider the instance for which FIFO does not achieve a competitive ratio of $\frac{3}{\epsilon}$. Let J_i be the job with the maximum flow F_i at this instance, so OPT $< \frac{\epsilon}{3}F_i$ by assumption. Since no jobs that arrive later than J_i has any effect on how or when J_i is scheduled due to FIFO's scheduling policy, in our instance J_i is the last job to arrive.

We begin by showing that during the time interval job J_i is alive in FIFO's schedule, the processors must be busy for most of the interval. We define one **time step** as the time period for a s-speed processor to execute one unit of work. In other words, in one time step m processors with speed s can finish m work of jobs. Note that on processors with different speeds, the length of a time step will be different. Hence, the number of time steps on a s-speed processor in T time is sT, while it is T on unit speed processor.

Lemma 82 *During the interval $[r_i, c_i]$ in FIFO's schedule, there can be no more than $\frac{\epsilon}{3}F_i$ time steps where not all m processors are busy working on jobs.*

Proof. For the sake of contradiction, suppose there is at least $\frac{\epsilon}{3}F_i$ time steps during $[r_i, c_i]$ where not all processors are busy. Consider FIFO's scheduling policy. Anytime during $[r_i, c_i]$

where FIFO is not processing nodes on every processor, FIFO must be scheduling all of the ready nodes of J_i. Due to this, at these times FIFO is working on the critical path length of J_i by Proposition 80. Let this path length be P_i, then we have $P_i \geq \frac{\epsilon}{3} F_i$.

Also note that OPT cannot finish a job in less time than its critical-path length, this leads to $\text{OPT} \geq P_i \geq \frac{\epsilon}{3} F_i$, so the competitive ratio is $\frac{F_i}{\text{OPT}} \leq \frac{3}{\epsilon}$, a contradiction. □

The previous lemma shows that for most of the time steps in $[r_i, c_i]$ FIFO has m processors busy working. In the next lemma, we show that the work done by FIFO during $[r_i, c_i]$ is concentrated on jobs which did not arrive before $r_i - F_i$. We define **processor idling steps** to be the aggregate number of time steps per processor where the processor is not working on any job. Hence, during one time step that not all m processors are busy working, there can be at most m processor idling steps in total.

Lemma 83 *During $[r_i, c_i]$, FIFO does more than $m(1 + \frac{\epsilon}{3})F_i$ work on jobs which arrived after $r_i - F_i$.*

Proof. Since J_i is the job with the maximum flow time F_i, all previous jobs must have had less flow time than F_i. Therefore, all jobs which received any processing during $[r_i, c_i]$ must have arrived at earliest $r_i - F_i$.

Now to complete the lemma we calculate the total work done during $[r_i, c_i]$. From Lemma 82 the number of processor idling steps is at most $m\frac{\epsilon}{3}F_i$ during $[r_i, c_i]$. Since the processors have speed $1 + \epsilon$, the total work that is done during $[r_i, c_i]$ is at least

$$m(1+\epsilon)F_i - m\frac{\epsilon}{3}F_i > m(1+\frac{\epsilon}{3})F_i$$

which completes the lemma. □

Using the previous lemmas we can complete the proof.

Proof of Theorem 81. We consider the work of the optimal schedule. OPT achieves a flow time of OPT $< \frac{\epsilon F_i}{3}$ from the assumption that FIFO does not achieve a competitive ratio of $\frac{3}{\epsilon}$.

Consider all the jobs which arrived during $[r_i - F_i, r_i]$, OPT must finish every such job before $r_i + \frac{\epsilon}{3} F_i$. During the interval $[r_i - F_i, r_i + \frac{\epsilon}{3} F_i]$ the optimal schedule can do at most $m(1 + \frac{\epsilon}{3}) F_i$ work with 1 speed.

However from Lemma 83 the jobs which arrive after $r_i - F_i$ have more than $m(1 + \frac{\epsilon}{3}) F_i$ work. Hence the optimal schedule cannot possibly finish all jobs by time $r_i + \frac{\epsilon}{3} F_i$, a contradiction. □

10.3 Unweighted Maximum Flow Time using Work Stealing

In this section, we consider a variation of work stealing, called steal-k-first work stealing scheduler, the formal definition of which will be discussed later. Our goal is to show the following theorem.

Theorem 84 *The maximum unweighted flow time of the steal-k-first work stealing scheduler with $(k + 1 + (k + 2)\epsilon)$ speed is $O(\frac{1}{\epsilon^2} \max\{\text{OPT}, \ln n\})$ for any $k \geq 0$ and any $0 < \epsilon < \frac{1}{k+2}$ with high probability.*

By scaling the constant ϵ using the constant k in Theorem 84, we can trivially get the Corollary below.

Corollary 85 *The maximum unweighted flow time of the steal-k-first work stealing scheduler with $(k + 1 + \epsilon)$ speed is $O(\frac{1}{\epsilon^2} \max\{\text{OPT}, \ln n\})$ for any $k \geq 0$ and any $0 < \epsilon < 1$ with high probability.*

For a version of steal-k-first, namely admit-first, where the constant $k = 0$, we have the following result.

Corollary 86 *The maximum unweighted flow time of the admit-first work stealing scheduler with* $(1+\epsilon)$ *speed is* $O(\frac{1}{\epsilon^2}\max\{\mathrm{OPT}, \ln n\})$ *for any* $0 < \epsilon < 1$ *with high probability. In particular, if* $\mathrm{OPT} \geq \ln n$*, then the scheduler is* $(1+\epsilon)$*-speed* $O(\frac{1}{\epsilon^2})$*-competitive with high probability.*

Work Stealing for a Single Job.

The work stealing scheduler [33] is a distributed scheduler for scheduling a single parallel program. It dispatches work dynamically, rather than statically. Scheduling is done in a distributed manner, which results in scalability and lower overhead. Specifically, the runtime system creates a worker thread for every available core. Each worker maintains a local double-ended queue, called *deque*. When a worker generates new work (enables a ready node from the job's DAG), it pushes the work onto the bottom of its deque. When a worker finishes its current node, it pops a ready node from the bottom of its deque. If the local deque is empty, the worker becomes a *thief* and randomly picks a *victim* worker and tries to steal work from the top of the victim's deque. We assume that it takes a unit time step to steal work between workers.

Note that most of the time, workers work off their own queues and don't need to communicate with each other. Hence, this randomized work-stealing strategy is very effective in practice and the amount of scheduling and synchronization overhead is small. Therefore, work stealing is the default strategy used for executing parallel DAGs in many parallel runtime systems such as Cilk Plus, TBB, X10, and PPL [33, 41, 89, 134, 147].

Theoretically, however, because of this randomized and distributed characteristic, work stealing is not a strictly greedy strategy. However, work stealing provides strong probabilistic guarantees of linear speedup for a single job. Researchers have shown that work-stealing is provably efficient with high-probability when scheduling a single job [33].

Work Stealing for Multiple Jobs.

Though work stealing scheduler is designed for scheduling a single job, we can extend it to scheduling multiple jobs in a straightforward way. In addition to the deque of each worker, a global FIFO queue is dedicated for the arrival and admission of new jobs. When a new job is released, it is inserted into the tail of the global queue. A worker will *admit* a job by popping if from the head of the global queue in a FIFO order.

Under different admission strategies, workers could choose to steal work or admit a job in different manners. In this chapter, we consider a strategy, namely **steal-k-first work stealing**, in which each worker always tries to randomly steal first and only tries to admit a new job if there are k consecutive unsuccessful steal attempts for some constant $k \geq 0$. Now we analyze the theoretical performance of steal-k-first and we present its empirical performance in Section 10.5.

Intuitions for Proving Theorem 84.

As discussed in Section 11.1, to prove steal-k-first is competitive for maximum flow time, we need to show that it does not fall far behind the optimal schedule. We assume for the sake of contradiction that it does at some time t. Then we go back in time to a point t' where the algorithm was not far behind the optimal solution. This time is carefully defined by recursively going back in time ensuring (1) that the algorithm is always doing a significant amount of work during $[t', t]$ and (2) that we can actually find t' while ensuring (1) is true. After finding such a time t', we are able to show that while the algorithm may fall far behind the optimal schedule during $[t', t]$ due to not taking advantage of the parallelizability of jobs, it eventually is able to do a large amount of work. With faster speed, it catches up and this allows us to bound its performance. Before formally proving the theorem, we first show that steal-k-first does not idle much when there are jobs to execute.

Idling Steps in Steal-k-First.

We define **processor idling steps** to be the aggregate number of time steps per processor where the processor is not working on a job (and is stealing instead). WLOG, we assume that each steal attempt takes 1 time step. To bound the idling time in steal-k-first's schedule, we first state a theorem from [33], which provides the bound on the time that a work stealing scheduler spends on stealing during the execution of a *single* job.

Lemma 87 *During the time interval $[e_i, c_i]$ where e_i and c_i are the execution start time and completion time of a job J_i respectively, the expected number of steal attempts is bounded by $32mP_i$ where P_i is the critical-path length and m is the number of processors. Moreover, for any $\delta > 0$, the number of steal attempts is bounded by $64mP_i + 16\ln(1/\delta)$ with probability at least $1 - \delta$.*

Although the Lemma above only applies to the case of a single job, by extending it we can obtain a useful lemma for the case with n jobs. In the following lemma, let e_i denote the time that job J_i is admitted from the global queue by a processor. This is the first time the job is started.

Lemma 88 *For a time interval that lies between the start time e_i and completion time c_i of a job J_i, with probability at least $1 - \frac{1}{n}$, the number of processor idling steps is bounded by $64mP_i + 32\ln(n) \leq 64m\mathrm{OPT} + 32\ln(n)$.*

Proof. Consider Lemma 87 and choose $\delta = \frac{1}{n^2}$. The probability of any job J_i exceeding the idling time bound $64mP_i + 16\ln(n^2) = 64mP_i + 32\ln(n)$ during $[e_i, c_i]$ is $\frac{1}{n^2}$. This idling time bound holds for any time interval that is between $[e_i, c_i]$. Union bounding over all n jobs and subtracting from 1 yields the probability in the lemma. □

W will use the following lemma to later bound the idling time due to steal attempts between the arrival time r_i and the start time e_i of a job J_i.

Lemma 89 *Under steal-k-first with a speed of $s = k + 1 + (k + 2)\epsilon$, the number of idling steps during a time interval $[t', t]$ that is contained in $[r_i, e_i]$, the time between when a job arrives and is removed from the global queue, is at most $\frac{k}{k+1}(k+1+(k+2)\epsilon)m(t-t')+km$.*

Proof. Every time a processor has more than k steal attempts, the processor will do one unit of work. Thus for any time interval of length $(t-t')$ there can be at most a $s\frac{k}{k+1}(t-t')+k$ steal attempts per processor. The lemma follows by aggregating over all processors. □

Now we can bound the amount of work steal-k-first does. We say that a job J_i **spans** a time interval $[t_a, t_{a-1}]$, if its release time $r_i \leq t_a$ and its completion time $c_i \geq t_{a-1}$.

Lemma 90 *If a job spans a time interval $[t_a, t_{a-1}]$, then steal-k-first work stealing with speed $k + 1 + (k + 2)\epsilon$ does at least $\frac{k+1+(k+2)\epsilon}{k+1}m(t_b - t_a) - (km + 64m\text{OPT} + 32\ln(n))$ work with probability at least $1 - \frac{1}{n}$.*

Proof. By definition, $[t_a, t_{a-1}]$ lies between $[r_i, c_i]$. From Lemma 89, the number of idling steps during $[t_a, e_i]$ is at most $\frac{k}{k+1}(k + 1 + (k + 2)\epsilon)m(e_i - t_a) + km \leq \frac{k}{k+1}(k + 1 + (k + 2)\epsilon)m(t_{a-1} - t_a) + km$. From Lemma 88, the number of idling steps during $[e_i, t_b]$ is at most $64m\text{OPT} + 32\ln(n)$ with probability at least $1 - \frac{1}{n}$.

Thus, during $[t_a, t_{a-1}]$ the amount that work steal-k-first with speed $k+1+(k+2)\epsilon$ does is at least

$$\begin{aligned}
&(k+1+(k+2)\epsilon)\, m(t_{a-1}-t_a) - (64m\text{OPT} + 32\ln(n)) \\
&-\left(\frac{k}{k+1}(k+1+(k+2)\epsilon)\, m(t_{a-1}-t_a) + km\right) \\
=&\frac{k+1+(k+2)\epsilon}{k+1} m(t_b - t_a) - (km + 64m\text{OPT} + 32\ln(n))
\end{aligned}$$

with probability at least $1 - \frac{1}{n}$. □

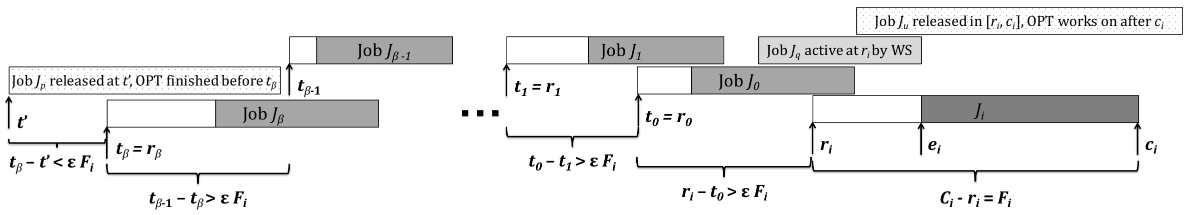

Job J_u released in $[r_i, c_i]$, OPT works on after c_i
Job $J_{\beta-1}$
Job J_1
Job J_q active at r_i by WS
Job J_p released at t', OPT finished before t_β
$t_{\beta-1}$
$t_1 = r_1$
Job J_0
t'
Job J_β
$t_0 = r_0$
J_i
r_i
e_i
c_i
$t_\beta = r_\beta$
$t_\beta - t' < \varepsilon F_i$
$t_0 - t_1 > \varepsilon F_i$
$t_{\beta-1} - t_\beta > \varepsilon F_i$
$r_i - t_0 > \varepsilon F_i$
$C_i - r_i = F_i$

Figure 10.1: An example execution trace of work-stealing identifying jobs' release and completion times.

Time Intervals in Steal-k-First.

The main challenge in analyzing steal-k-first is that it is difficult to show that the remaining processing time of jobs in its queue is comparable to that of OPT's queue. Rather than directly bounding the differences between the two queues as done in previous section, we will construct a set of time intervals where steal-k-first must be busy most of the time. Using the assumption that steal-k-first has resource augmentation, we will draw a contradiction by showing that steal-k-first has completed a large amount of work which is even more than the total amount of work available during a time interval.

From here on, our goal is to show that the steal-k-first with $(k + 1 + (k + 2)\epsilon)$-speed achieves a maximum flow time of $O(\frac{1}{\epsilon^2} \max\{\text{OPT}, \ln(n)\})$ with high probability. To simplify the proof, we rewrite the objective to eliminate the max and show instead that steal-k-first achieves a maximum flow of $\frac{65}{\epsilon^2}(\text{OPT} + \ln(n) + k)$, $k \geq 0$ is a constant and $0 < \epsilon < \frac{1}{k+2}$.

Let J_i be the job in steal-k-first's schedule with the maximum flow time F_i. Recall that r_i and c_i are the arrival and completion time of J_i, respectively. To show contradiction, we assume that $F_i \geq \frac{65}{\epsilon^2}(\text{OPT} + \ln(n) + k)$.

We will recursively define a set of time intervals

$$T = \{[t', t_\beta], [t_\beta, t_{\beta-1}], [t_{\beta-1}, t_{\beta-2}] \ldots [t_1, t_0], [t_0, r_i], [r_i, c_i]\}$$

where $t' \leq t_\beta \leq t_{\beta-1} \leq \ldots \leq t_1 \leq t_0 \leq r_i \leq c_i$. To illustrate the time intervals, Figure 10.1 shows an example execution trace of steal-k-first.

Let t_0 be the arrival time of the earliest arriving job among the jobs that are not finished by steal-k-first right before time r_i. For instance, in Figure 10.1 there are two jobs (job J_0 and job J_q) that are active right before r_i. Among then, job J_0 has the earliest arrival time, so t_0 is defined using it. If there are no jobs right before r_i, let $t_0 = r_i$. Now we define further intervals recursively. Given the time t_{a-1}, we want to define t_a. If $t_{a-1} - t_a \leq \epsilon F_i$, then we are done defining intervals; otherwise, we define t_a as the arrival time of the earliest arriving job among those that are not finished by steal-k-first right before time t_{a-1}. We say that a certain job J_a **defines** an interval $[t_a, t_{a-1}]$, if it is the earliest arriving job unsatisfied by steal-k-first right before t_{a-1} and t_a is its arrival time.

Note that this process of defining intervals will always terminate. The procedure terminates when $t_{a-1} - t_a \leq \epsilon F_i$, which must happen if one goes back to the first time a job arrives. We let β denote the maximum value that a takes during this inductive definition. Hence, $[t_\beta, t_{\beta-1}]$ is the earliest time interval defined in this scheme. Moreover, the arrival time t' of the earliest arriving job among those that are unfinished right before time t_β satisfies $t' - t_\beta \leq \epsilon F_i$. As in Figure 10.1, interval $[t', t_\beta]$ is the first such interval that has length less than ϵF_i.

Work Done by Steal-k-First.

We intend to show that steal-k-first does a lot of work during the interval $[t_\beta, c_i]$. In fact, we will show that if the assumption of $F_i \geq \frac{65}{\epsilon^2}(\text{OPT} + \ln(n) + k)$ is true, then steal-k-first would have done more work than the total work of all jobs that are active during $[t_\beta, c_i]$, which is not possible and leads to a contradiction.

To do so, we partition $[t_\beta, c_i]$ into two sets of time intervals, specifically $S_1 = \{[t_a, t_{a-1}], \forall\, 0 < a \leq \beta\} \cup \{[t_0, r_i]\}$ during $[t_\beta, r_i]$, and $S_2 = \{[r_i, c_i]\}$. We first show that for intervals in S_1, steal-k-first does more work than OPT.

Lemma 91 *For any time interval $[t_a, t_{a-1}] \in S_1$ during $[t_\beta, r_i]$, with probability at least $1 - \frac{1}{n}$ the work that steal-k-first does is more than $m(t_{a-1} - t_a)$, which is as much as* OPT *does.*

Proof. By definition, there is a job J_a which defines this time interval. Specifically, this job spans the time interval. According to Lemma 90, we know that with probability $1 - \frac{1}{n}$ the amount of work steal-k-first does is at least $\frac{k+1+(k+2)\epsilon}{k+1} m(t_{a-1} - t_a) - (km + 64m\text{OPT} + 32\ln(n))$.

Recall that by assumption that $F_i > \frac{65}{\epsilon^2}(\text{OPT} + \ln(n) + k)$ and by definition that $(t_{a-1} - t_a) > \epsilon F_i$, we have

$$\begin{aligned}(t_{a-1} - t_a) > \epsilon F_i > \frac{65}{\epsilon}(\text{OPT} + \ln(n) + k) \\ = \frac{1}{\epsilon}\frac{1}{m}(65km + 65m\text{OPT} + 65m\ln(n)) \\ > \frac{1}{\epsilon}\frac{1}{m}(km + 64m\text{OPT} + 32\ln(n))\end{aligned}$$

Hence, $(km + 64m\text{OPT} + 32\ln(n)) < \epsilon m(t_{a-1} - t_a)$

Thus during any time interval $[t_a, t_{a-1}]$ in S_1, the work done by steal-k-first (with speed $k + 1 + (k+2)\epsilon$) on jobs is at least:

$$\begin{aligned}&\frac{k+1+(k+2)\epsilon}{k+1} m(t_{a-1} - t_a) - (km + 64m\text{OPT} + 32\ln(n)) \\ &> m(t_{a-1} - t_a) + \frac{(k+2)\epsilon}{k+1} m(t_{a-1} - t_a) - \epsilon m(t_{a-1} - t_a) \\ &= m(t_{a-1} - t_a) + \frac{\epsilon}{k+1} m(t_{a-1} - t_a)\end{aligned}$$

Clearly OPT with only 1 speed can only do at most $m(t_{a-1} - t_a)$ work during this time interval. □

We now show that for S_2, steal-k-first does a lot of work too.

Lemma 92 *During $[r_i, c_i] \in S_2$, the amount of work that steal-k-first does on jobs is more than $mF_i + \epsilon m F_i + m\text{OPT}$ with probability $1 - \frac{1}{n}$.*

Proof. Consider the work that steal-k-first does during $[r_i, c_i]$. By definition this interval has a length of F_i and we know that J_i spans this interval. Directly applying Lemma 90, we derive that with probability $1 - \frac{1}{n}$ the amount of work done by steal-k-first during $[r_i, c_i]$ is at least

$$\begin{aligned}
&\frac{k+1+(k+2)\epsilon}{k+1} mF_i - (km + 64m\mathrm{OPT} + 32\ln(n)) \\
=& mF_i + \epsilon mF_i + \frac{\epsilon}{k+1} mF_i - (km + 64m\mathrm{OPT} + 32\ln(n))
\end{aligned}$$

By definition, $0 < \epsilon < \frac{1}{k+2}$, so $\frac{1}{k+1}\frac{1}{\epsilon} > 1$. Also recall that by assumption that $F_i > \frac{65}{\epsilon^2}(\mathrm{OPT} + \ln(n) + k)$, we have

$$\begin{aligned}
\frac{\epsilon}{k+1} mF_i >& \frac{m}{k+1}\frac{65}{\epsilon}(\mathrm{OPT} + \ln(n) + k) \\
>& 65m(\mathrm{OPT} + \ln(n) + k) \\
>& (km + 64m\mathrm{OPT} + 32\ln(n)) + m\mathrm{OPT}
\end{aligned}$$

Hence, $\frac{\epsilon}{k+1} mF_i - (km + 64m\mathrm{OPT} + 32\ln(n)) > m\mathrm{OPT}$. Therefore, the amount of work done by steal-k-first during $[r_i, c_i]$ is more than $mF_i + \epsilon mF_i + m\mathrm{OPT}$ with probability $1 - \frac{1}{n}$.

