Memory Efficient Scheduling for Multicore Real-time Systems

by

Ahmed Alhammad

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2016

Author's Declaration

Statement of Contributions

This dissertation is split into six chapters. In Chapter 2, we provide background material and discuss the related work. In particular, we first discuss the memory subsystem and focus on the relevant issues that may influence the predictability of timing behavior. Second, a survey of related work is provided for the problem of memory bus contention including the co-scheduling approach. For the latter, we discuss the limitations that may arise in some practical settings. We conclude Chapter 2 by providing an overview of multiprocessor real-time scheduling theory for both sequential and parallel tasks. Some of the discussion about cache management in Section 2.1.1 is reprinted, with permission, from [62] in which I am a co-author.

We recall that the main contribution of this dissertation is integrating memory time into multiprocessor real-time scheduling for both sequential and parallel tasks. In Chapter 3 and 4, we focus on sequential tasks while in Chapter 5 we focus on parallel tasks. The main findings of these three chapters have been disseminated in four published papers in which I am the main author [4, 7, 6, 5]. In particular, Chapter 3 extends the co-scheduling approach [108], originally with two phases of load and computation, by adding an extra phase to unload modified data back to main memory. As well, we show how to dynamically schedule the new 3-phase execution model in a multitasking environment. We further propose an algorithm to globally schedule 3-phase sequential tasks on a multicore processor. Since the schedulability test is as important as the scheduling algorithm, we also provide a schedulability analysis for this algorithm. In particular, a new concept of *schedule hole* is introduced to account for memory time. Furthermore, an analysis method is proposed for the same algorithm where each task is split into multiple segments. This scheme is necessary for some large tasks that cannot fit inside the local memory of one core at one time. We note that sections 3.4 and 3.6 are reprinted, with permission, from [4]. The notations and the figures have been modified to harmonize with the rest of the dissertation.

While the previous chapter assumes that processor cores are stalled while loading from main memory, Chapter 4 proposes a new memory efficient algorithm to globally schedule 3-phase sequential tasks on a multicore processor equipped with a DMA component to further enhance the memory utilization. In this chapter, the DMA is used to load one task while the core is busy executing another task. Since the system comprises two types of active components (processor cores and DMA), a new concept of *scheduling interval* is introduced to account for the workload of 3-phase tasks on processor cores and the DMA. Sections 4.1 to 4.3 of this chapter are reprinted, with permission, from [7]. Again, the notations and the figures have been modified.

In Chapter 5, a novel method is proposed to trade in processor cores for memory bandwidth. This method works for parallel tasks that need more than one core to execute. In particular, a federated scheduling scheme is adopted in which each parallel task is assigned a dedicated number of cores and memory bandwidth. Unlike the co-scheduling approach, we assume no arrival pattern of memory requests (no load and unload phases) and the scheduling algorithm of each parallel task on its cluster of cores can be any dynamic work-conserving algorithm. To further enhance the execution performance, we propose a static algorithm to schedule parallel tasks on their dedicated cluster of cores. The objective of this algorithm is to minimize the *makespan* of real-time parallel tasks. In addition, the co-scheduling approach is used to enhance the memory time. Sections 5.8 and 5.9 are reprinted, with permission, from [5], and the rest of the sections are reprinted, with permission, from [6].

We conclude this dissertation in Chapter 6 and provide directions to carry out relevant research in the future.

Abstract

Modern real-time systems are becoming increasingly complex and requiring significant computational power to meet their demands. Since the increase in uniprocessor speed has slowed down in the last decade, multicore processors are now the preferred way to supply the increased performance demand of real-time systems.

A significant amount of work in the real-time community has focused on scheduling solutions for multicore processors for both sequential and parallel real-time tasks. Even though such solutions are able to provide strict timing guarantees on the overall response time of real-time tasks, they rely on the assumption that the worst-case execution time (WCET) of each individual task is known. However, physical shared resources such as main memory and I/O are heavily employed in multicore processors. These resources are limited and therefore subject to contention. In fact, the execution time of one task when run in parallel with other tasks is significantly larger than the execution time of the same task when run in isolation. In addition, the presence of shared resources increases the timing unpredictability due to the conflicts generated by multiple cores. As a result, the adoption of multicore processors for real-time systems is dependent upon solving such sources of unpredictability.

In this dissertation, we investigate memory bus contention. In particular, two main problems are associated with memory contention: (1) unpredictable behavior and (2) hindrance of performance. We show how to mitigate these two problems through scheduling. Scheduling is an attractive tool that can be easily integrated into the system without the need for hardware modifications. We adopt an execution model that exposes memory as a resource to the scheduling algorithm. Thus, the theory of real-time multiprocessor scheduling, that has seen significant advances in recent years, can be utilized to schedule both processor cores and memory. Since the real-time workload on multicore processors can be modeled as sequential or parallel tasks, we also study parallel task scheduling by taking memory time into account.

Acknowledgements

First of all, I would like to thank my advisor Rodolfo Pellizzoni. It has been an honor to be his first PhD student. I really appreciate all his efforts of time and ideas that made my PhD productive and fruitful. His dedication and contiguous enthusiasm on my research have been motivational for me to proceed during the tough times.

I also would like to thank my committee members, Marko Bertogna, Sebastian Fischmeister, Hiren Patel and Bill Cowan for their valuable feedback and suggestions.

Special thanks go to my colleague Saud Wasly for his great support in many occasions during my study. In particular, the times before the comprehensive exam and the PhD defense. I am also grateful for my meetings with him to brainstorm new ideas and discuss the small details of my research.

My thanks extend to the rest of my research group that I sadly did not have a chance to write papers with them, but their help and feedback are highly appreciated. I would also like to express my thanks and appreciation to my co-authors Giovani Gracioli and Renato Mancuso.

I am grateful to King Saud University for their financial support. Without their support, I probably would not have the opportunity to attend a great school like the University of Waterloo. I am also grateful for Saudi Cultural Bureau in Canada for being always available to answer my questions and provide support.

My time at Waterloo was made enjoyable due to the many friends that became a part of my life. I would like to thank the Saudi Club at Waterloo for organizing the weekly gatherings and outdoor activities. I am grateful for the time spent with my friends cycling and kayaking the Grand river.

Foremost, I thank my parents Haya and Ibrahim for their endless love, optimism and being with me at all times with their prayers. You always inspirit me to do the best in my life. My great thanks go to my wife Hajer for her love, patience and trust that I was doing the right thing. I could not have finished my PhD without you.

Dedication

To my wife Hajer and my son Ibrahim

Table of Contents

List of Figures

Chapter 1

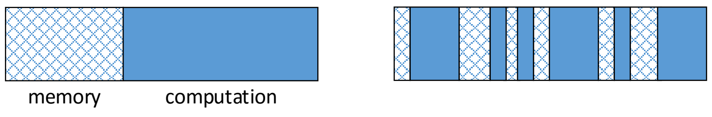
Introduction

Modern real-time applications are becoming increasingly complex and asking for more computational power. They expand beyond simple control loops to include image and video processing, advanced I/O and distributed coordination. In order to meet the increasing demands of real-time applications, a higher performance computing platform is needed. Unfortunately, *uniprocessor* design hits a fundamental problem: heat dissipation. The faster the processor runs; the more heat it generates. In fact, the development of a new uniprocessor design from Intel, the world's largest chip maker, has been canceled in 2004 due to heat dissipation and extreme power consumption. Two years later, Intel released the first dual-core processor which marked a trend in current technology to replicate multiple processors in order to increase the processing power. Thus, real-time applications are inevitably forced to use *multicore* processors to supply their increasing processing demand. Moreover, the speed of memory subsystem is equally important for the overall execution performance. As a result, multicore processors need to use architectural features such as caches or scratchpads to bridge the gap between the processing power and memory speed.