□

We need one more critical argument to complete the analysis. The reason we defined these time intervals inductively is to identify the jobs that are active under steal-k-first during $[t_\beta, c_i]$. The total volume of these jobs is bounded by the work that OPT can finish. However, just showing that steal-k-first does more work than OPT during $[t_\beta, c_i]$ will not suffice, as OPT could have done part of this work either before t_β or after c_i. As shown in Figure 10.1, the two jobs (job J_p and job J_u) in dotted shade are executed by steal-k-first during $[t_\beta, c_i]$, while OPT finished job J_p before t_β and started working on job J_u after c_i. The next lemma bounds the maximum amount of work that are available for steal-k-first to work on during $[t_\beta, c_i]$.

Lemma 93 *For jobs that are active under steal-k-first during $[t_\beta, c_i]$, their total amount of work is at most $m(r_i - t_\beta) + m(\epsilon F_i + \mathrm{OPT} + F_i)$.*

Proof. By definition, $[t_\beta, c_i]$ consists of time intervals of S_1 during $[t_\beta, r_i]$ and time interval of $S_2 = \{[r_i, c_i]\}$. Also recall that the length of interval $[r_i, c_i]$ is F_i. Hence, the total length of $[t_\beta, c_i]$ is $(r_i - t_\beta) + F_i$.

Moreover, by definition of t_β, the earliest arriving job that is unsatisfied by steal-k-first just before time t_β must have arrived no earlier than time $t_\beta - \epsilon F_i$. Thus, the jobs that are active under steal-k-first during $[t_\beta, c_i]$ all arrived during $[t_\beta - \epsilon F_i, c_i]$.

Further, all these jobs have an optimal maximum flow time no more than OPT under the optimal scheduler. Therefore, OPT must be able to complete all of them by time $c_i + \mathrm{OPT}$. Knowing that OPT can only work on these jobs during $[t_\beta - \epsilon F_i, c_i + \mathrm{OPT}]$, the total amount of work of those jobs can have volume at most $m(r_i - t_\beta) + m(\epsilon F_i + \mathrm{OPT} + F_i)$. □

Finally, we are ready to complete the proof.

Proof of Theorem 84. To prove the theorem, we consider the jobs that are active under steal-k-first during $[t_\beta, c_i]$. By Lemma 93, we know that the total amount of work of these jobs, denoted as X, is bounded: $X \le m(r_i - t_\beta) + m(\epsilon F_i + \mathrm{OPT} + F_i)$. Note that these jobs are the only ones available for steal-k-first to work on during $[t_\beta, c_i]$. Therefore, during $[t_\beta, c_i]$ steal-k-first cannot do more than X work even with speedup.

On the other hand, consider the minimum amount of work that steal-k-first must have done during $[t_\beta, c_i]$, denoted as Y, assuming that $F_i > \frac{65}{\epsilon^2}(\mathrm{OPT} + \ln(n) + k)$ is true. We will see that $Y > X$, which leads to a contradiction.

From Lemma 91, we know that during $[t_\beta, r_i]$ the amount of work steal-k-first does is more than

$$m\left(\sum_{0<a\le\beta}(t_{a-1} - t_a) + (r_i - t_0)\right) = m(r_i - t_\beta)$$

From Lemma 92, we know that during $[r_i, c_i]$, steal-k-first does more than $mF_i + \epsilon m F_i + m\mathrm{OPT}$ work. Thus, for interval $[t_\beta, c_i]$, we get $Y > m(r_i - t_\beta) + mF_i + \epsilon m F_i + m\mathrm{OPT}$.

Now we compare X and Y:

$$\begin{aligned} Y - X >& m(r_i - t_\beta) + mF_i + \epsilon m F_i + m\mathrm{OPT} \\ & - m(r_i - t_\beta) - m(\epsilon F_i + \mathrm{OPT} + F_i) > 0 \end{aligned}$$

Hence, $Y > X$. In other words, if the assumption of F_i is true, during $[r_i, c_i]$ steal-k-first must have done more work than the total available work, which gives a contradiction.

Thus, we obtain the theorem. □

Discussion about Steal-k-First.

Note that for steal-k-first work stealing with $k = 0$, instead of steal first, this scheduler will in fact admit all jobs from the global queue first. We denote this special case as **admit-first**. From Theorem 84, we know that the theoretical performance of steal-k-first is better with smaller constant k. Hence, admit-first has the best theoretical performance and is $O(\frac{1}{\epsilon^2})$-competitive with high probability with $1 + \epsilon$ speed, as it guarantees that a job's execution is not delayed by unnecessary random stealing.

However, as shown in Section 10.5 steal-k-first for a relatively large k performs better than admit-first empirically. Intuitively, if there is any job available for stealing, then in expectation m consecutive random steal attempts would be able to find the stealable work. Thus, for $k \geq m$, steal-k-first better approximates FIFO, which we know works well.

In contrast, in admit-first jobs could run sequentially when there are more than m unfinished jobs. During these times, jobs at the end of the global queue take long time to be admitted and they further take longer time to finish sequential execution in the worst case. Hence, this could increase the maximum flow time of the system.

Moreover, steal-k-first requires a speed of more than $(k+1)$ theoretically to be competitive, mainly due to the worst case scenario where each job has a unit time of work but takes k stealing steps to admit. However, in practice, jobs have much larger work and the constant k steal attempts for admitting a job is negligible in practice.

10.4 Work Stealing Lower Bound for Maximum Flow Time

In this section we give a lower bound for the work stealing algorithm. We show that in the online setting, the scheduler when given any constant speed is $\Omega(\log n)$ competitive. This shows that our upper bound analysis of the algorithm is effectively tight.

Lemma 94 *Work stealing is $\Omega(\log n)$-competitive for maximum flow time in the online setting when given any constant resource augmentation.*

Proof. Let n be an input parameter and let the number of machines be $m = \log n$. Let s be a constant specifying the resource augmentation given to work stealing. The schedule consists of n jobs, which are identical. A job consists of one task which is the predecessor of $m/10$ independent tasks. Note that the total work of the job is $m/10+1$ and can be competed by a 1 speed scheduler scheduler in 2 time steps. A single job is released at multiples of $2m$ starting at time 0. Note that even if a job is executed sequentially, it will complete in only $m/10+1$ time steps. Thus, these jobs do not have overlapping times where multiple jobs are alive in any non-idling schedule.

Now fix any job and consider the probability that the job executes completely sequentially by a work stealing scheduler. This occurs if every steal attempt fails to find the processor holding the tasks for the job. In a single time step, the probability that $m-1$ processors do not successfully steal is $(1-\frac{1}{m-1})^{m-1} \geq \frac{1}{2e}$ for sufficiently large m. The probability that all processors fail to steal for $m/10$ time steps is greater than $(\frac{1}{2e})^{m/10}$.

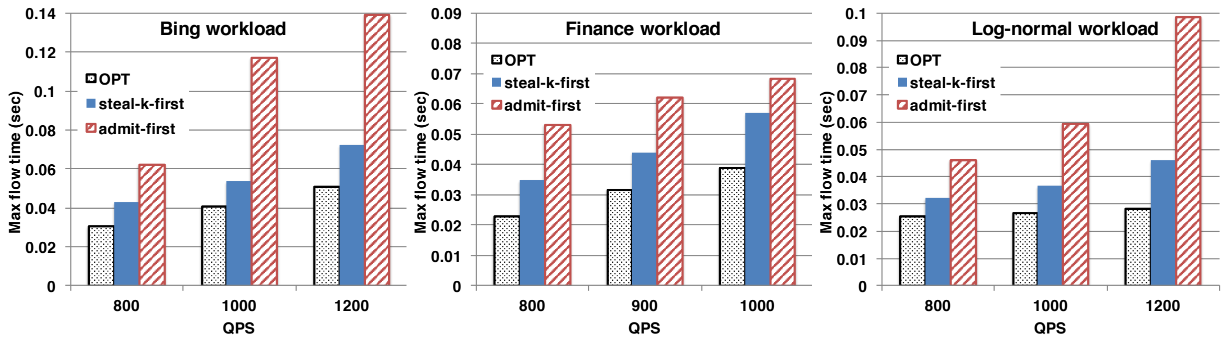
Now consider the expected number of jobs which execute sequentially by work stealing. There are $n = 2^m$ jobs released. The expected number of jobs to execute sequentially is $2^m(\frac{1}{2e})^{m/10} \geq 1$. Thus, the expected maximum flow time of work stealing with s speed is $\frac{m/10+1}{s} = \frac{\log n}{s}$. Knowing that the optimal solution has maximum flow time 2 and $s = O(1)$, the lemma follows. □

10.5 Experimental Results for Unweighted Maximum Flow Time

In this section we present the experimental results using realistic and synthetic workloads to compare the performance of OPT and two work stealing strategies: (1) *Admit-first* where workers preferentially admit jobs from the global queue and only steal if the queue is empty, and (2) *Steal-k-first* where workers preferentially steal and only admit a new job if k steal attempts fail (we use $k = 16$). Our experiments indicate that steal-k-first performs better and is almost comparable to an optimal scheduler.

Setup: We conduct experiments on a server with dual eight-core Intel Xeon 2.4Ghz processors with 64GB main memory. The server runs Linux version 3.13.0, with processor throttling, sleeping, and hyper-threading disabled. The work-stealing algorithms are implemented in Intel Thread Building Block (TBB) [134] version 4.3, a well-engineered popular work-stealing runtime library. We extended TBB to schedule multiple jobs arriving online by adding a global FIFO queue for admitting jobs and we implement both admit-first and steal-k-first.

Since we do not know the optimal scheduler, we approximate it using a simulated scheduler by reducing a parallel scheduling problem to a sequential scheduling problem on a single processor. In particular, for this lower bound, we assume that there is no preemption overhead and that each job can get linear speedup (fully parallelizable). Thus, we can execute



(a) Bing workload (b) Finance workload (c) Log-normal workload

Figure 10.2: Experimental results comparing the maximum flow time running on three work distributions with three different load settings and scheduled using simulated OPT, steal-k-first, and admit-first (from left to right). Note that the scale of the y-axis for the figures differ. From all different settings, OPT has the smallest max flow time, while admit-first has the largest max flow time.

each job one at a time assuming it is a sequential job with execution time equal to its W/m where W is its total work. We then run all jobs using FIFO which is optimal in this setting. When jobs are fully parallelizable, this reduces the problem to the case where there is only one machine. In this setting, it is well known that FIFO is optimal for maximum flow time [50]. Thus, this scheduler has the performance for maximum flow time that is at least as good as any feasible scheduler, including the optimal schedule.

Workloads: We evaluate different strategies on work distributions from two real-world applications shown in Figure 8.1 and additional synthetic workloads with log-normal distribution. Henceforth we shall refer to workload generated from the three distributions as the *Bing workload*, the *finance workload* and the *log-normal workload*, respectively. For each distribution, we select a set of queries-per-second, QPS, to generate workloads with low ($\sim$ 50%), medium ($\sim$ 60%), and high ($\sim$ 70%) machine utilization respectively, and the inter-arrival time between jobs is generated by a Poisson process with a mean equal to $1/QPS$. Each job contains CPU-intensive computation and is parallelized using parallel for loops. $100,000$ jobs are used to obtain a single point in the experiments.

Figure 10.2 shows the experimental results comparing simulated OPT, steal-k-first and admit-first under three different work distributions and three different load settings (i.e., query-per-second). The experiments indicate that, even though our results on OPT are lower bounds on maximum flow time, steal-k-first performs comparably to OPT — matching our intuition that it is a closer approximation for maximum flow time, as discussed at the end of Section 10.3.

Recall that steal-k-first has worse theoretical performance than admit-first. However, in practice, admit-first generally performs worse in terms of maximum flow time and the performance difference increases as load increases (for instance, for Bing and log-normal workloads with high utilization, admit first has twice the maximum flow). This matches our intuition — at higher loads, admit-first executes jobs more or less sequentially, while steal-first provides parallelism to already admitted jobs before admitting new jobs. Therefore steal-first is closer to FIFO in that it tries to execute jobs that arrived earlier with more parallelism. Therefore, in practice, steal-first is likely to be a good implementation for schedulers that want to minimize maximum flow time without incurring the large overheads of FIFO.

10.6 Maximum Weighted Flow Time using Biggest-Weight-First

In this section, we consider the case where jobs have different importances. The goal of the scheduler is to minimize $\max_{i \in [n]} w_i F_i$, where w_i is the weight of job J_i — this weight is known to the scheduler when the job arrives and may not be correlated to the work of the job. For the unweighted setting, $w_i = 1$ for all jobs.

We present the algorithm Biggest-Weight-First (BWF), which is a scalable algorithm for minimizing the maximum weighted flow time. BWF is defined as follows, which is similar to

FIFO except that priority is given to the jobs with the biggest weight. At any time t, BWF orders the jobs in decreasing order by their weight, breaking ties arbitrarily. The algorithm then assigns all of the ready nodes for the first job to some processor, then recursively does the same for the next job in the list. This continues until all processors have been assigned some node or there are no more ready nodes available. Like FIFO, BWF may have a choice on which ready nodes of a job to schedule if the job has more ready nodes than the number of processors which have not been assigned to a node when the job is considered. In this case, we assume the scheduler chooses an arbitrary set of ready nodes.

Theorem 95 *Biggest-Weight-First (BWF) is* $(1+\epsilon)$ *speed* $O(\frac{1}{\epsilon^2})$ *competitive for minimizing the maximum weighted flow for any* $\epsilon > 0$.

The reminder of this section is devoted to proving Theorem 95. For the rest of this section, we assume that BWF is given $(1+3\epsilon)$-speed for some constant $0 < \epsilon < \frac{1}{3}$ and we will show that BWF is $\frac{3}{\epsilon^2}$ competitive. Fix any sequence of jobs and let OPT denote the optimal schedule on this instance as well as the optimal maximum weighted flow time. Let F_a^* be the flow time of a job J_a in OPT.

Let J_i be the job in BWF's schedule with the maximum weighted flow time w_iF_i. For the sake of contradiction, we assume that $w_iF_i > \frac{3}{\epsilon^2}\text{OPT}$. Since $\text{OPT} = w_iF_i^*$, $F_i > \frac{3}{\epsilon^2}F_i^*$, where F_i^* is the flow time of J_i in OPT. By comparing the weight w_i of job J_i, any jobs with weight at least w_i are referred as **heavy** jobs, and any jobs with less weight than w_i are referred as **light** jobs.

Time Intervals in BWF

Similar to the time intervals specified in Section 10.3, we will inductively define a set of time intervals

$$T = \{[t', t_\beta], [t_\beta, t_{\beta-1}], [t_{\beta-1}, t_{\beta-2}] \ldots [t_1, t_0], [t_0, r_i], [r_i, c_i]\}$$

where $t' \leq t_\beta \leq t_{\beta-1} \leq \ldots \leq t_1 \leq t_0 \leq r_i \leq c_i$.

Recall that r_i and c_i are the arrival and completion time of J_i, respectively. Consider the *heavy* jobs that BWF is scheduling right before r_i. Let t_0 be the arrival time of the *earliest* arriving one of those jobs. If there are no heavy jobs right before r_i, let $t_0 = r_i$. Now we define further intervals recursively. Given the times t_{a-1}, we want to define t_a. If $t_{a-1} - t_a \leq \epsilon F_i$, then we are done defining time intervals; otherwise, we define t_a to be the arrival time of the earliest arriving heavy job J_a that are unsatisfied under BWF right before time t_{a-1}. Again if there are no heavy jobs unsatisfied by BWF just before time t_{a-1} then let $t_a = t_{a-1}$. We let β denote the maximum value that a takes during this inductive definition. Hence, $[t_\beta, t_{\beta-1}]$ is the earliest time interval defined in this scheme.

Note that this process of defining intervals is almost the same as in Section 10.3. The only difference is that the job J_a, which defines the interval $[t_a, t_{a-1}]$, is the earliest unfinished *heavy* job under *BWF*. We only consider heavy jobs, because under BWF only heavy jobs can preempt job J_i and other heavy jobs and any light jobs can only execute when all the available nodes of all the active heavy jobs are already executing by some processors. Thus, when analyzing the flow time of J_i and other heavy jobs, we can ignore the remaining light jobs, since light jobs cannot in interfere the execution of heavy ones. Hence, the processor idling steps in the remaining of this section is refering to the time steps where a processor is not working on nodes corresponding to heavy jobs.

We begin the proof by showing that during all time intervals between $[t_\beta, r_i]$, BWF is using most time steps to process nodes for heavy jobs.

Lemma 96 *During any interval $[t_a, t_{a-1}]$ where $a \leq k$, the number of processor idling steps (where a processor is not working on nodes corresponding to heavy jobs) is at most $m\frac{\epsilon^2}{3}F_i$.*

Proof. For the sake of contradiction, assume that this is not true. Then consider the job that defines $[t_a, t_{a-1}]$ and let this job be J_a. By definition this heavy job arrived at t_a and is still being processed at time t_{a-1}. From BWF's scheduling policy, every time step during $[t_a, t_{a-1}]$, where some processors find no nodes from heavy jobs to work on, all ready nodes of J_a are being scheduled. Hence the processors are decreasing the remaining critical path of J_a at these times by Proposition 80. Since the job is not finished until at t_{a-1}, this job must have a critical-path length P_a longer than $P_a > t_{a-1} - t_a > \frac{\epsilon^2}{3} F_i$. Also since J_a is a heavy job and $w_a \geq w_i$ and by assumption $w_i F_i > \frac{3}{\epsilon^2}\mathrm{OPT}$, its weighted flow time is at least

$$w_a(t_{a-1} - t_a) > w_a \frac{\epsilon^2}{3} F_i \geq w_i \frac{\epsilon^2}{3} F_i > \frac{\epsilon^2}{3}\frac{3}{\epsilon^2}\mathrm{OPT} \geq \mathrm{OPT}$$

However, OPT cannot complete a job faster than its critical-path length, so $F_a^* \geq P_a$. Further, J_a's weighted flow time under OPTis at most the maximum weighted flow time OPT. We have

$$\mathrm{OPT} \geq w_a F_a^* \geq w_a P_a > w_a(t_{a-1} - t_a) > \mathrm{OPT}$$

This gives a contradiction. □

Using the previous lemma, we bound the aggregate amount of work done by BWF on heavy jobs during $[t_\beta, r_i]$.

Lemma 97 *During $[t_\beta, r_i]$, the amount of work that BWF does on heavy jobs is more than $m(1 + 2\epsilon)(r_i - t_\beta)$.*

Proof. From Lemma 96, we know that there are only $m\frac{\epsilon^2}{3}F_i$ processor idling steps where a processor is not working on nodes corresponding to heavy jobs during any time interval $[t_a, t_{a-1}]$. In addition, we know $t_{a-1} - t_a > \epsilon F_i$, since $a \leq \beta$. Hence, the work done by BWF (with $1 + 3\epsilon$ speed) on heavy jobs during $[t_a, t_{a-1}]$ is at least:

$$\begin{aligned}
&m(1 + 3\epsilon)(t_{a-1} - t_a) - m\frac{\epsilon^2}{3}F_i \\
>&m(1 + 3\epsilon)(t_{a-1} - t_a) - m\frac{\epsilon}{3}(t_{a-1} - t_a) \\
>&m(1 + 2\epsilon)(t_{a-1} - t_a)
\end{aligned}$$

Summing over all the intervals yields the lemma. □

Similarly, we can bound the amount of work done by BWF on heavy jobs during $[r_i, c_i]$.

Lemma 98 *During $[r_i, c_i]$, the amount of work that BWF does on heavy jobs is more than $m(1 + 2\epsilon)F_i$.*

Proof. By assumption, $F^* < \frac{\epsilon^2}{3}F_i$. Since OPT cannot finish a job in less time than its critical-path length, job J_i has $P_i \le F_i^* < \frac{\epsilon^2}{3}F_i$. From Proposition 80, we can derive that the number of processor idling steps where a processor is not working on heavy jobs is at most mP_i. Hence, the amount of work done by BWF during $[r_i, c_i]$ is at least $m(1+3\epsilon)F_i - mP_i > m(1 + 3\epsilon)F_i - m\frac{\epsilon^2}{3}F_i > m(1 + 2\epsilon)F_i$, since $\epsilon < \frac{1}{3}$. □

Now we bounds the maximum amount of work that are available for BWF to work on during $[t_\beta, c_i]$.

Lemma 99 *For jobs that are active under BWF during $[t_\beta, c_i]$, their total amount of work is at most $m(r_i - t_\beta) + m(1 + \epsilon + \frac{\epsilon^2}{3})F_i$.*

Proof. By definition, the total length of $[t_\beta, c_i]$ is $(r_i - t_\beta) + F_i$. Moreover, by definition of t_β, the earliest arriving heavy job that is unsatisfied BWF just before time t_β must have arrived no earlier than time $t_\beta - \epsilon F_i$. Thus, the heavy jobs that are active under BWF during $[t_\beta, c_i]$ all arrived during $[t_\beta - \epsilon F_i, c_i]$.

Furthermore, all these heavy jobs have an optimal maximum weighted flow time no more than OPT under the optimal scheduler, i.e., OPT $\ge F_a^* w_a$. By definition of a heavy job $w_a \ge w_i$ and by assumption $w_iF_i > \frac{3}{\epsilon^2}$OPT, we have $w_aF_i \ge w_iF_i > \frac{3}{\epsilon^2}\text{OPT} > \frac{3}{\epsilon^2}F_a^* w_a$. Thus, the flow time F_a^* of these heavy jobs under the optimal schedule is $F_a^* < \frac{\epsilon^2}{3}F_i$.

Therefore, OPT must be able to complete all of them by time $c_i + \frac{\epsilon^2}{3}F_i$. Knowing that OPT can only work on these jobs during $[t_\beta - \epsilon F_i, c_i + \frac{\epsilon^2}{3}F_i]$, the total amount of work of those jobs can have volume at most $m(r_i - t_\beta + F_i) + m(\epsilon F_i + \frac{\epsilon^2}{3}F_i) = m(r_i - t_\beta) + m(1 + \epsilon + \frac{\epsilon^2}{3})F_i$.

□

Finally, we are ready to complete the proof.

Proof of Theorem 95. To prove the theorem, we consider the heavy jobs that are active under BWF during $[t_\beta, c_i]$. By Lemma 99, we know that the total amount of work of these jobs, denoted as X, is bounded: $X \le m(r_i - t_\beta) + m(1 + \epsilon + \frac{\epsilon^2}{3})F_i$. Note that these jobs are the only ones available for BWF to work on, so during $[t_\beta, c_i]$ BWF cannot do more than X work even with speedup.