Most hard real-time systems are safety critical in which system failure leads to severe damage or loss of life. These systems are required to respond to events within a limited time-span referred to as a *deadline*. Thus, hard real-time systems require a timing validation known as *schedulability analysis* to guarantee, before executing the system, that system's tasks will complete by their deadlines. However, the schedulability analysis needs as input the **worst-case execution time (WCET)** of each task. The estimation of WCET has to be *safe*, i.e., above or equal any possible execution time, and should be *tight*, i.e., close to the actual execution time. It is worthwhile to note that in real-time systems there is no advantage in completing the task before its deadline; instead, the matter is how to precisely estimate its WCET. Often, overestimates can cause a task to be unschedulable even though the existing system has the capacity to accommodate its demands. A related property is *predictability* of temporal behavior. Instead of having Boolean (black and white) definition, [10] defines predictability as the ratio between the best-case and the worst-case behavior. That is, a component with constant-time behavior is 100% predictable, and this percentage decreases as the difference between the best-case and the worst-case behavior increases.

Still, estimating the WCET of any task is difficult because it may vary from execution to execution for several reasons. First, the control flow of the application may contain different conditional branches in which their flow depends on the input and these branches have different execution times. Second, the hardware platform usually has stateful components such as caches or pipelines. The state of these components affects the application execution time as the component state depends on the execution history. Third, a core inside multicore processor is not an independent entity but rather shares physical resources with other cores such as last-level cache and main memory. In single core processor, concurrent tasks are executed sequentially, even though virtual parallelism is permitted. The sequential execution implies that two tasks can never access two shared resources simultaneously. However, the situation is different in multicore processor. Two tasks can run simultaneously on different cores and access shared resources at the same time. It has been shown in [148] that the execution time of one application can vary significantly due to shared resource contention. In particular, an experiment is conducted on Intel 4-core processor with SPEC CPU2006 applications. The results show that the execution time of one application can increase by 56% compared to the execution time of the same application when run in isolation. Resource contention has also been acknowledged by certification authorities [55], and it represents a source of concern for the use of multicore processors in avionics systems.

Previous research has mainly focused on cache-space contention [62]; however, the results in [148] indicate, through extensive experimentation on real systems, that memory bus contention is also significant in influencing the execution time of applications. Memory requests of processor cores are mediated by an *arbiter*. Once one core gains access to main memory, all other cores are blocked, and hence experience delays. Thus, memory contention introduces additional blocking times during execution. These additional delays must be accounted for in the response time of real-time tasks. In this dissertation, we will focus on this problem that affects the predictability of WCET of real-time tasks.



(a) (b)

Figure 1.1: In (a), memory accesses are concentrated at the beginning while in (b) they are spread across task's execution.

1.1 Memory Bus Contention

The classical multiprocessor real-time scheduling theory focuses on processor cores in which the scheduling algorithm knows nothing about the resulting memory latency of its schedule. It either assumes an idealized multicore processor where tasks access memory without contention or the contention delay is integrated into the WCET of tasks. Prior work has proposed analysis-based methods to account for these memory delayes [120, 36, 127, 109, 125, 43]. These methods rely on these two pieces of information to derive the memory delay: the arbitration policy of the hardware and the memory access pattern of real-time tasks. However, these approaches have some drawbacks. First, it is very hard to capture the task's memory access pattern. In a cache-based system, for instance, memory accesses are results of cache misses, and these misses depend on the program execution path as well as the replacement policy and the current state of the cache. Second, some platforms employ unfair arbitration policies in which some requests are favored against others to utilize the memory bandwidth or, in some cases, the arbitration policy is not documented. Third, the notion of task priority is not reflected in hardware arbiters. Thus, the analysis often assumes that tasks on other cores have higher priority memory requests. Finally, most of these works analyze individual tasks or assume tasks repeat a simple offline schedule. To avoid these drawbacks, we adopt in this dissertation the co-scheduling approach [108] which can provide a tighter upper-bound on memory time while using the existing solutions for multiprocessor real-time scheduling.