On the other hand, consider the minimum amount of work that BWF must have done during $[t_\beta, c_i]$, denoted as Y, assuming that $w_i F_i > \frac{3}{\epsilon^2}\mathrm{OPT}$ is true. We will see that $Y > X$, which leads to a contradiction.

From Lemma 97, we know that during $[t_\beta, r_i]$ the amount of work BWF does is more than $m(1 + 2\epsilon)(r_i - t_\beta)$ From Lemma 98, we know that during $[r_i, c_i]$, BWF does more than $m(1 + 2\epsilon)F_i$ work. Thus, for interval $[t_\beta, c_i]$, we get $Y > m(r_i - t_\beta) + m(1 + 2\epsilon)F_i$.

Now we compare X and Y and note that $\epsilon < \frac{1}{3}$:

$$Y - X > m(r_i - t_\beta) + m(1 + 2\epsilon)F_i - m(r_i - t_\beta) - m(1 + \epsilon + \frac{\epsilon^2}{3})F_i > 0$$

Hence, if the assumption of $w_i F_i$ is true, then during $[r_i, c_i]$ BWF must have done more work than the total available work, which gives a contradiction. By scaling ϵ, we obtain the theorem. □

Remarks: The result of weighted flow time can be applied to maximum stretch. In the sequential setting, weighted flow time captures maximum stretch by setting the weight to be the inverse of the processing time. In other words, the flow of a job is scaled by the inverse of its processing time in the stretch objective for sequential jobs. However, stretch is not well-defined for DAG jobs. In particular, should the flow time be scaled by the inverse of the total work or the critical path length? Although there are two natural interpretations of the stretch in the DAG setting, both of them can be still captured by weighted flow time. Since BWF is $(1+\epsilon)$-speed $O(\frac{1}{\epsilon^2})$-competitive for maximum weighted flow time and there are strong lower bounds without speed augmentation, so this result can be viewed as essentially the best positive theoretical result for maximum stretch.

Chapter 11

Scheduling Parallel Jobs Online to Minimize Average Flow Time

Today, most hardware vendors have moved to manufacturing multicore machines and there is increasing interest in enabling parallelism. Many languages and libraries, such as Cilk, Cilk Plus [89], Intel's Threading Building Blocks [134], OpenMP [125], X10 [147], have been designed to allow programmers to write parallel programs. In addition, there has been extensive research on provably good and practically efficient schedulers for these programs in the case where a single job (program) is executing on the parallel machine [31–33].

In most of this research, the parallel job is modeled as a directed acyclic graph (DAG) where each node of the DAG is a sequential sequence of instructions and each edge is a dependence between nodes. A node is *ready* to be executed when all its predecessors have been executed. For the case of a single job, schedulers such as a list scheduler [77] and a work-stealing scheduler [33] are known to be asymptotically optimal with respect to the makespan of the job.

In this chapter, we are interested in multiprogrammed environments where multiple DAG jobs (say n jobs) share a single parallel machine with m processors, jobs arrive and leave online, and the scheduling objective is to provide a quality of service guarantee. This problem has been extensively studied for *sequential* (non-parallizable) jobs and several quality of service metrics have been considered. The *flow time* of a job i is the amount of time job i waits after it arrives until it is completed under some schedule. The most widely considered objectives are minimizing the average flow time (or equivalently, the total flow time), the maximum flow time and more generally, the ℓ_k-norms of flow time. In this chapter, we focus on the average flow time objective, which optimizes the average quality of service; this is the most popular objective considered in online scheduling theory.

As stated above, this problem has been widely considered for sequential jobs where each job can be scheduled on only one processor at a time. In this case, when all m processors are identical it is known that any algorithm is $\Omega(\min\{\log P, \log n/m\})$-competitive where P is the ratio of the largest to smallest processing time of the jobs [109]. In the face of these strong lower bounds, previous work has considered a *resource augmentation* analysis where the algorithm is given extra speed over the adversary [93]. With resource augmentation, several algorithms are known to be $(1+\epsilon)$-speed $O(f(\epsilon))$-competitive for average flow time where $\epsilon > 0$ and some function f which depends only on ϵ [49]. Such an algorithm is known as *scalable* and is the best positive result one can show for problems that have strong lower bounds on the competitive ratio. In particular, several greedy algorithms are known to be scalable including Shortest-Remaining-Processing-Time (SRPT) and Shortest-Job-First (SJF) [27, 40, 74, 149]. Similar results are also known in more general machine environments [4, 45, 86].

Parallel jobs have also been considered in this online multiprogrammed setting; however, the parallelism model most widely considered is the *arbitrary speed-up curve model.* In the speed-up curve model, each job i is associated with a sequence of phases. Phase j for job i is denoted by a tuple $(W_{i,j}, \Gamma_{i,j}(m'))$. The value $W_{i,j}$ denotes the total work of the jth phase of job i. The work for each phase must be processed in sequential order. $\Gamma_{i,j}(m')$ is a function that specifies the processing rate $W_{i,j}$ when given $1 \leq m' \leq m$ processors. It is generally assumed that $\Gamma_{i,j}(m')$ is a non-decreasing sublinear function. The speed-up curve model was introduced by [63] and a scalable algorithm, denoted Latest-Arrival-Processor-Sharing (LAPS) is known for the model [66]. This algorithm and its analysis have been very influential in scheduling theory [12, 46–48, 65, 73, 80, 81].

While the speed-up curve model is a theoretically elegant model, most languages and libraries generate parallel programs that are more accurately modeled using DAGs. The work of [136] consider a hybrid of the DAG model and the speed-up curve setting where each node in the DAG has a speed-up curve. They show a $(2+\epsilon)$-speed $O(\frac{\kappa}{\epsilon})$-competitive algorithm for any $\epsilon > 0$ where κ is the maximum number of independent tasks in a job's DAG. Previous work leaves many open questions. In particular, does there exist online scalable algorithms for average flow time as in the arbitrary speed-up curve setting? Further, is there an algorithm whose competitive ratio does not depend on κ?

Challenges with the DAG model:

- Interestingly, the speed-up curve and the DAG models appear to be incomparable. In particular, for the speed-up curve model, the instantaneous parallelism (the number of processors a job can use effectively at a particular instant) depends only on the phase the job is in, which in turn depends only on how much work of the job has been completed. In contrast, for the DAG model, the instantaneous parallelism depends also on which particular nodes have been processed so far. Since there are many possible ways to do the same amount of work, the instantaneous parallelism at a particular instant depends on the previous schedule. Since the DAG is unknown in advance, it is impossible to compute the best possible schedule that leads to best possible future parallelizability. [12]

- One of the goals of this chapter is to design a greedy algorithm for DAG jobs. Interestingly, this presents unique challenges. In Section 11.3.2, we show a counterintuitive result for the DAG model. We construct an example showing that a greedy scheduling algorithm may actually fall behind in the total aggregate amount of work processed when compared to the same algorithm with less resource augmentation. Note that this can never happen for sequential jobs. This occurs for DAG jobs due to the dependences

[12]The speed-up curve model also cannot be simulated using the DAG model. In the speed-up curve model one could have a speed-up curve of the form $\Gamma(m') = \sqrt{m'}$. In this case, a job is processed at a rate of $\sqrt{m'}$ when given $1 \leq m' \leq m$ processors. In the DAG setting, a job's parallelizability is linear up to the number of nodes ready to be scheduled and thus it is unclear how to simulate this speed-up curve.

— by processing jobs faster, the scheduler later may not be able to efficiently pack the tasks of different jobs on the processors as it did in the slower schedule, due to the DAG structures of jobs. The example shows that standard scheduling techniques are not directly applicable to the DAG model, as typically the faster schedule never falls behind the slower schedule.

- A widely used analysis technique for bounding the total flow time, is the *fractional* flow time technique. Fractional flow time is an alternative objective function for which competitiveness is typically easier to prove. In addition, one can usually easily convert an algorithm that is competitive for fractional flow time to one that is competitive for average flow time by speeding up the algorithm by a small factor. Unfortunately, there are several hurdles for this technique in the DAG setting. In particular, it is not immediately clear how to define the fractional objective and, further, since an algorithm may still fall behind by using extra speed in the DAG setting, it is not obvious how to convert an algorithm that is competitive for fractional flow to one that is competitive for average flow time.

Results: We consider minimizing average flow time in the DAG scheduling model. The most natural algorithm to consider for average flow time in the DAG model is LAPS, since this algorithm is known to work well in the speed-up curve model. However, LAPS is a generalization of Round Robin and [136] showed that in the hybrid model, where jobs consist of a DAG and every node has it own speed-up curve, Round Robin like algorithms must have a competitive ratio that depends on $\log \kappa$ even if they are given any $O(1)$ speed augmentation. We are able to show that this hybrid model is strictly harder than the DAG model and that LAPS is a scalable algorithm in the DAG model.

Theorem 100 *LAPS is* $(1+\epsilon)$*-speed* $O(\frac{1}{\epsilon^3})$*-competitive for minimizing the average flow time in the DAG model.*

The result of LAPS also implies the following bound for Round Robin.

Corollary 101 *Round Robin is $(2+\epsilon)$-speed $O(1)$-competitive for any fixed $\epsilon > 0$ for minimizing the average flow time in the DAG model.*

LAPS is a nonclairvoyant algorithm in the sense that it schedules jobs without knowing the processing time of jobs or nodes until they have been completed. Theoretically, LAPS is a natural algorithm to consider. On the other hand, LAPS is a challenging algorithm to implement. In particular, LAPS requires a set of jobs to receive equal processing time, which is hard to achieve in practice with low overheads. More importantly, LAPS has another disadvantage that it is parameterized. The algorithm effectively splits the processors evenly amongst the ϵ fraction of the latest arriving jobs. This ϵ is the same constant used in the resource augmentation. In practice, it is unclear how to set ϵ. Theoretically, this type of algorithm is known as *existentially* scalable. That is, for each possible speed $(1+\epsilon)$ there exists a constant to input to the algorithm which makes it $O(1)$-competitive for any fixed $\epsilon > 0$. Note that in the speed-up curve model it is an intriguing open question whether an algorithm exists which is *universally* scalable. That is, the algorithm is $O(1)$-competitive given any speed $(1+\epsilon)$ without knowledge of ϵ.

In practice, the most widely used algorithms are simple greedy algorithms. They are easy to implement and features can be added to them to ensure low overhead from preemptions. Unfortunately, it is not clear how to adapt known greedy algorithms to the parallel scheduling environments. None are known to perform well for the speed-up curve settings. In this work, we consider a natural adaptation of Shortest-Job-First (SJF) to the DAG model and show the following theorem.

Theorem 102 *SJF is $(2+\epsilon)$-speed $O(\frac{1}{\epsilon^4})$-competitive for average flow time in the DAG model for any $\epsilon > 0$.*

To prove the theorem, we extend the definition of fractional flow time to the DAG model. As mentioned, it is not obvious how to convert an algorithm that is competitive for fractional flow to one that is competitive for total flow time. We give an analysis of such a conversion to the DAG model, but this is perhaps the most challenging part of the analysis and it is where we lose the factor of 2 speed.

This is the first greedy algorithm shown to perform well for parallel jobs in the online setting. The algorithm is simple and natural and could be used in practice. Unfortunately, we were unable to show it is a scalable algorithm. However, we hope our analysis techniques can be useful to resolving whether there exists universally scalable algorithms for scheduling parallel jobs.

11.1 Preliminaries

In the problem considered, there are n jobs that arrive over time that are to be scheduled on m identical processors. Each job i has an arrival time r_i and is represented as a Directed-Acyclic-Graph (DAG). A node in the DAG is *ready* to execute, if all its predecessors have completed. We assume the scheduler knows the ready nodes for a job at a point in time, but does not know the DAG structure a priori. Any set of ready nodes can be processed at once, but each processor can only execute one node at a time. A DAG job can be represented with two important parameters. The total *work* W_i is the sum of the processing time of the nodes in job i's DAG. The *critical-path length* C_i is the length of the longest path in job i's DAG, where the length of the path is the sum of the processing time of nodes on the path. We now state two straightforward observations regarding work and critical-path length.

Observation 103 *If a job i has all of its n ready nodes being executed by a schedule with speed s on m cores, where $n \leq m$, then the remaining critical-path length of i decreases at a rate of s. In other words, at each time step where not all m processors are executing jobs, all ready nodes of all unfinished jobs are being executed; hence, the remaining critical-path length of each unfinished job reduces by s.*

Observation 104 *Any job i takes at least* $\max\{\frac{W_i}{m}, C_i\}$ *time to complete in any schedule with unite speed, including OPT.*

Throughout the chapter we will use A to specify the algorithm being considered unless otherwise noted. We let $W_i^A(t)$ denote the remaining processing time of all the nodes in job i's DAG at time t in A's schedule. Let $C_i^A(t)$ be the remaining length of the longest path in i's DAG where each node contributes its remaining processing time in job A's schedule at time t. Let $A(t)$ denote the set of jobs which are released and unsatisfied in A's schedule at time t. In the above, we replace A with O to denote the same quantity in some fixed optimal solution. Note that $\int_{t=0}^{\infty} |A(t)|$ is exactly the total flow time, the objective we consider. Finally, let $\overline{W}_i(t) = \min\{W_i - W_i^O(t), W_i^A(t)\}$. We overload notation and let OPT refer to both the optimal solution's schedule and its final objective.

Potential Function Analysis: Throughout this chapter we will utilize the potential function framework, also known as amortized analysis. See [88] for a survey on the technique. For this technique, one defines a potential function $\Phi(t)$ which depends on the state of the algorithm being considered and the optimal solution at time t. Let $G_a(t)$ denote the current cost of the algorithm at time t. This is the total waiting time of all the arrived jobs up to time t if the objective is total flow time. Similarly let $G_o(t)$ denote the current cost of the optimal solution up to time t. We note that $\frac{\mathtt{d}G_a(t)}{\mathtt{dt}}$ is the change in the algorithm's objective at time t and this is equal to the number of unsatisfied jobs in the algorithm's schedule at time t, i.e. $\frac{\mathtt{d}G_a(t)}{\mathtt{dt}} = |A(t)|$. To bound the competitiveness of an algorithm, one shows the following conditions about the potential function.

Boundary condition: Φ is zero before any job is released and Φ is non-negative after all jobs are finished.

Completion condition: Summing over all job completions by the optimal solution and the algorithm, Φ does not increase by more than $\beta \cdot \text{OPT}$ for some $\beta \geq 0$.

Arrival condition: Summing over all job arrivals, Φ does not increase by more than $\alpha \cdot$ OPT for some $\alpha \geq 0$.

Running condition: At any time t when no job arrives or is completed,

$$\frac{\mathtt{d}G_a(t)}{\mathtt{dt}} + \frac{\mathtt{d}\Phi(t)}{\mathtt{dt}} \leq c \cdot \frac{\mathtt{d}G_o(t)}{\mathtt{dt}} \tag{11.1}$$

Integrating these conditions over time one gets that $G_a - \Phi(0) + \Phi(\infty) \leq (\alpha + \beta + c) \cdot \mathrm{OPT}$ by the boundary, arrival and completion conditions. This shows the algorithm is $(\alpha+\beta+c)$-competitive

11.2 Algorithm: LAPS

In this section, we analyze the LAPS scheduling algorithm for the DAG model. LAPS is a generalization of round robin. Round robin essentially splits the processing power evenly among all jobs. In contrast, at each step, LAPS splits the processing power evenly among the ϵ fraction of the jobs which arrived the latest. Note that LAPS is parametrized by the constant ϵ, the same constant used for the resource augmentation.

Specifically, let $A(t)$ denote the set of unsatisfied jobs in LAPS's queue at time t. Let $0 < \epsilon < \frac{1}{10}$ be some fixed constant. Let $A'(t)$ contain the $\epsilon|A(t)|$ jobs from $A(t)$ which arrived the latest. Each job in $A'(t)$ receives $\frac{m}{|A'(t)|}$ processors. Each DAG job in $A'(t)$ then assigns an arbitrary set of $\frac{m}{|A'(t)|}$ ready tasks on the processors it receives. If the job does not have $\frac{m}{|A'(t)|}$ ready tasks, it schedules as many tasks as possible and idles the remaining alloted processors.

We assume that the LAPS is given $1 + 10\epsilon$ resource augmentation. As mentioned in Section 11.1, $W_i^A(t)$ and $C_i^A(t)$ denote the aggregate remaining work and critical-path length, respectively, of job i at time t in the LAPS's schedule. $W_i^O(t)$ is the aggregate remaining work of job i in the optimal schedule at time t. Now we compare LAPS to the optimal schedule. To do this, we define a variable $Z_i(t) := \max\{W^A(t) - W^O(t), 0\}$ for each job i. The variable $Z_i(t)$ is the total amount of work job i has fallen behind in the LAPS's schedule as compared to the optimal schedule at time t. Finally, we define $\texttt{rank}_i(t) = \sum_{j \in A(t), r_j \leq r_i} 1$ of job i to be the number of jobs in $A(t)$ that arrived before job i, including itself. Without loss of generality, we assume each job arrives at a distinct time.

Now we are ready to define our potential function.

$$\Phi(t) = \frac{10}{\epsilon} \sum_{i \in A(t)} \left(\frac{1}{m} \texttt{rank}_i(t) Z_i(t) + \frac{100}{\epsilon^2} C_i^A(t) \right)$$

The following proposition follows directly from the definition of the potential function.

Proposition 105 $\Phi(0) = \Phi(\infty) = 0$.

We begin by showing the increase in the potential function is bounded by OPT over the arrival and completion of all jobs.

Lemma 106 *The potential function never increases due to job completion by the LAPS or optimal schedule.*

Proof. When the optimal schedule completes a job, it has no effect on the potential. When the LAPS completes a job i at time t, a term is removed from the summation. Notice that $Z_i(t) = 0$ and $C_i^A(t) = 0$, since the algorithm has completely processed the job. Thus the removal of this term has no effect on the potential. The only other change is that $\texttt{rank}_j(t)$ decreases by 1 for all jobs $j \in A(t)$ where $r_j > r_i$. However, $Z_j(t)$ is always positive by definition, so this can only decrease the potential. $\square$

Lemma 107 *The potential function increases by at most* $O(\frac{1}{\epsilon^3})\mathrm{OPT}$ *over the arrival of the jobs.*

Proof. When job i arrives at time t, it does not effect the rank of any other job since its arrival is after them. Further, by definition $Z_i(t)$ is 0 when job i arrives, since both

LAPS and OPT cannot have worked on job i yet at the time it arrives. Finally, the value of $C_i^A(t) = C_i$. The increase in the potential will be $\frac{1000}{\epsilon^3}C_i$. By summing over the arrival of all jobs, the total increase is $\frac{1000}{\epsilon^3}\sum_{i\in[n]} C_i$. We know that each job i must wait at least C_i time units to be satisfied in OPT by Observation 122, so this is at most $O(\frac{1}{\epsilon^3})$OPT. $\square$

The remaining lemmas bound the change in the potential due to the processing of jobs by OPT and LAPS. We first consider the change in the potential due to the OPT and LAPS separately. Then we combine both changes and bound the aggregate change to be at most $-10|A(t)| + O(\frac{1}{\epsilon^2})|O(t)|$.

Lemma 108 *At any time t, the potential function increases by at most $\frac{10}{\epsilon}|A(t)|$ due to the processing of jobs by OPT.*

Proof. Notice that the variables $C_i^A(t)$ do not change due to OPT. The only change occurs due to the optimal schedule decreasing $Z_i(t)$ for some jobs i. Let job i' be the job in $A(t)$ which arrived the latest. In the worst case, the optimal schedule uses all m processors to process job i' to decrease $Z_i(t)$ at a rate of m. This is the worst case because the rank of job i' is the largest. The total increase in the change of the potential is then $\frac{10}{\epsilon}\frac{1}{m}\texttt{rank}_{i'}(t)m$. Knowing that $\texttt{rank}_{i'}(t) = |A(t)|$, hence $\frac{10}{\epsilon}\frac{1}{m}\texttt{rank}_{i'}(t)m = \frac{10}{\epsilon}|A(t)|$. $\square$

Lemma 109 *At any time t, the potential function increases by at most $-\frac{10}{\epsilon}(1+\epsilon)|A(t)| + O(\frac{1}{\epsilon^2})|O(t)|$ due the processing of jobs by LAPS.*

Proof. Consider the set $A'(t)$ of jobs LAPS processes at time t. We break the analysis into two cases. In either case we show that the total change in the potential is a most $-\frac{10}{\epsilon}(1+\epsilon)|A(t)| + O(\frac{1}{\epsilon^2})|O(t)|$.

Case 1: At least $\frac{\epsilon}{10}|A'(t)|$ jobs in $A'(t)$ have less than $\frac{m}{|A'(t)|}$ ready nodes at time t. Let $A_c(t)$ be this set of jobs.

Since each of these jobs has less than $\frac{m}{|A'(t)|}$ ready tasks at time t, then LAPS schedules all available tasks for these jobs. Hence, LAPS decreases $C_i^A(t)$ at a rate of $1+10\epsilon$ for each job $i \in A_c(t)$ since LAPS has $1 + 10\epsilon$ resource augmentation. We denote the change in the potential as C_1. Therefore,

$$C_1 = -\frac{1000}{\epsilon^3}(1 + 10\epsilon)|A_c(t)|$$

Note that $|A_c(t)| \geq \frac{\epsilon}{10}|A'(t)|$ and , we have

$$C_1 \leq -\frac{100}{\epsilon^2}(1 + 10\epsilon)|A'(t)| \leq -\frac{100}{\epsilon}(1 + 10\epsilon)|A(t)|$$

Finally, because $|A'(t)| = \epsilon|A(t)|$, we get

$$C_1 \leq -\frac{10}{\epsilon}(1 + \epsilon)|A(t)| + O(\frac{1}{\epsilon^2})|O(t)|$$

Case 2: At least $(1 - \frac{\epsilon}{10})|A'(t)|$ jobs in $A'(t)$ have at least $\frac{m}{|A'(t)|}$ nodes ready at time t. Let $A_w(t)$ be this set of jobs, so $|A_w(t)| \geq (1 - \frac{\epsilon}{10})|A'(t)|$.

In this case, we ignore the decrease in the C variables and focus on the decrease in the Z variables due to the algorithms processing. We further ignore the decrease in the $Z_i(t)$ for jobs in $A_w(t) \cap O(t)$.

Notice that for every job i in $A_w(t) \setminus O(t)$ it is the case that $Z_i(t)$ decreases at a rate of $(1 + 10\epsilon)\frac{m}{|A'(t)|}$. This is because: (1) each of these jobs is given $\frac{m}{|A'(t)|}$ processors; (2) LAPS has $(1 + 10\epsilon)$ resource augmentation; (3) OPT completed job i by time t, if job i is in $A_w(t) \setminus O(t)$. Knowing this, we can bound the total change in the potential due to LAPS.

We will replace $1 + 10\epsilon$ with k in some intermediate steps for ease of notation and we denote the total change in the potential due to LAPS as C_2.

$$\begin{aligned}
C_2 &= -\frac{10}{\epsilon} \sum_{i \in A_w(t) \setminus O(t)} \frac{1}{m} \texttt{rank}_i(t) \frac{(1+10\epsilon)m}{|A'(t)|} \\
&= -\frac{10k}{\epsilon} \sum_{i \in A_w(t) \setminus O(t)} \texttt{rank}_i(t) \frac{1}{|A'(t)|} \\
&\leq -\frac{10k}{\epsilon} \sum_{i \in A_w(t) \setminus O(t)} (1-\epsilon)|A(t)| \frac{1}{|A'(t)|}
\end{aligned}$$

Note that $\texttt{rank}_i(t) \geq (1-\epsilon)|A(t)|$ for $i \in A'(t)$ and $|A'(t)| = \epsilon|A(t)|$, we have

$$C_2 \leq -\frac{10k}{\epsilon^2} \sum_{i \in A_w(t) \setminus O(t)} (1-\epsilon) \leq -\frac{10k}{\epsilon^2} \left(\sum_{i \in A_w(t)} (1-\epsilon) - \sum_{i \in O(t)} 1 \right)$$

We can also derive $|A_w(t)| \geq (1 - \frac{\epsilon}{10})|A'(t)|$. By replacing $|A'(t)|$ with $\epsilon|A(t)|$, we get

$$\begin{aligned}
C_2 &\leq -\frac{10k}{\epsilon^2} \left((1 - \frac{\epsilon}{10}) \sum_{i \in A'(t)} (1-\epsilon) - \sum_{i \in O(t)} 1 \right) \\
&\leq -\frac{10k}{\epsilon} \left((1 - \frac{\epsilon}{10}) \sum_{i \in A(t)} (1-\epsilon) - \frac{1}{\epsilon} \sum_{i \in O(t)} 1 \right)
\end{aligned}$$

Finally, because $\epsilon < 1/10$, we can derive

$$\begin{aligned}
C_2 &\leq -\frac{10}{\epsilon}(1+10\epsilon)(1 - \frac{\epsilon}{10}) \sum_{i \in A(t)} (1-\epsilon) + O(\frac{1}{\epsilon^2})|O(t)| \\
&\leq -\frac{10}{\epsilon}(1+\epsilon)|A(t)| + O(\frac{1}{\epsilon^2})|O(t)|
\end{aligned}$$

In either case the total change in the potential is at most $-\frac{10}{\epsilon}(1+\epsilon)|A(t)| + O(\frac{1}{\epsilon^2})|O(t)|$. □

Lemma 110 *Fix any time t. The total change in the potential is at most $-10|A(t)| + O(\frac{1}{\epsilon^2})|O(t)|$ due the processing of jobs by both algorithms.*

Proof. Now we know from Lemma 108 the change due to OPT processing jobs is at most $\frac{10}{\epsilon}|A(t)|$. Combining the change due to both algorithms in Lemma 108 and 109, we see that the aggregate change in the potential is at most $-\frac{10}{\epsilon}(1+\epsilon)|A(t)| + O(\frac{1}{\epsilon^2})|O(t)| + \frac{10}{\epsilon}|A(t)| \leq -10|A(t)| + O(\frac{1}{\epsilon^2})|O(t)|$. □

Thus, by the potential function framework and combining Lemma 106, 107 and 110 and Proposition 105 we have Theorem 100.

11.3 Algorithm: SJF

In this section we analyze a generalization of SJF to parallel DAG jobs. In this algorithm, the jobs are sorted according to their *original* work and the smallest have the highest priority. The algorithm takes the highest priority job and assigns all of its ready nodes to machines and then recursively considers the next highest priority job. This continues until all machines have a node to execute or there are no more ready nodes. In the event that a job being considered has more ready nodes than machines available, the algorithm choses an arbitrary set of nodes to schedule on the remaining machines. At first glance, this might be counterintuitive, since it doesn't take the critical-path length into consideration at all; one might think that we should give higher priority to jobs with longer critical-path length. However, as the analysis shows, it turns out that prioritizing based on just work provides good bounds.

11.3.1 Analysis of SJF for Fractional Flow Time

We use fractional flow time to do this analysis. In this section, to avoid confusion, we refer to total flow time as *integral flow time* — recall that a job contributes 1 to the objective each time unit the job is alive and unsatisfied. In contrast, in fractional flow time, it contributes the fraction of the *work* which remains for the job; that is, the goal is to minimize $\sum_{t=0}^{\infty} \sum_{i \in A(t)} \frac{W_i^A(t)}{W_i}$. Our analysis is structured as follows: We first compare the fractional flow time of SJF (with resource augmentation) to the integral flow time of the optimal algorithm. We then compare the integral flow time of SJF (with further resource augmentation) to its fractional flow time.

We will utilize a potential function analysis and define the potential functions as follows. Throughout the analysis we will assume without loss of generality that each job arrives at a distinct time and has a unique amount of work.

$$\Phi(t) = \frac{1}{\epsilon} \sum_{j \in A(t)} C_j^A(t) \quad + \frac{1}{\epsilon m} \sum_{j \in A(t)} \left(\frac{\overline{W}_j(t)}{W_j} \sum_{\substack{i \mid i \in A(t) \cup O(t) \\ W_i \leq W_j}} W_i^A(t) - W_i^O(t) \right)$$

Using this potential function, our goal is to show the following theorem.

Theorem 111 *SJF is $(1+\epsilon)$-speed $O(\frac{1}{\epsilon})$-competitive when SJF's fractional flow time is compared against the optimal schedule's integral flow time.*

Note that $\Phi(0) = \Phi(\infty) = 0$, thus the boundary condition is true. We will now show the arrival and completion conditions.

Lemma 112 *The potential function increases by at most $O(\frac{1}{\epsilon}\mathrm{OPT})$ due to the arrival and completion of jobs.*

Proof. First consider the arrival condition. Suppose job j' arrives at a time t', then in the first term a new term is created, $\frac{1}{\epsilon}C_{j'}$. This is less than $\frac{1}{\epsilon}$ multiplied by the amount of time this job must wait to be completed in an optimal schedule because C_i is a lower bound on a job's integral flow time, according to Observation 122. The change of $\Phi(t')$ over all job arrivals in the first term is at most $\frac{1}{\epsilon}\mathrm{OPT}$. Now consider the second term of $\Phi(t')$ when j' just arrives. The quantity $\overline{W}_{j'}(t') = 0$, because OPT has not worked on job j' yet. Though j' is a new term in the outer summation of the second term, this term is 0. Finally, j' may appear as a new term in the inner summation for all jobs $i \in A(t')$ with $W_i > W_{j'}$. However then $W_{j'}^A(t') - W_{j'}^O(t') = 0$ because both algorithm and optimal schedule have yet to work on j'. These are all the possible changes due to the arrival of job j', therefore the arrival condition holds.

Now consider when the optimal schedule completes some job j' at time t'. The only effect on the potential, is that a term may be removed from the inner summation of the second term if j' is no longer in $A(t') \cup O(t')$. This only happens if the job is also not in $A(t')$. If the job is not in $A(t')$ then $W_{j'}^A(t') - W_{j'}^O(t') = 0$ and there is no change to the potential due to the removal of the term.

Now consider when the algorithm completes some job j' at time t'. Because the job has completed, so $C_{j'}^A(t') = 0$ and $\overline{W}_{j'}(t') = 0$. Thus, removing terms from the either the first summation or the outer summation of the second term has no effect on the potential. However we may remove a job from the inner summation of the second term. Again, this only occurs if $j' \notin O(t')$, which means that inner summation is 0. Therefore this does not cause a change in the potential. Overall, there is no change in the potential due to jobs being completed by either the algorithm or the optimal schedule. $\square$

Thus, we have shown the boundary conditions as well as the bounded the non-continuous changes in Φ. It remains to show how the potential changes due to the algorithm and optimal schedule processing jobs. These are the only remaining ways the potential may change. Fix some time t. Our goal is to bound $\frac{d\Phi(t)}{dt}$.

Lemma 113 *The total change in Φ at time t due to the optimal schedule processing jobs is $O(|O(t)|) + \frac{1}{\epsilon}\sum_{i \in A(t)} \frac{W_i^A(t)}{W_i}$.*

Proof. Notice that the only changes that can occur due to the optimal schedule processing some job j is due to the changes in $W_j^O(t)$ and $\overline{W}_j(t)$, both of which are in the second term of $\Phi(t)$. Fix some job j that OPT processes at time t and suppose that OPT uses m'_j processors to process job j. Consider the change in $\Phi(t)$ due to $W_j^O(t)$ decreasing. The change only increases $W_j^A(t) - W_j^O(t)$ in the inner summation only if job i in the outer summation has $W_i \geq W_j$. Each machine in OPT has 1 speed and all work values are distinct, so the change is the following.

$$\frac{1}{\epsilon}\frac{m'_j}{m}\frac{\overline{W}_j(t)}{W_j} + \frac{1}{\epsilon}\frac{m'_j}{m}\sum_{\substack{i\in A(t)\\ W_i > W_j}}\frac{\overline{W}_i(t)}{W_i}$$

The first term is the job j itself and the second is the other jobs effected. Since $\frac{\overline{W}_i(t)}{W_i} \leq \frac{W_i^A(t)}{W_i}$ by definition of $\overline{W}_i(t)$, we have

$$\frac{1}{\epsilon}\frac{m'_j}{m}\sum_{\substack{i\in A(t)\\ W_i \geq W_j}}\frac{\overline{W}_i(t)}{W_i} \leq \frac{1}{\epsilon}\frac{m'_j}{m}\sum_{\substack{i\in A(t)\\ W_i \geq W_j}}\frac{W_i^A(t)}{W_i}$$

Now consider the change induced in $\overline{W}_j(t)$ by OPT's processing. This variable could, in the worst case, increase at a rate of m'_j. This changes all of the inner summation terms where $W_i \leq W_j$. We omit the $-W_i^O(t)$ part of the inner summation, as this part only decreases the potential. The change is then the following.

$$\frac{1}{\epsilon}\frac{m'_j}{m}\frac{W_j^A(t)}{W_j} + \frac{1}{\epsilon}\frac{m'_j}{mW_j}\sum_{\substack{i\in A(t)\\ W_i < W_j}}W_i^A(t)$$

By definition, $\frac{W_j^A(t)}{W_j} \leq 1$. Additionally, in the summation we have $W_i < W_j$. Therefore the overall change from processing job j is:

$$\frac{1}{\epsilon}\frac{m'_j}{m} + \frac{1}{\epsilon}\frac{m'_j}{mW_j}\sum_{\substack{i\in A(t)\\ W_i < W_j}}W_i^A(t) \leq \frac{1}{\epsilon}\frac{m'_j}{m} + \frac{1}{\epsilon}\frac{m'_j}{m}\sum_{\substack{i\in A(t)\\ W_i < W_j}}\frac{W_i^A(t)}{W_i}$$

Let $P^O(t)$ be the set of jobs the optimal schedule processes at time t. Clearly, the optimal schedule can use at most m processors at time t, i.e. $\sum_{j\in P^O(t)} m'_j \leq m$. Knowing this, we have the overall change is

$$\sum_{j \in P^O(t)} \left(\frac{1}{\epsilon} \frac{m'_j}{m} + \frac{1}{\epsilon} \frac{m'_j}{m} \sum_{i \in A(t)} \frac{W_i^A(t)}{W_i} \right) \leq \left(\frac{1}{\epsilon} + \frac{1}{\epsilon} \sum_{i \in A(t)} \frac{W_i^A(t)}{W_i} \right)$$

Finally we know that OPT must have at least one alive job if it processes some job. Thus we charge the $\frac{2}{\epsilon}$ to OPT's increase in its objective. This gives the lemma. □

Now we consider the change in $\Phi(t)$ due to the algorithm processing jobs.

Lemma 114 *The total change in Φ at time t due to the algorithm processing jobs is $O(|O(t)|) - (1+\epsilon)\frac{1}{\epsilon} \sum_{i \in A(t) \setminus O(t)} \frac{W_i^A(t)}{W_i}$.*

Proof. For any job j, we know that either the algorithm is processing jobs $i \in A(t)$ where $W_i \leq W_j$ using all m processors or the algorithm is decreasing the critical-path, $C_j^A(t)$, at a rate of $(1+\epsilon)$. This is because, the algorithm by definition has either has assigned all processors to higher priority jobs or it is scheduling all available ready nodes for job j by Observation 123. Suppose that the algorithm decreases the critical-path of j. If this is the case then, this decreases $C_j^A(t)$ at a rate of $-(1+\epsilon)$. Alternatively, say the algorithm assigned all processors to jobs with higher priority than j. Then it is the case that $\sum_{i \mid i \in A(t) \cup O(t), W_i \leq W_j} W_i^A(t) - W_i^O(t)$ decreases at a rate of $-(1+\epsilon)m$ due to the algorithms processing.

Consider all jobs i in the potential. The decreases in the potential function due to the change in $W_i^A(t)$ and $C_i^A(t)$ over all jobs i the algorithm processes is at least the following,

$$-\frac{(1+\epsilon)}{\epsilon} \sum_{i \in A(t)} \frac{\overline{W}_i(t)}{W_i}$$

Consider the jobs in this summation, if $i \notin O(t)$ it is the case that $\overline{W}_i(t) = W_i^A(t)$. If $i \in O(t)$ then in the worst case $\overline{W}_i(t) = 0$. Nevertheless dropping all $i \in O(t)$ the decrease in the potential still

$$-\frac{(1+\epsilon)}{\epsilon}\sum_{i\in A(t)\setminus O(t)}\frac{W_i^A(t)}{W_i}$$

The only other change that can occur is that the algorithm can cause $\overline{W}_j(t)$ to decrease for jobs j that the algorithm processes. When multiplied by $-W_i^O(t)$ this causes an increase in the potential function. Say that the algorithm processes job j using m'_j processors at time t. Let $P^A(t)$ be the set of jobs the algorithm processes. In the worst case, $\overline{W}_j(t)$ decreases at a rate of $(1+\epsilon)m'_j$ for each job $j \in P^A(t)$. The change is at most,

$$\begin{aligned}
&\frac{(1+\epsilon)}{m\epsilon}\sum_{j\in P^A(t)}\frac{m'_j}{W_j}\sum_{\substack{i\in O(t)\\ W_i\le W_j}} W_i^O(t)\\
\le\ &\frac{1+\epsilon}{m\epsilon}\sum_{j\in P^A(t)} m'_j \sum_{\substack{i\in O(t)\\ W_i\le W_j}} 1 \quad [W_i^O(t)\le W_i\le W_j]\\
\le\ &\frac{(1+\epsilon)}{m\epsilon}\sum_{j\in P^A(t)} m'_j \sum_{i\in O(t)} 1\\
\le\ &\frac{(1+\epsilon)}{\epsilon}\sum_{i\in O(t)} 1 \quad [\textstyle\sum_{j\in P^A(t)} m'_j \le m]\\
=\ &\frac{(1+\epsilon)}{\epsilon}|O(t)|
\end{aligned}$$

Thus, the lemma follows assuming that $0<\epsilon\le 1$ is a constant. □

Now we are ready to show SJF's guarantees for fractional flow time.

Proof of Theorem 111.

The total change in the potential due to the algorithm and optimal schedule processing jobs is the following from Lemmas 114 and 113. Note that we are summing over the terms, some of which are negative due to decreasing the potential.

$$
\begin{aligned}
&O(|O(t)|) + \frac{1}{\epsilon}\sum_{i\in A(t)}\frac{W_i^A(t)}{W_i} - \frac{(1+\epsilon)}{\epsilon}\sum_{i\in A(t)\setminus O(t)}\frac{W_i^A(t)}{W_i}\\
\leq &O(|O(t)|) + \frac{1}{\epsilon}\sum_{i\in A(t)\setminus O(t)}\frac{W_i^A(t)}{W_i} + \frac{1}{\epsilon}\sum_{i\in O(t)}\frac{W_i^A(t)}{W_i} - (1+\epsilon)\frac{1}{\epsilon}\sum_{i\in A(t)\setminus O(t)}\frac{W_i^A(t)}{W_i}\\
\leq &O(|O(t)|) + \frac{1}{\epsilon}\sum_{i\in A(t)\setminus O(t)}\frac{W_i^A(t)}{W_i} + \frac{1}{\epsilon}\sum_{i\in O(t)}1 - (1+\epsilon)\frac{1}{\epsilon}\sum_{i\in A(t)\setminus O(t)}\frac{W_i^A(t)}{W_i}\\
\leq &O(|O(t)|) + \frac{1}{\epsilon}\sum_{i\in A(t)\setminus O(t)}\frac{W_i^A(t)}{W_i} - (1+\epsilon)\frac{1}{\epsilon}\sum_{i\in A(t)\setminus O(t)}\frac{W_i^A(t)}{W_i}\\
\leq &O(|O(t)|) - \sum_{i\in A(t)\setminus O(t)}\frac{W_i^A(t)}{W_i}
\end{aligned}
$$

Consider the second term. We know that

$$
-\sum_{i\in A(t)\setminus O(t)}\frac{W_i^A(t)}{W_i} = -\sum_{i\in A(t)}\frac{W_i^A(t)}{W_i} + \sum_{i\in A(t)\cap O(t)}\frac{W_i^A(t)}{W_i} \leq -\sum_{i\in A(t)}\frac{W_i^A(t)}{W_i} + |O(t)|
$$

Thus, we have proved that the total change in the potential plus the increase in the algorithm's objective, $\sum_{i\in A(t)}\frac{W_i^A(t)}{W_i}$, is bounded by $O(\frac{1}{\epsilon}\mathrm{OPT})$. This completes the proof of the continues change in the potential. The theorem follows by this, Lemma 112 and the potential function framework. □

11.3.2 SJF Falls Behind with Resource Augmentation

Before we show how to convert the fractional flow time of SJF to its integral flow time and how to prove the competitiveness for SJF, we first present the challenges in the proof.

In particular, we show that SJF can fall behind with more resource augmentation. This is surprising because essentially the same scheduling algorithm is used, yet with speed augmentation it is actually possible for the fast schedule to have performed less aggregate work

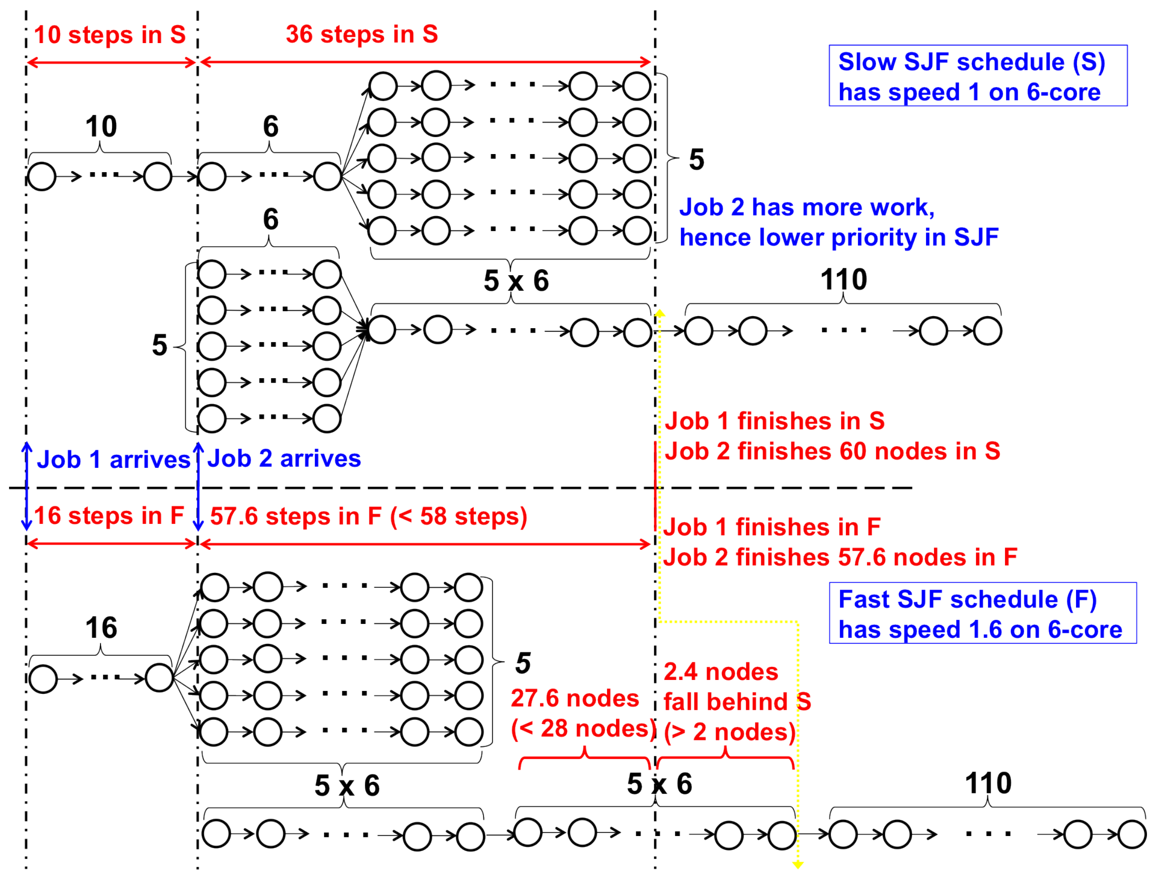

10 steps in S
36 steps in S
Slow SJF schedule (S) has speed 1 on 6-core
10
6
5
Job 2 has more work, hence lower priority in SJF
6
5 x 6
110
5
Job 1 finishes in S
Job 2 finishes 60 nodes in S
Job 1 arrives
Job 2 arrives
16 steps in F
57.6 steps in F (< 58 steps)
Job 1 finishes in F
Job 2 finishes 57.6 nodes in F
Fast SJF schedule (F) has speed 1.6 on 6-core
16
5
27.6 nodes (< 28 nodes)
2.4 nodes fall behind S (> 2 nodes)
5 x 6
5 x 6
110

Figure 11.1: An example schedule of slow and fast SJF on 6 processors

than the slow schedule at some time t. This difficult arises specifically due to the intricacies of the DAG model.

We consider two schedules: one slow schedule S with unit speed and one fast schedule F with speed s for some fixed constant S. We will show that for a given speed augmentation s and m processors, where $1 < s < 2 - \frac{2}{m}$, we can always construct a counterexample showing that the fast schedule F falls behind the slow schedule S using two jobs J_1 and J_2.

First we shall give a concrete example where with 1.6 speed, F does less aggregate work than S does at some time t. Then, the general example for any speed $s < 2 - \frac{2}{m}$ will be given. Intuitively, we show that the structure of J_1 on the fast schedule forces J_2 to be executed entirely sequentially, this severely limits the amount of work that can be done on J_2 by the fast schedule. As both schedules complete J_1, this directly shows that the fast schedule completes less aggregate work.

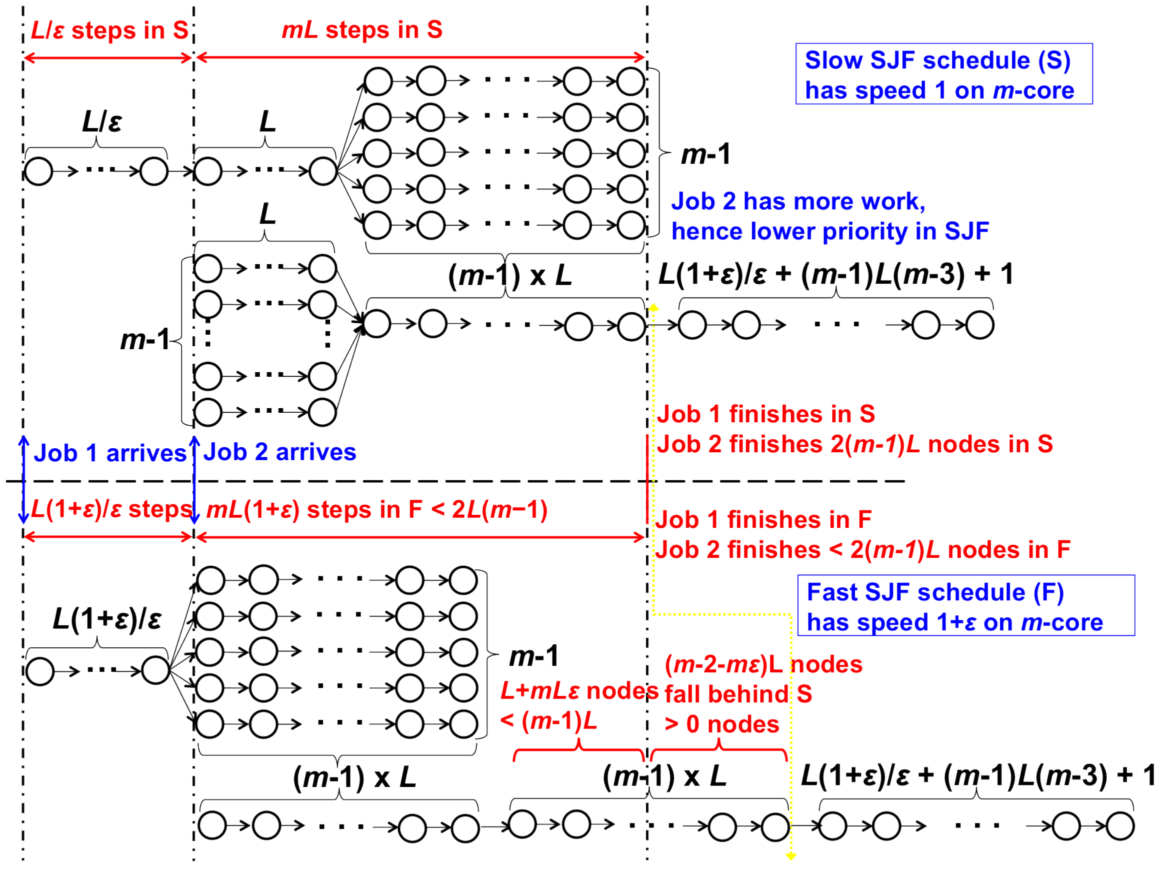
Example for Speed 1.6 on 6 processors

In the concrete example, the fast schedule have 1.6 speed. Consider two jobs J_1 and J_2 as given in the figure. J_1 consists of a sequential chain of nodes of total length 16, followed by 5 chains of nodes all having total length 30 (i.e. a block of width 5 and length 30). Note the construction of the DAG means that at time 10 the fast schedule will have finished the entire chain, while the slow one will still have 6 nodes to do. J_2 arrives at the absolute time of 10 and consists of a block of width 5 with length 6, followed by a long sequential chain of nodes. In this example, the length of this chain is 140. Note that the total work of J_2 is 170, which is more than J_1's total work 166. Thus, J_2 has lower priority under both slow and fast SJF.

The time we consider to contradict the lemma is $t = 46$. By this point, both F and S have finished J_1, therefore it is sufficient to compare the amount of work done on J_2. In the slow schedule for the first 6 steps once J_2 arrives, due to the fact that J_1 can only utilize 1 processor, 30 nodes of J_2 is finished. A further 30 nodes of J_2 finishes for a total of 60 at time t.

The fast schedule is of more interest. With 1.6 speed augmentation, effectively 16 nodes can be finished in the time that the slow schedule requires to finish 10 nodes. Therefore, when J_2 arrives, the fast schedule has already finished the first chain and reached the highly parallel portion of J_1. As J_1 has higher priority than J_2, this forces J_2 to be executed on the only remaining processor sequentially. Hence, due to the length of the block in J_1, the first block (30 nodes) of J_2 is executed completely sequentially. The rest of J_2 is a chain and has to run sequentially due to the structure of the DAG. Therefore, J_2 is performed entirely sequentially.

Now we compare the amount of work of J_2 done by S and F during the time interval $[10, 46]$, which has length 36. Slow schedule with unit speed finishes 60 nodes of J_2. Taking the speed augmentation of 1.6 into account, F can sequentially execute $36 * 1.6 = 57.6$ nodes



L/ε steps in S
mL steps in S
Slow SJF schedule (S) has speed 1 on m-core
L/ε
L
m-1
Job 2 has more work, hence lower priority in SJF
L
(m-1) x L
L(1+ε)/ε + (m-1)L(m-3) + 1
m-1
Job 1 finishes in S
Job 2 finishes 2(m-1)L nodes in S
Job 1 arrives
Job 2 arrives
L(1+ε)/ε steps
mL(1+ε) steps in F < 2L(m−1)
Job 1 finishes in F
Job 2 finishes < 2(m-1)L nodes in F
Fast SJF schedule (F) has speed 1+ε on m-core
L(1+ε)/ε
m-1
(m-2-mε)L nodes fall behind S
L+mLε nodes < (m-1)L
> 0 nodes
(m-1) x L
(m-1) x L
L(1+ε)/ε + (m-1)L(m-3) + 1

Figure 11.2: An example schedule of slow and fast SJF for m processors.

of J_2. Hence, less than 60 nodes of J_2 finishes executing by F. This means that F has fallen behind in comparison to S in terms of aggregate work at time $t = 46$.

General Case for Speed s on m processors

We now show the general case where a speed of $2 - \frac{2}{m}$ is necessary. We assume that the fast schedule is given some speed $s = 1 + \epsilon$ with the restriction that $0 < \epsilon < 1 - \frac{2}{m}$. Similar to the concrete example, we construct the two jobs with J_1 being a chain followed by a block and J_2 being almost the opposite but having larger work and lower priority. The key idea is that for J_1, the fast schedule must reach the highly parallel portion earlier, more precisely, at the release time of J_2. Note that for every node processed by the slow schedule in the initial chain of J_1, the fast schedule processes $1 + \epsilon$ nodes, gaining ϵ nodes over the slow schedule.

Consider Figure 11.2, for similarity to the previous example we introduce a constant L. In the previous example, we had $L = 6$. Let J_1 begin with a chain of length $\frac{L}{\epsilon} + L$, followed by a block of length $(m-1)L$ and parallelism (width) $(m-1)$. J_2 will consist of a block of length L with parallelism $(m-1)$ followed by a long chain of sufficient length such that J_2 has more work and lower priority than J_1. J_2 arrives at exactly time $\frac{L}{\epsilon}$.

The time that will be examined is time $t = \left(\frac{L}{\epsilon} + L\right) + (m-1)L$. Note that at this point both the schedules have finished J_1 and therefore it is sufficient to compare the amount of work done on J_2. In the slow schedule, J_2 arrives when only 1 processors is used to execute J_1, as the highly parallel block has not been reached. Therefore, for the next L time steps a total of $(m-1)L$ nodes of J_2 are finished with parallelism $m-1$. On the following $(m-1)L$ steps, J_1 occupies $m-1$ processors, while J_2 reaches its chain and is processed sequentially. A total of $2L(m-1)$ nodes of J_2 are finished at time t. We also note that a total of mL time steps have passed in the slow schedule between the arrival of J_2 and time t.

From the construction of the initial chain of J_1, the fast schedule completes all $\frac{L}{\epsilon} + L = \frac{L}{\epsilon}(1+\epsilon)$ nodes of the strand by the time $\frac{L}{\epsilon}$ that J_2 arrives. Due to the higher priority of J_1, the parallel block of J_1 take precedence over that of J_2. Note that the parallel block of J_1 has a width of $m-1$, which occupies all but one processor for as long as $(m-1)L$ steps. This forces J_2 to only execute sequentially on the remaining single processor for all its $(m-1)L$ nodes of the parallel block in J_2. When J_1 finally completes and all m processors are free, J_2 reaches its sequential chain. Therefore, J_2 is processed entirely sequentially in the fast schedule.

The amount of time which passes between the arrival of J_2 and t is just mL. Consider the speed augmentation of the fast schedule and recall that $\epsilon < 1 - \frac{2}{m}$. The total number of nodes of J_2, that the fast processor can sequential execute between the arrival of J_2 and t, is $mL(1+\epsilon) < mL(2 - \frac{2}{m}) = 2L(m-1)$. Recall that the slow schedule performed exactly $2L(m-1)$ nodes of J_2 during the same time interval. Therefore, the fast schedule with $1+\epsilon$ speed performs less total aggregate work at time t in comparison to the slow schedule.

Note that this example does not hold when $\epsilon \geq 1$ as the final calculation would result in the fast processor finishing more nodes of J_2.

11.3.3 From Fractional to Integral

We now compare the fractional flow time of SJF to its integral flow time and prove the following lemma. Note that this lemma, combined with Lemma 111 proves Theorem 102.

Lemma 115 *If SJF is s-speed c-competitive for fractional flow time then SJF is $(2+\epsilon)s$-speed $O(\frac{c}{\epsilon^3})$-competitive for the integral flow time for any $0 < \epsilon \leq 1/2$.*

To show the Lemma 115, for the remaining portion of the section we will consider two schedules created by SJF. One schedule has s speed and the other $(2+\epsilon)s$ for some fixed $0 < \epsilon \leq 1/2$ and some constant s. To avoid confusion, we use F to denote the fast schedule and S to denote the slow schedule. Since both schedules are SJF, we assume that the tasks for a job are given the same priority in both algorithms — this priority can be arbitrary.

To begin the proof, we first show that F has always processed as much work as S at any time given a $(2+\epsilon)$ factor more speed. It may seem obvious that a faster schedule should do more work than the slower schedule. However, showing this is not straightforward in the DAG model. In fact, in Section 11.3.2, we have already showed that if the faster schedule has less than a $(2-\frac{2}{m})$ factor speed it will actually fall behind in total aggregate work compared to the slow schedule in some instances. In other words, F does not always process as much of each individual job as S at each point in time. This could cause F to later not achieve as much parallelism as S. Here we will show that F does not fall behind S given a $(2+\epsilon)$ factor more speed.

First, we give some more notations. Let $\mathcal{S}(t)$ ($\mathcal{F}(t)$) denote the queued jobs in S's (F's) schedule at time t, which have been released but not finished. Let $W_i^S(t)$ ($W_i^F(t)$) and $C_i^S(t)$ ($C_i^F(t)$) denote the remaining work and remaining critical-path length, respectively, for job i in S's (F's) schedule at time t. The following lemma states that if we only focus on jobs whose original processing time is less than some value ρ, it must be the case that F did more total work on these jobs than S. This lemma is where we require the 2 speed in the conversion from fractional to integral flow time.

Lemma 116 *At all times t and for all $\rho \geq 0$, it is the case that $\sum_{i \in \mathcal{F}(t), W_i \leq \rho} W_i^F(t) \leq \sum_{i \in \mathcal{S}(t), W_i \leq \rho} W_i^S(t)$.*

Proof. For the sake of contradiction, say the lemma is not true and let t be the first time it is false for some ρ. Then at this time t, there must be some job i where $W_i^S(t) < W_i^F(t)$ and $W_i \leq \rho$.

At release time r_i the lemma still holds, i.e. $\sum_{i \in \mathcal{F}(r_i), W_i \leq \rho} W_i^F(r_i) \leq \sum_{i \in \mathcal{S}(r_i), W_i \leq \rho} W_i^S(r_i)$. Let V be the total volume of original work for jobs of size at most ρ which arrives during $[r_i, t]$. Note that S can do at most $ms(t - r_i)$ work during $[r_i, t]$ with speed s on m processors, we know that at time t

$$\sum_{i \in \mathcal{S}(t), W_i \leq \rho} W_i^S(t) \geq \sum_{i \in \mathcal{S}(r_i), W_i \leq \rho} (W_i^S(r_i) + V - ms(t - r_i))$$

Consider the time interval $[r_i, t]$. Notice that it must be the case that $t - r_i \geq (C_i - C_i^S(t))/s$, since the schedule S has decreased the critical-path of job i by $C_i - C_i^S(t)$ with a speed of s. Further, knowing that both of the schedules execute the nodes of a particular job in the same priority order for either schedule, then $C_i^S(t) \leq C_i^F(t)$. Therefore, we have

$$t - r_i \geq (C_i - C_i^S(t))/s \geq (C_i - C_i^F(t))/s \tag{11.2}$$

Now consider the amount of work done by F during $[r_i, t]$. Note that for at most a $\frac{C_i - C_i^F(t)}{s(2+\epsilon)}$ amount of time during $[r_i, t]$ the schedule F have some processors idling and not executing nodes of jobs with $W_i \leq \rho$. Otherwise, by Observation 123 F would have decreased the critical-path of job i during these non-busy time steps by strictly more than $\frac{C_i - C_i^F(t)}{s(2+\epsilon)} \cdot s(2+\epsilon) = C_i - C_i^F(t)$. Then the remaining critical-path of job i at time t in F would then be less than

$C_i^F(t)$, contradicting the definition of $C_i^F(t)$. Thus, F processes a total volume of at least $(2+\epsilon)ms(t - r_i - \frac{C_i - C_i^F(t)}{s(2+\epsilon)})$ on jobs with original size at most ρ during $[r_i, t]$. Hence the following. (Here have omitted the repeated indices on some sums for brevity, and invoked equation 11.2 for one of the steps).

$$\sum_{i \in \mathcal{F}(t), W_i \le \rho} W_i^F(t) \le \sum_{-} W_i^F(r_i) + V - (2+\epsilon)s(t - r_i - \frac{C_i - C_i^F(t)}{s(2+\epsilon)})$$
$$\le \sum_{-} W_i^F(r_i) + V - (2+\epsilon)s(t - r_i - \frac{t - r_i}{(2+\epsilon)}) = \sum_{-} W_i^F(r_i) + V - (1+\epsilon)s(t - r_i)$$
$$\le \sum_{i \in \mathcal{S}(r_i), W_i \le \rho} W_i^S(r_i) + V - s(t - r_i) = \sum_{i \in \mathcal{S}(t), W_i \le \rho} W_i^S(t)$$

This contradicts the definition of t. □

Let $t_{i,\epsilon}^S$ denote the latest time t in S's schedule where $\frac{W_i^S(t)}{W_i} \ge \epsilon$. For the fractional flow time objective, job i always pays a cost of at least ϵ at each time during $[r_i, t_{i,\epsilon}^S]$ in S's schedule. Let $f_{i,\epsilon}^S = t_{i,\epsilon}^S - r_i$. It must be the case that job i's fractional flow time is greater than $\epsilon f_{i,\epsilon}^S$ in S. For integral flow time we know that a job pays a cost of 1 each time unit it is unsatisfied. Thus, if the integral flow time of job i in F is bounded by $f_{i,\epsilon}^S$ we can charge this job's integral cost in F to the job's fractional cost in S. Also, according to Observation 122, for integral flow time the optimal schedule of speed 1 must make job i wait C_i time steps. Thus, if job i's flow time is bounded by C_i in F then we can charge job i's integral flow time in F directly to the optimal schedule. These two ideas are formalized in the following lemma.

For any schedule A, we let IntCost(A) denote the integral cost of A and FracCost(A) denote the fractional flow time of A. Finally, we let OPT^I denote the optimal schedule for integral flow time.

Lemma 117 *Let $E^F(t)$ be the set of jobs $i \in \mathcal{F}(t)$ such that $t \leq r_i + \frac{10}{\epsilon^2}(\max\{f^S_{i,\epsilon}, C_i\})$. Consider the quantity $\sum_{t=0}^{\infty} |E^F(t)|$, which is the contribution to the total integral flow at time t from jobs in $E^F(t)$. It is the case that $\sum_{t=0}^{\infty} |E^F(t)| \leq O(\frac{1}{\epsilon^3})(\text{FracCost}(S) + \text{IntCost}(\text{OPT}^I))$.*

Proof. **Case 1:** Consider a job i with $f^S_{i,\epsilon} = \max\{f^S_{i,\epsilon}, C_i\}$. In this case, job i can only be in $E^F(t)$ during $[r_i, r_i + \frac{10}{\epsilon^2} f^S_{i,\epsilon}]$. The total integral flow time that job i in F can accumulate during this interval is at most $\frac{10}{\epsilon^2} f^S_{i,\epsilon}$. By definition of $f^S_{i,\epsilon}$, job i's fractional flow in S is at least $\epsilon f^S_{i,\epsilon}$. Hence, the total integral flow time of all jobs in F where $f^S_{i,\epsilon} = \max\{f^S_{i,\epsilon}, C_i\}$ during times where they are in $E^F(t)$ is at most $O(\frac{1}{\epsilon^3})\text{FracCost}(S)$.

Case 2: Consider a job i, with $C_i = \max\{f^S_{i,\epsilon}, C_i\}$. The integral flow time in OPT^I for job i is at least C_i by definition of the critical-path. Thus, we bound the integral flow time of all such jobs in F while they are in $E^F(t)$ by $O(\frac{1}{\epsilon^2})\text{IntCost}(\text{OPT}^I)$. □

Intuitively, we think of the jobs in $E^F(t)$ as jobs which are *early* at time t. Let $L^F(t) = \mathcal{F}(t) \setminus E^F(t)$ be the set of *late* jobs at time t. The remaining portion of the proof focuses on bounding the integral flow time of jobs in F's schedule at times when they are in $L^F(t)$. We will prove that $O(\frac{1}{\epsilon}) \sum_{i \in \mathcal{S}(t)} \frac{W^S_i(t)}{W_i} \geq |L^F(t)|$ at all times t. That is, the total fractional weight of jobs in S is greater than the number of late jobs in L at all times t. Thus, we can charge the integral flow time of jobs in $L^F(t)$ to the fractional flow time of S's schedule. This will complete the proof.

To prove this, we will show the following structural lemma about S and F. Let $\mathcal{S}_{=h}(t)$ ($\mathcal{F}_{=h}(t)$) denote the remaining jobs i in S's (F's) schedule at time t whose original work satisfies $2^{h-1} \leq W_i < 2^h$ for some integer $h \geq 1$. Let $W^S_{=h}(t) = \sum_{i \in \mathcal{S}(t), 2^{h-1} \leq W_i < 2^h} W^S_i(t)$ ($W^F_{=h}(t) = \sum_{i \in \mathcal{F}(t), 2^{h-1} \leq W_i < 2^h} W^F_i(t)$) denote the remaining work in S's (F's) schedule at time t for jobs i whose original work satisfies $2^{h-1} \leq W_i < 2^h$ for some $h \geq 1$. We will say job i is in *class* h, if $2^{h-1} \leq W_i < 2^h$.

Lemma 118 *At all times t and for all $h \geq 1$, $|\mathcal{F}_{=h}(t) \cap L^F(t)| \leq \frac{10}{\epsilon} \frac{1}{2^h} \sum_{h'=1}^{h} W^S_{=h'}(t)$.*

Before we prove this lemma, we show how it can be used to bound the number of jobs in $L^F(t)$ in terms of the fractional weight of jobs in $\mathcal{S}(t)$.

Lemma 119 *At all times t,*

$$O(\frac{1}{\epsilon}) \sum_{i \in \mathcal{S}(t)} \frac{W_i^S(t)}{W_i} \geq |L^F(t)|$$

Proof. Notice that $|L^F(t)| = \sum_{h=1}^{\infty} |\mathcal{F}_{=h}(t) \cap L^F(t)|$. Using Lemma 118 we have the following.

$$\begin{aligned}
|L^F(t)| &= \sum_{h=1}^{\infty} |\mathcal{F}_{=h}(t) \cap L^F(t)| \leq \sum_{h=1}^{\infty} \frac{10}{\epsilon} \sum_{h'=1}^{h} \frac{1}{2^h} W_{=h'}^S(t) \quad \text{[By Lemma 118]} \\
&= \sum_{h=1}^{\infty} \frac{10}{\epsilon} \sum_{h'=1}^{h} (\frac{1}{2^{h'}} W_{=h'}^S(t)) \frac{1}{2^{h-h'}} = \frac{10}{\epsilon} \sum_{h'=1}^{\infty} (\frac{1}{2^{h'}} W_{=h'}^S(t)) \sum_{h=h'}^{\infty} \frac{1}{2^{h-h'}} \\
&\leq \frac{20}{\epsilon} \sum_{h'=1}^{\infty} \frac{1}{2^{h'}} W_{=h'}^S(t) \leq \frac{20}{\epsilon} \sum_{i \in \mathcal{S}(t)} \frac{W_i^S(t)}{W_i} \quad [2^{h'-1} \leq W_i < 2^{h'} \text{ if } i \text{ in class } h']
\end{aligned}$$

□

The previous lemma with Lemma 117 implies Lemma 115. All that remains is to prove Lemma 118.

Proof of Lemma 118.

Assume for the sake of contradiction the lemma is not true. Let t be the earliest time the lemma is false for some class h, i.e. $|\mathcal{F}_{=h}(t) \cap L^F(t)| > \frac{10}{\epsilon} \sum_{i \in \mathcal{S}(t), W_i \leq 2^h} \frac{1}{2^h} W_i^S(t)$.

Let j^* denote the job in $L^F(t)$ which arrived the earliest and j^* is of some class $h' \leq h$. By definition of $L^F(t)$, this implies that S processed at least $(1-\epsilon)W_i$ for each job $i \in L^F(t)$ where $W_i \leq 2^h$ by time t. Since S has m processors of speed s, this means $t - r_{j^*} \geq \frac{1}{sm} \sum_{i \in L^F(t), W_i \leq 2^h} (1-\epsilon) W_i$.

Consider the interval $[r_{j^*}, t]$. We first make several observations about the length of this time interval. We know that $t - r_{j^*} \geq \frac{10}{\epsilon^2} C_{j^*}$ since $j^* \in L^F(t)$. We further know that during

$[r_{j^*}, t]$ there can be at most C_{j^*} time steps where F is not using all m processors to execute nodes for jobs which are in a class at most h. Otherwise job J^* would have finished all its C_{j^*} critical-path length by time t using Observation 123 and thus have been completed by t, a contradiction.

Now our goal is to bound the total work S and F can process for jobs in classes h or less during $[r_{j^*}, t]$. The schedule S can process at most $sm(t - r_{j^*})$ work on jobs of class at most h during $[r_{j^*}, t]$ since it has m machines of speed s. The schedule F processes at least $(2+\epsilon)sm(t - r_{j^*} - C_{j^*})$ work on jobs of class at most h by the observations above. Knowing that $t - r_{j^*} \geq \frac{10}{\epsilon^2}C_{j^*}$, we see that $(2+\epsilon)sm(t - r_{j^*} - C_{j^*}) \geq (2+\epsilon)(1 - \frac{\epsilon^2}{10})sm(t - r_{j^*})$.

We will use these arguments to bound the total volume of work in S at time t to draw a contradiction. Let V denote the total original processing time of jobs which are of class at most h that arrive during $[r_{j^*}, t]$. By Lemma 116, we have $\sum_{i \in \mathcal{F}(r_{j^*}), W_i \leq 2^h} W_i^F(r_{j^*}) \leq \sum_{i \in \mathcal{S}(r_{j^*}), W_i \leq 2^h} W_i^S(r_{j^*})$. Thus,

$$
\begin{aligned}
&\sum_{i \in \mathcal{S}(t), W_i \leq 2^h} W_i^S(t) - \sum_{i \in \mathcal{F}(t), W_i \leq 2^h} W_i^F(t) \\
&\geq \left(\sum_{\substack{i \in \mathcal{S}(r_{j^*}) \\ W_i \leq 2^h}} W_i^S(r_{j^*}) + V - sm(t - r_{j^*}) \right) - \\
&\qquad \left(\sum_{\substack{i \in \mathcal{F}(r_{j^*}) \\ W_i \leq 2^h}} W_i^F(r_{j^*}) + V - (2+\epsilon)(1 - \frac{\epsilon^2}{10})sm(t - r_{j^*}) \right) \\
&\geq (-sm(t - r_{j^*})) - \left(-(2+\epsilon)(1 - \frac{\epsilon^2}{10})sm(t - r_{j^*}) \right) \qquad \text{[Lemma 116]} \\
&\geq \frac{1+\epsilon}{2} sm(t - r_{j^*}) \quad [\epsilon \leq 1/2]
\end{aligned}
$$

This implies that

$$\sum_{i \in \mathcal{S}(t), W_i \leq 2^h} W_i^S(t) \geq \frac{1+\epsilon}{2} sm(t - r_{j^*})$$

We also know that

$$t - r_{j^*} \geq \frac{1}{sm} \sum_{i \in L^F(t), W_i \leq 2^h} (1-\epsilon) W_i$$

With $\epsilon \leq 1/2$ this means that

$$\sum_{i \in \mathcal{S}(t), W_i \leq 2^h} W_i^S(t) \geq \frac{1+\epsilon}{2} \sum_{i \in L^F(t), W_i \leq 2^h} (1-\epsilon) W_i \geq \frac{\epsilon}{4} \sum_{i \in L^F(t), W_i \leq 2^h} W_i$$

Knowing that jobs of class h have size at most 2^h and $\sum_{i \in \mathcal{S}(t), W_i \leq 2^h} W_i^S(t) \geq \frac{\epsilon}{4} \sum_{i \in L^F(t), W_i \leq 2^h} W_i$, we complete the proof:

$$|\mathcal{F}_{=h}(t) \cap L^F(t)| = \sum_{\substack{i \in L^F(t) \\ 2^{h-1} \leq W_i < 2^h}} 1 \leq 2 \sum_{\substack{i \in L^F(t) \\ 2^{h-1} \leq W_i < 2^h}} \frac{W_i}{2^h} \leq \frac{10}{\epsilon} \sum_{i \in \mathcal{S}(t), W_i \leq 2^h} \frac{1}{2^h} W_i^S(t)$$

This contradicts the definition of time t and thus we have proven the lemma. □

Chapter 12

A Distributed Scheduler for Minimizing Average Flow Time of Parallel Jobs

In many application environments such as clouds, grids and shard servers, clients send jobs to be processed on a server. The server has a scheduler that decides which jobs should be processed first. The goal of the scheduling policy is to both use the server resources efficiently and to provide a good quality of service to the client jobs. In this chapter, we consider the commonly used quality metric of *average flow time*. The **flow time (latency)** of a job is the amount of time between the job's arrival at the server and its completion. Minimizing average flow time optimizes the average quality of service provided to all jobs in the system.

For sequential programs that can only utilize one core at a time, minimizing average flow time has been studied extensively [27, 40, 49, 74, 149]. However, most machines now consist of multiple cores and the parallelism of machines is expected to increase. In addition, there is growing interest in parallelizing applications, so that individual jobs may themselves have internal parallelism and can execute on multiple cores on a server at the same time to shorten its processing time.

Programs written in parallel languages can often be abstractly represented as a directed acyclic graph (DAG) where each node of the DAG is a sequence of instructions and each edge is a dependence between nodes. A node is ready to be executed when all its predecessors have been executed — the parallelism is denoted by multiple nodes being ready at the same time. In this model, the problem of scheduling a single parallel job on m processors has been studied extensively in prior work. In earliest work, it was shown that list scheduling is $2 - \frac{2}{m}$ competitive for *makespan* — the total completion time of the single job. This theoretical result has influenced the design of efficient schedulers which are currently used in practice. The work-stealing [33] scheduler is known to be asymptotically optimal with respect to makespan while also being practically efficient in having low overhead [32] and minimizing cache misses [1]. Work-stealing is currently one of the most popular schedulers in practice and is part of the runtime systems of many parallel languages and libraries. The practical success of work-stealing scheduler indicates that theoretical work can be enormously influential in designing schedulers that are good in practice.

Most of the theoretical and practical work on work-stealing has focused on single job scheduling. For designing schedulers for the client-server model described above, we must consider multiprogrammed environments where multiple parallel jobs (say n) share a machine with m processors. In addition, these jobs arrive online where the scheduler is only aware of the job when it arrives. In this setting, the scheduler must know how to allocate resources to the jobs in order to achieve a target goal, which often is optimizing an objective over all of the jobs. Interestingly, there has been little work on scheduling parallelizable DAG jobs in multiprogrammed environments in the online setting.

As mentioned above, the most popular objective considered when jobs arrive over time is **average flow time**: Formally, if the completion (finish) time of job J_i is f_i and its release (arrival) time is r_i, then the flow time of job J_i is $f_i - r_i$. The average flow time objective focuses on minimizing $\sum_i (f_i - r_i)$.

It turns out that average flow time objective is difficult to optimize even for sequential jobs. It is known that any online algorithm is $\Omega(\min\{\log P, \log n/m\})$-competitive where P is the ratio of the largest to smallest processing time of the jobs [109]. Due to this strong lower bound, previous work has focused on using a *resource augmentation* analysis to differentiate between algorithms. In resource augmentation analysis, the algorithm is given faster processors over adversary [93]. Using resource augmentation, several $(1+\epsilon)$-speed $O(f(\epsilon))$-competitive for average flow time are known where $\epsilon > 0$ and the function f which depends only on ϵ. This is regarded as the best positive theoretical result that can be shown for problems which has strong lower bounds on their competitive ratio. The first such result was shown in [49] and several other algorithms have been shown to have similar guarantees for sequential jobs [27, 40, 74, 149].

Chapter 11 introduced the first theoretical results on average flow time for scheduling multiple DAG jobs. This work showed that the algorithm LAPS, an algorithm that generalizes round robin, is $(1 + \epsilon)$-speed $O(\frac{1}{\epsilon^3})$-competitive and a greedy algorithm called smallest work first (SWF) is $(2+\epsilon)$-speed $O(\frac{1}{\epsilon^4})$-competitive. Therefore, LAPS provides the best possible theoretical result for these jobs. However, the problem is far from being well-studied and there remains work to be done. In particular, from a practical standpoint, both algorithms have serious drawbacks that prevent them from being implemented and used in a real system. First, both of the algorithms have an arbitrary and unbounded (in terms of n and m) number of preemptions[13]. Second, both algorithms require a centralized global scheduler that has the knowledge of all alive jobs, processor assignments for each job, and all ready nodes for each job. Finally, SWF is a clairvoyant algorithm that requires the knowledge of the work (processing time of the job on 1 processor) of each job when the job arrives in order to prioritize jobs for scheduling. Although LAPS is non-clairvoyant, it has another disadvantage that it is parameterized. The constant ϵ in the resource augmentation is also used in scheduling to decide processor allocation (evenly distributed amongst the ϵ fraction of the latest arriving jobs). These drawbacks mean that implementations of these algorithms are likely to have significant overheads and are unlikely to be practical in real systems.

In this chapter, our goal is to design a practically efficient scheduler that has strong theoretical guarantees. Our goal is inspired by the work-stealing algorithm that provides theoretically good performance (for a single job) and is efficient in practice since it has very low scheduling overheads. In work stealing, each processor has a local work-queue. When a processor generates more work, it places the work on its own queue. When it needs

[13]Preemption in this context means a processor switches between two jobs that have not yet completed

more work, it takes work from its local queue. If the local queue is empty, the worker randomly picks a victim processor and steals work from its queue. The critical properties of this algorithm that enable it in having low overheads are the following: (1) Distributed decision making: no centralized scheduler distributes work to processors leading to small synchronization overhead. (2) Local work queues: Processors only take work from others when they have no work to do — therefore there is low communication and preemption overhead.

Ideally one could have a similar algorithm for scheduling multiple DAG jobs online. Unfortunately, the work-stealing algorithm does not naturally extend to the multiple job setting. In particular, since the algorithm does not preempt jobs, it can easily be shown to perform poorly for average flow time. For example, consider the case where m big fully parallel jobs arrive. Once these jobs are scheduled and occupy the processors then n small jobs arrive, which are blocked until the assigned big jobs complete. In contrast, the ideal scheduler will schedule the small jobs first, but work-stealing will finish the large jobs before scheduling any of the small jobs, thereby getting a much larger average flow time than the optimal scheduler. Therefore, one must tolerate some preemptions to optimize over average flow. However, our goal is to make most of the decisions in a distributed fashion and have a bounded small number of preemptions.

Results: In this chapter, we introduce a new scheduling algorithm, **Distributed Random Equi-Partition (DREP)**. The algorithm is very simple. When a new job arrives, each processor decides to allocate itself to the new job with probability $1/n_t$, where n_t is the number of incomplete jobs at time t. Then each processor that decides to work on the new job executes a ready node for this job if there is one available. Thus, preemption only happens when a new job arrives. The DREP algorithm is a distributed protocol, has a small number of preemptions, and is *non-clairvoyant*. A non-clairvoyant algorithm requires no information regarding the unrevealed portions of the job's DAG. That is, the algorithm is oblivious to the structure of a job's DAG.

We show the following theorem about the DREP.

Theorem 120 *DREP is $(2+\epsilon)$-speed $O(\frac{1}{\epsilon^3})$-competitive for minimizing average flow time in expectation for parallelizable DAG jobs on m identical processors for all fixed $\epsilon > 0$.*

The two largest improvements on previous results for average flow time in the DAG model are that this algorithm is distributed and that it minimizes preemptions. Previous algorithms required a scheduler to have global coordination. Additionally, the previous algorithms required a number of preemptions unbounded in terms of m and n. We show the following about the number of preemptions regarding DREP. Importantly, the algorithm only preempts a job when a new job arrives.

Theorem 121 *Using the DREP scheduling policy requires processors to switch between unsatisfied jobs at most $O(mn)$ times over the entire schedule. Further, if jobs are sequential, the total expected number of preemptions is $O(n)$.*

The total number of preemptions for DREP is at most $O(mn)$ for parallel jobs and $O(n)$ if all jobs are sequential. For sequential jobs, this matches the best known results for *clairvoyant* algorithms which require complete knowledge of a job's structure [49]. As far as we are aware, our result is the first for a non-clairvoyant algorithm having guarantees on the number of preemptions and on average flow time simultaneously (even for sequential jobs). The closest result is that for Shortest-Elapsed-Time-First for sequential jobs which is known to be $(1+\epsilon)$-speed $O(1)$-competitive for average flow time on identical processors [13, 93]. This algorithm is an idealized version of the Unix scheduler. The Unix scheduler provides practical methods to ensure the algorithm uses $O(n \log n)$ preemptions.

The practical improvements of the algorithm are slightly offset by having worse speed augmentation than what is known for LAPS in theory, but we believe that this our work is the first theoretical result which could realistically be implemented and used in systems.

Other Related Work: Besides the work written above, there has been other work regarding scheduling parallelizable jobs. Much previous work in this domain has considered a different model of parallelizability known as the *arbitrary speed-up curves* model [64]. In this model, each job i is processed in phases sequentially. During the jth phase for job i the job is associated with a speed-up function $\Gamma_{i,j}(m')$ specifying the rate at which the job is processed when given m' processors. Typically it is assumed that $\Gamma_{i,j}$ is a non-decreasing concave function; although there is some exceptions to this [62]. There have been great strides in understanding this model and $(1+\epsilon)$-speed $O(1)$-competitive algorithms are known for average flow time [66], the ℓ_k-norms of flow time [65, 80], and flow time plus energy [47] and results are known for maximum flow time [131]. The work of [136] considers a hybrid of the DAG model and the speed-up curves setting.

While the speed-up curve model has been extensively studied, the model is a idealized theoretical model. As argued in Chapter 11, the DAG model is the model of parallelizability that most closest corresponds to jobs generated by parallel programs from most languages and libraries. It is also argued there that the results for the two models can not be directly translated and no model subsumes the other directly.

12.1 Preliminaries

We consider the problem of scheduling n total jobs that arrive online and must be scheduled on m identical processors. Each job is in the form of a Directed-Acyclic-Graph (DAG). In a DAG, a node is *ready* to be executed if all of its predecessors have been completed. Note that a node in the DAG may only be executed by one processor at a time. For a given job i, there are two important parameters. First the job has a total amount of work, W_i, which is the sum of the processing times of all the nodes in the job's DAG. Furthermore, the job's **critical-path length** C_i is the length of the longest path through its DAG, where the length is defined to be the sum of the processing times of the nodes along that path. There are two notable observations which involve these parameters.

Observation 122 *Any job J_i takes at least* $\max\{\frac{W_i}{m}, C_i\}$ *time to complete in any schedule with unit speed.*

Observation 123 *If a job J_i has all of its r ready nodes being executed by a schedule with speed s on m cores, where $r \leq m$, then the remaining critical-path length of i decreases at a rate of s.*

In this chapter, we will specify the algorithm DREP we introduce using A when using indices. Let $W_i^A(t)$ refer to sum of the remaining processing times of all nodes in job J_i's DAG in A's schedule at time t. Let $C_i^A(t)$ refer to the remaining critical-path length for job i in A's schedule at time t, that is, the longest remaining path. Let $A(t)$ denote the set of jobs in A's schedule which have arrived and remain unfinished at time t. In all of the notations above, we replace the index A with O when refering to the same quantity in the optimal schedule. We overload notation and let OPT refer to both the final objective of the optimal schedule and the schedule itself.

Potential Function Analysis: Throughout this chapter we will utilize the potential function framework, also known as amortized analysis. For this technique, one defines a potential function $\Phi(t)$ which depends on the state of the algorithm being considered and the optimal solution at time t. Let $G_a(t)$ denote the current cost of the algorithm at time t. This is the total waiting time of all the arrived jobs up to time t if the objective is total flow time. Similarly let $G_o(t)$ denote the current cost of the optimal solution up to time t. We note that $\frac{\mathtt{d}G_a(t)}{\mathtt{dt}}$ is the change in the algorithm's objective at time t and this is equal to the number of unsatisfied jobs in the algorithm's schedule at time t, i.e. $\frac{\mathtt{d}G_a(t)}{\mathtt{dt}} = |A(t)|$. To bound the competitiveness of an algorithm, one shows the following conditions about the potential function.

Boundary condition: Φ is zero before any job is released and Φ is non-negative after all jobs are finished.

Completion condition: Summing over all job completions by the optimal solution and the algorithm, Φ does not increase by more than $\beta \cdot \mathrm{OPT}$ for some $\beta \geq 0$.

Arrival condition: Summing over all job arrivals, Φ does not increase by more than $\alpha \cdot \mathrm{OPT}$ for some $\alpha \geq 0$.

Running condition: At any time t when no job arrives or is completed,

$$\frac{\mathtt{d}G_a(t)}{\mathtt{dt}} + \frac{\mathtt{d}\Phi(t)}{\mathtt{dt}} \leq c \cdot \frac{\mathtt{d}G_o(t)}{\mathtt{dt}} \tag{12.1}$$

Integrating these conditions over time one gets that $G_a - \Phi(0) + \Phi(\infty) \leq (\alpha + \beta + c) \cdot \mathrm{OPT}$ by the boundary, arrival and completion conditions. This shows the algorithm is $(\alpha+\beta+c)$-competitive

12.2 DREP: A New Sequential Algorithm

We first introduce our algorithm Distributed Random Equi-Partition (DREP) when jobs are sequential. The algorithm is a randomized distributed scheduling algorithm, which essentially distributes processors evenly in a randomized manner. The algorithm re-assigns processors to jobs only when a job arrives or completes.

When a new job arrives, there are two cases in how processors are re-distributed to the new job. If there are one or more free processors then one such processor takes the new job. Otherwise, if all processors are busy, then each processor switches to the new job with probability $\frac{1}{|A(t)|}$ (breaking ties arbitrarily to give the job to at most one processor), where $|A(t)|$ is the number of alive jobs at the moment. Jobs that are not taken by any processor are stored in a queue. A job J_j may be on this queue for two reasons: (1) J_j was not assigned to a processor on arrival (no processor happened to switch to it); or (2) J_j was executing on some processor and that processor preempted J_j and switched to working on another job that arrived later. When a job is completed, the processor assigned to the job chooses a job to work on uniformly at random from the queue of jobs.

One important consideration for a practical scheduling algorithm is the number of preemptions caused by the schedule. Notice that the only time that there are preemptions of jobs is when a job arrives. It is easy to see that the total number of preemptions is $O(n)$ in expectation implying the second part of Theorem 121. This is because either there is a processor that is free to take the new job and no preemption occurs. Otherwise, there are at least m jobs being processed and the probability an individual processor preempts is $\frac{1}{|A(t|} \leq \frac{1}{m}$. Over all processors, at most 1 preempts in expectation per job arrival.

This algorithm's theoretical guarantees for average flow time are subsumed by the analysis for the case where jobs are parallelizable, which can be found in the next section. We note that this is the first non-clairvoyant algorithm, even on a single machine, in the sequential setting to use $O(n)$ preemptions and be competitive for average flow time.

Observation 124 *DREP performs $O(n)$ preemptions in expectation when jobs are sequential.*

12.3 DREP: A New Parallel Algorithm and Analysis

In this section our goal is to prove the main Theorem 120 for DREP. We will first describe the scheduling algorithm of DREP for parallel jobs, then show how DREP maintains the probability of a processor working on a job, and finally prove the theorem via potential function. From this point on, we assume that our algorithm is given $(2 + 2\epsilon)$ speed for positive constant $\epsilon < \frac{1}{2}$.

12.3.1 DREP Algorithm for Parallel Jobs

First we introduce Distributed Random Equi-Partition (DREP) when scheduling jobs that are parallelizable. Like the sequential setting, the algorithm distributes processors in a randomized manner except now more than one processor can be assigned to a single job. The algorithm assigns (or re-assigns) processors to jobs only when a job arrives or completes.

When a new job arrives, each processor switches to the new job with probability $\frac{1}{|A(t)|}$. Jobs are allowed to have more than one processor assigned to them. Jobs not assigned to processors are stored in a queue. When a job is completed, all processors assigned to the job choose an unsatisfied job to work on uniformly at random (in the queue or currently being processed).

Processors work on the available ready nodes for the job they are assigned to using any work-conserving scheduler. Note that there may be more processors assigned to a job than the number of ready nodes in that job. In such a case, the extra processors may simply idle until more ready nodes become available.

The algorithm requires neither the information about the overall structure of the DAG for a job nor the knowledge of the total work of a job. Notice that on each processor a preemption of jobs only happen when a new job arrives. Therefore, it is easy to see that the total number of preemptions is $O(mn)$ implying the first part of Theorem 121.

Observation 125 *DREP performs $O(mn)$ preemptions in expectation when jobs are parallelizable.*

12.3.2 Probability of Working on a Job under DREP

Before we prove the competitive ratio of DREP, we first give a lemma about the probability that a processor is working on a specific job, followed by a corollary.

Lemma 126 *For any alive job J_j and a processor i the probability that processor i is working on job J_j is $\frac{1}{|A(t)|}$.*

Proof. We prove the lemma inductively on the arrival and completion of jobs. Fix any time t and let $n' = |A(t)|$ be the current number of alive jobs in the algorithm just before time t.

First consider the arrivals of jobs. Initially, when there are no jobs the lemma statement is vacuously true. Say that at this time a new job $J_{n'+1}$ arrives. The probability of any processor switching to this job $J_{n'+1}$ is $\frac{1}{n'+1}$ since there are now $n'+1$ jobs alive. Now consider any job J_j that was alive before the new job arrived. By the inductive hypothesis processor i is working on J_j with probability $\frac{1}{n'}$ just before job $J_{n'+1}$'s arrival. A processor which was working on J_j has a probability of $(1 - \frac{1}{n'+1})$ of not switching to the newly arrived job. Therefore, the overall probability of the processor continues working on J_j is then $\frac{1}{n'}(1 - \frac{1}{n'+1}) = \frac{1}{n'+1}$, the desired probability.

For the second case, say that a job $J_{j'}$ is completed at time t. Suppose a processor i becomes free after a job finishes. In the algorithm, the processor chooses a new job to work on at random. This precisely gives a probability of $\frac{1}{n'-1}$ chance to process any job, the desired probability. The lemma holds for any alive job and any processor i that became free. Alternatively, consider a processor i not working on the job completed. Let $i \to j$ be the event that processor i is working on job J_j just before time t and $i \nrightarrow j$ be the event it is not. This processor is working on any alive J_j with probability

$$\Pr[i \to j \mid i \nrightarrow j'] = \Pr[i \to j \text{ and } i \nrightarrow j'] / \Pr[i \nrightarrow j']$$

Inductively, we have $\Pr[i \nrightarrow j'] = 1 - \frac{1}{n'}$ and $\Pr[i \to j \text{ and } i \nrightarrow j'] = \Pr[i \to j] = \frac{1}{n'}$. Thus, $\Pr[i \to j \mid i \nrightarrow j'] = \frac{1}{n'-1}$. □

Using lemma 126 and the fact that there are a total of m processors, we can show the following lemma using the definition of expected value.

Corollary 127 *For any alive job J_j at any time t, the expected number of processors it is assigned is $\frac{m}{|A(t)|}$.*

12.3.3 Potential Function Analysis for DREP

We analyze this algorithm using potential functions. Let $Z_i(t) := \max\{W^A(t) - W^O(t), 0\}$ for each job J_i. The variable $Z_i(t)$ is the total amount of work job J_i has fallen behind in algorithm at time t as compared to the optimal solution (the lag of i). Further, let $C_i^A(t)$ be the remaining critical path length for job J_i in the algorithm's schedule.

Define $\texttt{rank}_i(t) = \sum_{j \in A(t), r_j \leq r_i} 1$ of job J_i to be the number of jobs in $A(t)$ that arrived before job J_i.

The potential function we use is given here. Note that it is a summation over all jobs in the algorithm's queue.

$$\Phi(t) = \frac{10}{\epsilon} \sum_{i \in A(t)} \left(\frac{1}{m} \texttt{rank}_i(t) Z_i(t) + \frac{10}{\epsilon^2} C_i^A(t) \right)$$

We will show the arrival and completion conditions, followed by the running condition.

Lemma 128 *The completion of jobs by either the algorithm or the optimal schedule do not increase the potential.*

Proof. Consider when the algorithm completes a job J_j at time t. First look at its contribution to its own term in the summation. Note that when j is completed, by definition both $C_j^A(t)$ and $Z_j(t)$ are zero because j must not have any work remaining. Therefore, removing this term in the summation has no effect on the potential. Now look at the job's contribution to the terms for other jobs in the summation. The only effect of removing j is that the rank of jobs which arrived after it are decreased. Therefore, this decreases the potential as all terms are positive. Hence, in either case, the potential does not increase. □

Lemma 129 *The arrival of all jobs increases the potential function by at most* $O(\frac{1}{\epsilon^3})\mathrm{OPT}$.

Proof. Let i be a job that arrives at some time t. The arrival of a job does not change the rank of any previous jobs, therefore, its only effect is the addition of a term in the potential for itself. Examine the corresponding term of the potential. Since neither the algorithm nor the optimal schedule have worked on this job yet, $Z_i(t) = 0$.

Job i has a critical path length of C_i. Therefore, the change in the potential is $(\frac{10}{\epsilon})\frac{10}{\epsilon^2}C_i$. However, recall that C_i is a lower bound on the flow time of job J_i. Therefore, calculating the total over all jobs, the change in potential is bounded by $O(\frac{1}{\epsilon^3})\mathrm{OPT}$. □

Now we must show that the running condition holds. Recall that the running condition involves the instantaneous change at any moment in time. We bound this for each fixed time. Consider any time t. We first show the change in the potential function $\Phi(t)$ due to the processing of OPT.

Lemma 130 *The optimal schedule's processing of jobs at t increases the potential function by at most $\frac{10}{\epsilon}|A(t)|$.*

Proof. The optimal schedule's processing only changes the first term of any item in the summation of the potential, since $C_i^A(t)$ only depends on the algorithm. The first term for any job is a product of the rank and work remaining of the job. Therefore, the increase in potential is maximized if OPT were to use all m processors to work on the job with maximum rank. Note that the maximum rank is exactly equal to the number of jobs remaining in $A(t)$. Therefore, the increase in potential is no more than $m\frac{10}{\epsilon}\frac{1}{m}|A(t)| = \frac{10}{\epsilon}|A(t)|$. □

Now we are ready to show the running condition.

Lemma 131 *In expectation, at any time t the running condition holds as following.*

$$\frac{dG_a(t)}{dt} + \frac{d\Phi(t)}{dt} \leq O(\frac{1}{\epsilon^2}) \cdot \frac{dG_o(t)}{dt}$$

At time t, consider the set of jobs alive in the algorithm and fix this set to be $A(t)$. Though $A(t)$ is a random variable dependent on the processing of the algorithm, we will show that the running condition holds for any set $A(t)$. If we do so, then by the definition of expected value we have shown that in expectation the running condition holds. We fix a set $A(t)$ for the analysis and break this analysis into two cases.

In the first case, there are a few jobs which possess a small amount of ready nodes.

Lemma 132 *Suppose that there are at least $\frac{\epsilon}{10}|A(t)|$ jobs for which the number of ready nodes is no more than $\frac{m}{|A(t)|}$, then the running condition is satisfied in expectation.*

Proof. Let the set of such jobs with the required number of ready nodes be S. In this case, because of Corollary 127 we know that all ready nodes of these jobs are being scheduled in expectation. From Observation 123, the critical path length $C_i^A(t)$ of this job is being decreased at a rate of $(2+2\epsilon)$ which changes the potential. Therefore, due to the processing of the algorithm, the expected change in potential at least the following:

$$\begin{aligned}\mathbb{E}\left[\frac{\mathtt{d}\Phi(t)}{\mathtt{dt}}\right] &\leq -\frac{10}{\epsilon}\sum_{i\in S}\left(\frac{10}{\epsilon^2}\frac{\mathtt{d}C_i^A(t)}{\mathtt{dt}}\right) \leq -\frac{10}{\epsilon}\sum_{i\in S}\left(\frac{10}{\epsilon^2}(2+2\epsilon)\right)\\ &\leq -\frac{10}{\epsilon}\frac{\epsilon}{10}|A(t)|\frac{10}{\epsilon^2}(2+2\epsilon) \leq -\frac{10}{\epsilon^2}(2+2\epsilon)|A(t)|\\ &\leq -\frac{20}{\epsilon}|A(t)|\end{aligned}$$

Combining this with the Lemma 130 we see that the change of the potential in expectation is at most $-\frac{10}{\epsilon}|A(t)|$. Therefore, in expectation the running condition is satisfied (since $\frac{\mathtt{d}G_a(t)}{\mathtt{dt}} = |A(t)|$). □

Now it remains to show the other case. But first, we make the following useful claim.

Claim 133 *At time t, if $|O(t)| \geq \frac{\epsilon}{10}|A(t)|$, then the running condition is satisfied. In other words, if there are at least $\frac{\epsilon}{10}|A(t)|$ jobs alive in the optimal schedule at time t, the running condition is satisfied.*

Proof. Note that the algorithm cannot increase the potential function due to its processing since it may only decrease the remaining critical path length or work on jobs. Now, suppose $|O(t)| \geq \frac{\epsilon}{10}$, we will ignore the algorithm's impact on the potential. We combine this condition and Lemma 130 to examine the running condition.

$$\frac{\mathtt{d}G_a(t)}{\mathtt{dt}} + \frac{\mathtt{d}\Phi(t)}{\mathtt{dt}} \leq |A(t)| + \frac{10}{\epsilon}|A(t)| \leq (1+\frac{10}{\epsilon})|\frac{10}{\epsilon}|O(t)|$$
$$\leq O(\frac{1}{\epsilon^2})|O(t)| \leq O(\frac{1}{\epsilon^2})\frac{\mathtt{d}G_o(t)}{\mathtt{dt}}$$

□

This claim effectively restricts the number of alive jobs in the optimal schedule. Now we can prove the second case, in which most jobs have a large amount of ready nodes.

Lemma 134 *Suppose that there are at least $(1-\frac{\epsilon}{10})|A(t)|$ jobs for which there are more ready nodes than $\frac{m}{|A(t)|}$, then the running condition is satisfied in expectation.*

Proof. In this case, let all jobs which satisfy the ready nodes condition be the set S, note that $|S| \geq (1-\frac{\epsilon}{10})|A(t)|$. Note that if there are many jobs in $|O(t)|$ the proof is done due to claim 133. Therefore, we assume that the optimal schedule has less than $\frac{\epsilon}{10}|A(t)|$ jobs alive. Let S_A be the set of jobs in S which are in $|A(t)|$ but not $|O(t)$. Since there is a bound on the number of jobs in $|O(t)|$, we can show that $|S_A| \geq |S| - \frac{\epsilon}{10}|A(t)| \geq (1-\frac{\epsilon}{5})|A(t)|$. For each job J_i in S_A, the algorithm's processing affects the $Z_i(t)$ term in the potential function. Note that since $|A(t)|$ is fixed, the rank of each such job is fixed. Therefore we may bound the expected change in potential due to the effects on the Z terms. Recall that in expectation, each job receives $\frac{m}{|A(t)|}$ processors (Corollary 127).

$$\mathbb{E}\left[\frac{\mathtt{d}\Phi_a(t)}{\mathtt{dt}}\right] \leq -\frac{10}{\epsilon}\sum_{i\in S_A}\left(\frac{1}{m}\mathtt{rank}_i(t)\mathbb{E}\left[\frac{\mathtt{d}Z_i(t)}{\mathtt{dt}}\right]\right)$$
$$\leq -\frac{10}{\epsilon}\sum_{i\in S_A}\left(\frac{1}{m}\mathtt{rank}_i(t)\frac{m}{|A(t)|}(2+2\epsilon)\right)$$
$$\leq -\frac{20+20\epsilon}{\epsilon|A(t)|}\sum_{i\in S_A}\mathtt{rank}_i(t)$$

Here in the worst case, the jobs alive in S_A have the lowest rank, therefore we can switch the summation to a summation of ranks.

$$\mathbb{E}\left[\frac{\mathtt{d}\Phi_a(t)}{\mathtt{dt}}\right] \leq -\frac{20+20\epsilon}{\epsilon|A(t)|}\sum_{i=1}^{(1-\frac{\epsilon}{5})|A(t)|} i \leq -\frac{20+20\epsilon}{\epsilon|A(t)|}\frac{(1-\frac{\epsilon}{5})^2|A(t)|^2}{2}$$
$$\leq -\frac{1}{\epsilon}|A(t)|\left((10+10\epsilon)(1-\frac{\epsilon}{5})^2\right) \leq -\frac{1}{\epsilon}|A(t)|(1+3\epsilon) \quad [\epsilon \leq \tfrac{1}{2}]$$

This is the expected change in potential. Note that from Lemma 130 we know that the optimal schedule changes the potential by $\frac{1}{\epsilon}|A(t)|$. Therefore, the total change in potential is $-3|A(t)|$. This means that for the running condition we have the following in aggregate over the algorithm and optimal solutions processing jobs:

$$\mathbb{E}\left[\frac{\mathtt{d}G_a(t)}{\mathtt{dt}} + \frac{\mathtt{d}\Phi(t)}{\mathtt{dt}}\right] \leq |A(t)| - 3|A(t)| \leq 0$$

This means that the running condition is satisfied in expectation for this case. □

Proof of 131. By Lemmas 132 and 134, we have shown the running condition is satisfied for any given set $A(t)$ in expectation. This result is conditioned on a given set $A(t)$. To show that the running condition holds in expectation, it is necessary to invoke the definition of expected value. However in this case all the outcomes are the same. Therefore, the running condition holds in expectation at any time t and the lemma is proved. □

Since we have proved the completion, arrival, and running conditions. Theorem 120 directly follows due to the potential function framework. Note that the competitive ratio is $O(\frac{1}{\epsilon^3}) + O(\frac{1}{\epsilon^2}) = O(\frac{1}{\epsilon^3})$.

Chapter 13

Conclusion

Today, all computing platforms, from cellphones to desktops to clouds, are parallel machines. In addition, the imperatives to reduce power consumptions as well as assembly and production costs are pushing system deployment toward combining multiple applications onto a common platform. Examples span from clouds where multiple clients submit their applications to consolidated Electronic Control Units on modern cars that host multiple applications with diverse functionalities from brake control to infotainment. As this integration continues, we face challenges in designing platforms which can provide appropriate guarantees to each latency-critical application while preserving scalability. Hence, this thesis focuses on developing theoretical foundations and practical implementations to increase the efficiency of parallel computing platforms, with a particular focus on systems that have latency-critical applications.

In scheduling latency-critical jobs on multicore systems, this thesis focuses on jobs which require parallelism. While researchers have looked at how to schedule multiple sequential jobs with various kinds of latency-related constraints, there has been little research on how to schedule multiple parallel jobs. However, with increasingly complex functionalities, todays applications must exploit internal parallelism and utilize multiple cores at the same time, in order to complete their growing computational demands within stringent timing constraints.

Part I considers static real-time systems where the arrival patterns and computational demands of jobs are known in advance and modeled as reoccurring real-time tasks; and the temporal correctness (i.e., schedulability) of tasks must be guaranteed prior to the execution. For this type of systems, the thesis first analyzed the classic GEDF and GRM schedulers for parallel tasks, and proved an upper and lower bound of 2.6 for capacity augmentation of

GEDF and an upper bound of 3.7 for GRM. The thesis then proposed a novel scheduling paradigm, called federated scheduling, that assigns dedicated cores to each parallel task and calculates the best core allocation. Interestingly, despite the fact that a task is only allowed to execute on a limited number of cores instead of all the cores like GEDF, federated scheduling has the optimal capacity bound of 2. Moreover, it has better locality and lower overheads in practice. The thesis further considers the static real-time systems that have multiple criticality levels or only requires meeting soft deadlines, and generalize the federated paradigm to these systems.

Part II focuses on online system with latency-critical applications. In many application environments such as clouds, grids and shared servers, clients send jobs to be processed on a server and the server scheduler decides when to process them. The goal of the scheduler is both to use the resources efficiently and to provide a good quality of service to jobs. In these systems, jobs arrive over time and the scheduler does not know the existence of jobs until they arrive. Instead of guaranteeing schedulability prior to execution (which would be impossible), online schedulers try to optimize some application-specific performance objectives. Therefore, the thesis considers different objectives, including minimizing average latency, maximum latency, number of jobs missing their individual deadlines and number of jobs missing a target latency. For each objective, the thesis proposed corresponding schedulers that are provably good and practically efficient.

By developing new scheduling strategies, analysis tools, and practical platform design techniques, this thesis shows that via appropriately leveraging the internal parallelism of latency-critical applications, static and online real-time systems are able to exploit the untapped efficiencies in the multicore platforms to drastically improve the quality of service guarantees of applications with increasing computational demands.

References

[1] Umut A Acar, Guy E Blelloch, and Robert D Blumofe. The data locality of work stealing. In *12th ACM symposium on Parallel algorithms and architectures (SPAA)*, pages 1–12, 2000.

[2] Kunal Agrawal, Charles E Leiserson, Yuxiong He, and Wen Jing Hsu. Adaptive work-stealing with parallelism feedback. *ACM Transactions on Computer Systems (TOCS)*, 26(3):7, 2008.

[3] Christoph Ambühl and Monaldo Mastrolilli. On-line scheduling to minimize max flow time: an optimal preemptive algorithm. *Oper. Res. Lett.*, 33(6):597–602, 2005.

[4] S. Anand, Naveen Garg, and Amit Kumar. Resource augmentation for weighted flow-time explained by dual fitting. In *ACM-SIAM Symposium on Discrete Algorithms SODA*, pages 1228–1241, 2012.

[5] Björn Andersson, Sanjoy Baruah, and Jan Jonsson. Static-priority scheduling on multiprocessors. In *22nd IEEE Real-Time Systems Symposium (RTSS)*, pages 193–202, 2001.

[6] Björn Andersson and Dionisio de Niz. Analyzing global-edf for multiprocessor scheduling of parallel tasks. *Principles of Distributed Systems*, pages 16–30, 2012.

[7] Björn Andersson and Jan Jonsson. The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%. In *15th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 33–40, 2003.

[8] Nimar S Arora, Robert D Blumofe, and C Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34(2):115–144, 2001.

[9] Philip Axer, Sophie Quinton, Moritz Neukirchner, Rolf Ernst, Bjorn Dobel, and Hermann Hartig. Response-time analysis of parallel fork-join workloads with real-time constraints. In *Real-Time Systems (ECRTS), 25th Euromicro Conference on*, pages 215–224, 2013.

[10] Theodore P Baker and Sanjoy K Baruah. Sustainable multiprocessor scheduling of sporadic task systems. In *21st Euromicro Conference on Real-Time Systems (ECRTS)*, pages 141–150, 2009.

[11] Nikhil Bansal, Ho-Leung Chan, and Kirk Pruhs. Competitive algorithms for due date scheduling. *Algorithmica*, 59(4):569–582, 2011.

[12] Nikhil Bansal, Ravishankar Krishnaswamy, and Viswanath Nagarajan. Better scalable algorithms for broadcast scheduling. *ACM Transactions on Algorithms*, 11:3:1–3:24, 2014.

[13] Neal Barcelo, Sungjin Im, Benjamin Moseley, and Kirk Pruhs. Shortest-elapsed-time-first on a multiprocessor. In *Design and Analysis of Algorithms - 1st Mediterranean Conference on Algorithms (MedAlg)*, pages 82–92, 2012.

[14] Sanjoy Baruah. Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors. *Computers, IEEE Transactions on*, 53(6):781–784, 2004.

[15] Sanjoy Baruah. Certification-cognizant scheduling of tasks with pessimistic frequency specification. In *IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 31–38, 2012.

[16] Sanjoy Baruah. Semantics-preserving implementation of multirate mixed-criticality synchronous programs. In *20th International Conference on Real-Time and Network Systems (RTNS)*, pages 11–19, 2012.

[17] Sanjoy Baruah and Theodore Baker. Schedulability analysis of global edf. *Real-Time Systems*, 38(3):223–235, 2008.

[18] Sanjoy Baruah, Vincenzo Bonifaci, Gianlorenzo D'Angelo, Haohan Li, Alberto Marchetti-Spaccamela, Suzanne Van Der Ster, and Leen Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *Real-Time Systems (ECRTS), 24th Euromicro Conference on*, pages 145–154, 2012.

[19] Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Sebastian Stiller. Improved multiprocessor global schedulability analysis. *Real-Time Systems*, 46(1):3–24, 2010.

[20] Sanjoy Baruah, Alan Burns, and Robert Davis. Response-time analysis for mixed criticality systems. In *32nd IEEE Real-Time Systems Symposium (RTSS)*, pages 34–43, 2011.

[21] Sanjoy Baruah, Bipasa Chattopadhyay, Haohan Li, and Insik Shin. Mixed-criticality scheduling on multiprocessors. *Real-Time Systems*, 50(1):142–177, 2014.

[22] Sanjoy Baruah, Haohan Li, and Leen Stougie. Towards the design of certifiable mixed-criticality systems. In *16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 13–22, 2010.

[23] Sanjoy K Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leen Stougie, and Andreas Wiese. A generalized parallel task model for recurrent real-time processes. In *33rd IEEE Real-Time Systems Symposium (RTSS)*, pages 63–72, 2012.

[24] Sanjoy K. Baruah, Gilad Koren, Decao Mao, Bhubaneswar Mishra, Arvind Raghunathan, Louis E. Rosier, Dennis Shasha, and Fuxing Wang. On the competitiveness of on-line real-time task scheduling. *Real-Time Systems*, 4(2):125–144, 1992.

[25] Sanjoy K. Baruah, Gilad Koren, Bhubaneswar Mishra, Arvind Raghunathan, Louis E. Rosier, and Dennis Shasha. On-line scheduling in the presence of overload. In *Symposium on Foundations of Computer Science*, pages 100–110, 1991.

[26] Sanjoy K. Baruah, Aloysius K. Mok, and Louis E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *11th IEEE Real-Time Systems Symposium (RTSS)*, pages 182–190, 1990.

[27] Luca Becchetti, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Kirk Pruhs. Online weighted flow time and deadline scheduling. *Journal of Discrete Algorithms*, 4(3):339–352, 2006.

[28] Michael A. Bender, Soumen Chakrabarti, and S. Muthukrishnan. Flow and stretch metrics for scheduling continuous job streams. In *SODA '98*, pages 270–279, 1998.

[29] Marko Bertogna and Sanjoy Baruah. Tests for global edf schedulability analysis. *Journal of systems architecture*, 57(5):487–497, 2011.

[30] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *Parallel and Distributed Systems, IEEE Transactions on*, 20(4):553–566, 2009.

[31] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM*, 46(2):281–321, 1999.

[32] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.

[33] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[34] Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Sebastian Stiller, and Andreas Wiese. Feasibility analysis in the sporadic dag task model. In *25th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 225–233, 2013.

[35] Sem C Borst, Onno J Boxma, Rudesindo Núñez-Queija, and AP Zwart. The impact of the service discipline on delay asymptotics. *Performance Evaluation*, 54(2):175–206, 2003.

[36] Björn B. Brandenburg and James H. Anderson. On the implementation of global real-time schedulers. In *30th IEEE Real-Time Systems Symposium (RTSS)*, pages 214–224, 2009.

[37] Björn B Brandenburg, John M Calandrino, and James H Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 157–169, 2008.

[38] Mark Broadie and Paul Glasserman. Estimating security price derivatives using simulation. *Manage. Sci.*, 42:269–285, 1996.

[39] Alan Burns and Rob Davis. Mixed criticality systems: A review. *Department of Computer Science, University of York, Tech. Rep*, 2016.

[40] Carl Bussema and Eric Torng. Greedy multiprocessor server scheduling. *Operations research letters*, 34(4):451–458, 2006.

[41] Colin Campbell and Ade Miller. *A Parallel Programming with Microsoft Visual C++: Design Patterns for Decomposition and Coordination on Multicore Architectures.* Microsoft Press, 2011.

[42] Felipe Cerqueira and Björn B Brandenburg. A comparison of scheduling latency in linux, preempt-rt, and litmusrt. pages 19–29, 2013.

[43] Felipe Cerqueira, Manohar Vanga, and Björn B Brandenburg. Scaling global scheduling with massage passing. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), IEEE 20th*, pages 263–274, 2014.

[44] Felipe Cerqueira, Manohar Vanga, and Bjorn B Brandenburg. Scaling global scheduling with message passing. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 263–274, 2014.

[45] J. S. Chadha, N. Garg, A. Kumar, and V. N. Muralidhara. A competitive algorithm for minimizing weighted flow time on unrelated machines with speed augmentation. In *STOC*, 2009.

[46] Ho-Leung Chan, Jeff Edmonds, Tak Wah Lam, Lap-Kei Lee, Alberto Marchetti-Spaccamela, and Kirk Pruhs. Nonclairvoyant speed scaling for flow and energy. In *STACS*, pages 255–264, 2009.

[47] Ho-Leung Chan, Jeff Edmonds, and Kirk Pruhs. Speed scaling of processes with arbitrary speedup curves on a multiprocessor. *Theory Comput. Syst.*, 49:817–833, 2011.

[48] Sze-Hang Chan, Tak Wah Lam, and Lap-Kei Lee. Non-clairvoyant speed scaling for weighted flow time. In *ESA (1)*, pages 23–35, 2010.

[49] Chandra Chekuri, Ashish Goel, Sanjeev Khanna, and Amit Kumar. Multi-processor scheduling to minimize flow time with epsilon resource augmentationn. In *STOC*, pages 363–372, 2004.

[50] Chandra Chekuri, Sungjin Im, and Benjamin Moseley. Online scheduling to minimize maximum response time and maximum delay factor. *Theory of Computing*, 8(1):165–195, 2012.

[51] Hoon Sung Chwa, Jinkyu Lee, Kieu-My Phan, Arvind Easwaran, and Insik Shin. Global edf schedulability analysis for synchronous parallel tasks on multicore platforms. In *Real-Time Systems (ECRTS), 25th Euromicro Conference on*, pages 25–34, 2013.

[52] Sébastien Collette, Liliana Cucu, and Joël Goossens. Integrating job parallelism in real-time scheduling theory. *Information Processing Letters*, 106(5):180–187, 2008.

[53] G. Cortazar, M. Gravet, and J. Urzua. The valuation of multidimensional american real options using the lsm simulation method. *Comp. and Operations Research.*, 35(1):113–129, 2008.

[54] Robert I Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43(4):35, 2011.

[55] Dionisio de Niz and Linh TX Phan. Partitioned scheduling of multi-modal mixed-criticality real-time systems on multiprocessor platforms. In *20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 111–122, 2014.

[56] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[57] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 205–220, 2007.

[58] Umamaheswari C Devi. *Soft real-time scheduling on multiprocessors*. PhD thesis, University of North Carolina at Chapel Hill, 2006.

[59] UmaMaheswari C Devi and James H Anderson. Tardiness bounds under global edf scheduling on a multiprocessor. *Real-Time Systems*, 38(2):133–189, 2008.

[60] José Luis Díaz, Daniel F García, Kanghee Kim, Chang-Gun Lee, Lucia Lo Bello, José María López, Sang Lyul Min, and Orazio Mirabella. Stochastic analysis of periodic real-time systems. In *23rd IEEE Real-Time Systems Symposium (RTSS)*, pages 289–300, 2002.

[61] Arvind Easwaran. Demand-based scheduling of mixed-criticality sporadic tasks on one processor. In *34th IEEE Real-Time Systems Symposium (RTSS)*, pages 78–87, 2013.

[62] Roozbeh Ebrahimi, Samuel McCauley, and Benjamin Moseley. Scheduling parallel jobs online with convex and concave parallelizability. In *Approximation and Online Algorithms - 13th International Workshop (WAOA)*, pages 183–195, 2015.

[63] Jeff Edmonds. Scheduling in the dark. *Theor. Comput. Sci.*, 235(1):109–141, 2000. Preliminary version in *STOC* 1999.

[64] Jeff Edmonds, Donald D Chinn, Tim Brecht, and Xiaotie Deng. Non-clairvoyant multiprocessor scheduling of jobs with changing execution characteristics. *J. Scheduling*, 6(3):231–250, 2003.

[65] Jeff Edmonds, Sungjin Im, and Benjamin Moseley. Online scalable scheduling for the ℓ_k-norms of flow time without conservation of work. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2011.

[66] Jeff Edmonds and Kirk Pruhs. Scalably scheduling processes with arbitrary speedup curves. *ACM Transactions on Algorithms*, 8:28, 2012.

[67] Pontus Ekberg and Wang Yi. Bounding and shaping the demand of generalized mixed-criticality sporadic task systems. *Real-time systems*, 50(1):48–86, 2014.

[68] Jeremy Erickson, U Devi, and Sanjoy Baruah. Improved tardiness bounds for global edf. In *Real-Time Systems (ECRTS), 22nd Euromicro Conference on*, pages 14–23, 2010.

[69] Jeremy P Erickson, James H Anderson, and Bryan C Ward. Fair lateness scheduling: Reducing maximum lateness in g-edf-like scheduling. *Real-Time Systems*, 50(1):5–47, 2014.

[70] David Ferry, Gregory Bunting, Amin Maghareh, Arun Prakash, Shirley Dyke, Kunal Agrawal, Christopher Gill, and Chenyang Lu. Real-time system support for hybrid structural simulation. In *14th International Conference on Embedded Software (EMSOFT)*, page 25, 2014.

[71] David Ferry, Jing Li, Mahesh Mahadevan, Kunal Agrawal, Christopher Gill, and Chenyang Lu. A real-time scheduling service for parallel tasks. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 261–272, 2013.

[72] Nathan Fisher, Sanjoy Baruah, and Theodore P Baker. The partitioned scheduling of sporadic tasks according to static-priorities. In *18th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 10–pp, 2006.

[73] Kyle Fox, Sungjin Im, and Benjamin Moseley. Energy efficient scheduling of parallelizable jobs. In *ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 948–957, 2013.

[74] Kyle Fox and Benjamin Moseley. Online scheduling on identical machines using srpt. In *SODA*, pages 120–128, 2011.

[75] Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the cilk-5 multithreaded language. In *ACM Sigplan Notices*, volume 33, pages 212–223. ACM, 1998.

[76] Joël Goossens, Shelby Funk, and Sanjoy Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-time systems*, 25(2-3):187–205, 2003.

[77] Ronald L Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9):1563–1581, 1966.

[78] Xiaozhe Gu, Arvind Easwaran, Kieu-My Phan, and Insik Shin. Resource efficient isolation mechanisms in mixed-criticality scheduling. In *Real-Time Systems (ECRTS), 27th Euromicro Conference on*, pages 13–24, 2015.

[79] Nan Guan, Pontus Ekberg, Martin Stigge, and Wang Yi. Effective and efficient scheduling of certifiable mixed-criticality sporadic task systems. In *32nd IEEE Real-Time Systems Symposium (RTSS)*, pages 13–23, 2011.

[80] Anupam Gupta, Sungjin Im, Ravishankar Krishnaswamy, Benjamin Moseley, and Kirk Pruhs. Scheduling jobs with varying parallelizability to reduce variance. In *Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 11–20, 2010.

[81] Anupam Gupta, Sungjin Im, Ravishankar Krishnaswamy, Benjamin Moseley, and Kirk Pruhs. Scheduling heterogeneous processors isn't as easy as you think. In *ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 1242–1253, 2012.

[82] Md E Haque, Yong hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S McKinley. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 161–175, 2015.

[83] Y. He, S. Elnikety, J. Larus, and C. Yan. Zeta: Scheduling interactive services with partial execution. In *ACM Symposium on Cloud Computing (SOCC)*, page 12, 2012.

[84] Yuxiong He, Wen-Jing Hsu, and Charles E Leiserson. Provably efficient online nonclairvoyant adaptive scheduling. *Parallel and Distributed Systems, IEEE Transactions on (TPDS)*, 19(9):1263–1279, 2008.

[85] Yuxiong He, Charles E Leiserson, and William M Leiserson. The cilkview scalability analyzer. In *22nd ACM symposium on Parallelism in algorithms and architectures (SPAA)*, pages 145–156, 2010.

[86] Sungjin Im and Benjamin Moseley. Online scalable algorithm for minimizing lk-norms of weighted flow time on unrelated machines. In *Proceedings of the 22nd ACM-SIAM Symposium on Discrete Algorithms SODA, 2011*, pages 95–108, 2011.

[87] Sungjin Im and Benjamin Moseley. General profit scheduling and the power of migration on heterogeneous machines. In *Symposium on Parallelism in Algorithms and Architectures*, 2016.

[88] Sungjin Im, Benjamin Moseley, and Kirk Pruhs. A tutorial on amortized local competitiveness in online scheduling. *SIGACT News*, 42(2):83–97, 2011.

[89] Intel. Intel CilkPlus v1.2, Sep 2013. https://www.cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_1.2.htm.

[90] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. Speeding up distributed request-response workflows. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 219–230, 2013.

[91] Myeongjae Jeon, Yuxiong He, Sameh Elnikety, Alan L Cox, and Scott Rixner. Adaptive parallelism for web search. In *ACM European Conference on Computer Systems*, pages 155–168, 2013.

[92] Bala Kalyanasundaram and Kirk Pruhs. Fault-tolerant real-time scheduling. *Algorithmica*, 28(1):125–144, 2000.

[93] Bala Kalyanasundaram and Kirk Pruhs. Speed is as powerful as clairvoyance. *J. ACM*, 47(4):617–643, 2000.

[94] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G. Wei, and David Brooks. Profiling a warehouse-scale computer. In *ACM SIGARCH International Conference on Computer Architecture (ISCA)*, pages 158–169, 2015.

[95] Shinpei Kato and Yutaka Ishikawa. Gang edf scheduling of parallel task systems. In *30th IEEE Real-Time Systems Symposium (RTSS)*, pages 459–468, 2009.

[96] Junsung Kim, Hyoseung Kim, Karthik Lakshmanan, and Ragunathan Raj Rajkumar. Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In *4th International Conference on Cyber-Physical Systems (ICCPS)*, pages 31–40, 2013.

[97] Saehoon Kim, Yuxiong He, Seung-Won Hwang, Sameh Elnikety, and Seungjin Choi. Delayed-Dynamic-Selective (DDS) prediction for reducing extreme tail latency in web search. In *ACM International Conference on Web Search and Data Mining (WSDM)*, 2015.

[98] JFC Kingman. Inequalities in the theory of queues. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 102–110, 1970.

[99] Leonard Kleinrock. Time-shared systems: A theoretical treatment. *Journal of the ACM (JACM)*, 14(2):242–261, 1967.

[100] Gilad Koren and Dennis Shasha. MOCA: A multiprocessor on-line competitive algorithm for real-time system scheduling. *Theor. Comput. Sci.*, 128(1&2):75–97, 1994.

[101] Gilad Koren and Dennis Shasha. Dover: An optimal on-line scheduling algorithm for overloaded uniprocessor real-time systems. *SIAM J. Comput.*, 24(2):318–339, 1995.

[102] Karthik Lakshmanan, Dionisio de Niz, and Ragunathan Rajkumar. Mixed-criticality task synchronization in zero-slack scheduling. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 47–56, 2011.

[103] Karthik Lakshmanan, Shinpei Kato, and Ragunathan Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *31st IEEE Real-Time Systems Symposium (RTSS)*, pages 259–268, 2010.

[104] Doug Lea. A Java fork/join framework. In *ACM 2000 Conference on Java Grande*, pages 36–43, 2000.

[105] Jinkyu Lee and Kang G Shin. Controlling preemption for better schedulability in multi-core systems. In *33rd IEEE Real-Time Systems Symposium (RTSS)*, pages 29–38, 2012.

[106] Wan Yeon Lee and LEE Heejo. Optimal scheduling for real-time parallel tasks. *IEICE transactions on information and systems*, 89(6):1962–1966, 2006.

[107] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *Acm Sigplan Notices*, volume 44, pages 227–242, 2009.

[108] Juri Lelli, Giuseppe Lipari, Dario Faggioli, and Tommaso Cucinotta. An efficient and scalable implementation of global edf in linux. In *7th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, pages 6–15, 2011.

[109] Stefano Leonardi and Danny Raz. Approximating total flow time on parallel machines. *Journal of Computer and Systems Sciences*, 73(6):875–891, 2007.

[110] Hennadiy Leontyev and James H Anderson. Generalized tardiness bounds for global multiprocessor scheduling. *Real-Time Systems*, 44(1-3):26–71, 2010.

[111] J. Li, Jian-Jia Chen, K. Agrawal, C.Lu, C.D. Gill, and Abusayeed Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *Real-Time Systems (ECRTS), 26th Euromicro Conference on*, pages 85–96, 2014.

[112] Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Analysis of global edf for parallel tasks. In *Real-Time Systems (ECRTS), 25th Euromicro Conference on*, pages 3–13, 2013.

[113] Jing Li, Zheng Luo, David Ferry, Kunal Agrawal, Christopher Gill, and Chenyang Lu. Global edf scheduling for parallel real-time tasks. *Real-Time Systems*, 51(4):395–439, 2015.

[114] Cong Liu and James Anderson. Supporting soft real-time parallel applications on multicore processors. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), IEEE 18th International Conference on*, pages 114–123, 2012.

[115] Guangdong Liu, Ying Lu, Shige Wang, and Zonghua Gu. Partitioned multiprocessor scheduling of mixed-criticality parallel jobs. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), IEEE 20th International Conference on*, pages 1–10, 2014.

[116] José María López, José Luis Díaz, Joaquín Entrialgo, and Daniel García. Stochastic analysis of real-time systems under preemptive priority-driven scheduling. *Real-Time Systems*, 40(2):180–207, 2008.

[117] José María López, José Luis Díaz, and Daniel F García. Utilization bounds for edf scheduling on real-time multiprocessor systems. *Real-Time Systems*, 28(1):39–68, 2004.

[118] Jacob R. Lorch and Alan Jay Smith. Improving dynamic voltage scaling algorithms with PACE. In *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 50–61, 2001.

[119] Brendan Lucier, Ishai Menache, Joseph Naor, and Jonathan Yaniv. Efficient online scheduling for deadline-sensitive jobs: extended abstract. In *SPAA '13*, pages 305–314, 2013.

[120] Amin Maghareh, Shirley Dyke, Arun Prakash, Gregory Bunting, and Payton Lindsay. Evaluating modeling choices in the implementation of real-time hybrid simulation. *EMI/PMC*, 2012.

[121] G Manimaran, C Siva Ram Murthy, and Krithi Ramamritham. A new approach for scheduling of parallelizable tasks in real-time multiprocessor systems. *Real-Time Systems*, 15(1):39–60, 1998.

[122] Alex F Mills and James H Anderson. A stochastic framework for multiprocessor soft real-time scheduling. In *16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 311–320, 2010.

[123] Geoffrey Nelissen, Vandy Berten, Joël Goossens, and Dragomir Milojevic. Techniques optimizing the number of processors to schedule multi-threaded tasks. In *24th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 321–330, 2012.

[124] Luís Nogueira and Luis Miguel Pinho. Server-based scheduling of parallel real-time tasks. In *10th ACM international conference on Embedded software (EMSOFT)*, pages 73–82, 2012.

[125] OpenMP. OpenMP Application Program Interface v4.0, July 2013. http://http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

[126] Luigi Palopoli, Daniele Fontanelli, Nicola Manica, and Luca Abeni. An analytical bound for probabilistic deadlines. In *Real-Time Systems (ECRTS), 24th Euromicro Conference on*, pages 179–188, 2012.

[127] Miloš Panić, Eduardo Quiñones, Pavel G Zaykov, Carles Hernandez, Jaume Abella, and Francisco J Cazorla. Parallel many-core avionics systems. In *14th International Conference on Embedded Software (EMSOFT)*, page 26, 2014.

[128] Risat Mahmud Pathan. Schedulability analysis of mixed-criticality systems on multiprocessors. In *Real-Time Systems (ECRTS), 24th Euromicro Conference on*, pages 309–320, 2012.

[129] Rodolfo Pellizzoni, Patrick Meredith, Min-Young Nam, Mu Sun, Marco Caccamo, and Lui Sha. Handling mixed-criticality in soc-based real-time embedded systems. In *7th ACM international conference on Embedded software (EMSOFT)*, pages 235–244, 2009.

[130] Cynthia A Phillips, Cliff Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation. In *ACM symposium on Theory of computing*, pages 140–149, 1997.

[131] Kirk Pruhs, Julien Robert, and Nicolas Schabanel. Minimizing maximum flowtime of jobs with arbitrary parallelizability. In *Approximation and Online Algorithms - 8th International Workshop, (WAOA)*, pages 237–248, 2010.

[132] Kirk Pruhs and Clifford Stein. How to schedule when you have to buy your energy. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, 13th International Workshop, APPROX 2010, and 14th International Workshop, RANDOM 2010, Barcelona, Spain, September 1-3, 2010. Proceedings*, pages 352–365, 2010.

[133] Arun Raman, Hanjun Kim, Taewook Oh, Jae W Lee, and David I August. Parallelism orchestration using DoPE: The degree of parallelism executive. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, volume 46, pages 26–37, 2011.

[134] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* O'Reilly Media, 2010.

[135] Shaolei Ren, Yuxiong He, Sameh Elnikety, and Kathryn S McKinley. Exploiting processor heterogeneity in interactive services. In *10th International Conference on Autonomic Computing (ICAC)*, pages 45–58, 2013.

[136] Julien Robert and Nicolas Schabanel. Non-clairvoyant scheduling with precedence constraints. In *19th ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 491–500, 2008.

[137] Abusayeed Saifullah, David Ferry, Jing Li, Kunal Agrawal, Chenyang Lu, and Chris Gill. Parallel real-time scheduling of dags. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3242–3252, 2014.

[138] Abusayeed Saifullah, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Multi-core real-time scheduling for generalized parallel task models. *Real-Time Systems*, 49(4):404–435, 2013.

[139] Tao B Schardl, Bradley C Kuszmaul, I Lee, William M Leiserson, Charles E Leiserson, et al. The cilkprof scalability profiler. In *27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 89–100, 2015.

[140] Bianca Schroeder and Mor Harchol-Balter. Web servers under overload: How scheduling can help. *ACM Trans. Internet Technol.*, 6(1):20–52, 2006.

[141] Daniel D Sleator and Robert E Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[142] Dario Socci, Peter Poplavko, Saddek Bensalem, and Marius Bozga. Mixed critical earliest deadline first. In *Real-Time Systems (ECRTS), 25th Euromicro Conference on*, pages 93–102, 2013.

[143] Anand Srinivasan and James H Anderson. Efficient scheduling of soft real-time applications on multiprocessors. In *Real-Time Systems (ECRTS), 15th Euromicro Conference on*, pages 51–59, 2003.

[144] Anand Srinivasan and Sanjoy Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Information Processing Letters*, 84(2):93–98, 2002.

[145] Hang Su, Nan Guan, and Dakai Zhu. Service guarantee exploration for mixed-criticality systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), IEEE 20th International Conference on*, pages 1–10, 2014.

[146] Ola Svensson. Conditional hardness of precedence constrained scheduling on identical machines. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 745–754, 2010.

[147] Olivier Tardieu, Haichuan Wang, and Haibo Lin. A work-stealing scheduler for x10's task parallelism with suspension. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2012.

[148] Marc Tchiboukdjian, Nicolas Gast, Denis Trystram, Jean-Louis Roch, and Julien Bernard. A tighter analysis of work stealing. *Algorithms and Computation*, pages 291–302, 2010.

[149] Eric Torng and Jason McCullough. Srpt optimally utilizes faster machines to minimize flow time. *ACM Transactions on Algorithms*, 5(1), 2008.

[150] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *28th IEEE Real-Time Systems Symposium (RTSS)*, pages 239–243, 2007.

[151] Qi Wang and Gabriel Parmer. Fjos: Practical, predictable, and efficient system support for fork/join parallelism. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), IEEE 20th*, pages 25–36, 2014.

[152] Adam Wierman and Bert Zwart. Is tail-optimal scheduling possible? *Operations research*, 60(5):1249–1257, 2012.

[153] Gerhard J. Woeginger. On-line scheduling of jobs with fixed start and end times. *Theor. Comput. Sci.*, 130(1):5–16, 1994.

[154] Thomas Y. Yeh, Petros Faloutsos, and Glenn Reinman. Enabling real-time physics simulation in future interactive entertainment. In *ACM SIGGRAPH Symposium on Videogames*, Sandbox '06, pages 71–81, 2006.

[155] Jeonghee Yi, Farzin Maghoul, and Jan Pedersen. Deciphering mobile search patterns: A study of Yahoo! mobile search queries. In *ACM International Conference on World Wide Web (WWW)*, pages 257–266, 2008.

[156] Zircon Computing. Parallelizing a computationally intensive financial R application with zircon technology. In *IEEE CloudCom*, 2010.