1.2 The Co-Scheduling Approach

In this approach, each task is split into two phases: memory and computation as shown in Figure 1.1(a). During a memory phase, the code and data of each real-time task is loaded from main memory to local memory of processor core before execution. Then, the core, during the computation phase, executes the task out of its local memory without accessing main memory. This ensures that the memory accesses are concentrated at the beginning of the task, i.e., simple and clear access pattern. In contrast, memory accesses, if not controlled, are very hard to predict as in Figure 1.1(b). As mentioned early, memory requests are results of cache misses, and these requests depend on the execution path of the program and the state of the cache. In addition, by explicitly exposing the memory demand of each task to the scheduling algorithm, the toolbox of multiprocessor real-time scheduling can be leveraged to schedule both memory and processor cores, hence the name co-scheduling. This approach has been demonstrated on real hardware platforms in [119, 53, 30]. A detailed discussion about the implementation part of this approach is presented in Section 2.2.2.

Prior work has looked into the co-scheduling approach for different reasons. First, it is used in [108] to de-conflict the traffic to main memory between tasks running on single core and I/O devices. Second, the work in [15] uses **time division multiple access (TDMA)** to partition the memory bandwidth between cores. To utilize the slots of a memory schedule, the priority of memory phases is increased over computation phases during the allotted slots. Thus, a memory slot is never wasted while there is a ready memory phase. Third, it is used in [99] to speedup the transfer time of memory. The rationale behind this work is that it is generally easier to improve memory bandwidth than it is to reduce memory latency. In fact, cache designers tend to increase block size to take advantage of high memory bandwidth [70]. This advantage is clear with memory phases that span multiple cache lines. Moreover, **direct memory access (DMA)** engines are engineered to perform efficient back-to-back transfers.

In this dissertation, however, we adopt the co-scheduling approach as a way to (1) avoid memory contention in a multicore processor and (2) enhance memory utilization. First, memory contention is avoided by allowing only one memory phase to be active in the system at any time. In other words, the scheduler from software level ensures in proactive manner that tasks do not interfere for access to main memory. Second, the memory utilization is improved when the execution of one memory phase is overlapped with computation phases of other tasks. In general purpose computing, a large number of techniques, such as out-of-order execution and prefetching, are employed to harness such overlap at the level of one task or between different tasks (hardware multi-threading). That is, while waiting for main memory to return the requested data, the processor is busy executing other instructions. Although these techniques improve the average case performance, it is difficult to extract worst-case guarantees. In contrast, we focus this dissertation on techniques that are suitable for hard real-time systems with guaranteed bounds. Three

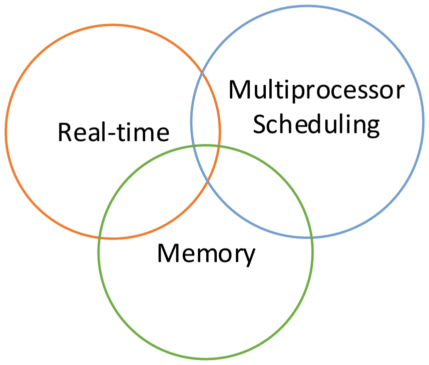


Figure 1.2: Dissertation theme and scope.

levels to overlap the memory phases can be distinguished: (1) between tasks on different cores, (2) between tasks on the same core and (3) within one task. We will focus in this dissertation on the first two and ignore the last one. The reason is that in order to exploit the overlap within one task, a compiler analysis is needed to insert prefetch commands in the code at the right places. In contrast, we capture the overlap between tasks from a higher level: the scheduler, without relying on compiler-level techniques.

As we mentioned earlier, the co-scheduling approach has the advantage to speedup the memory transfer time. Another advantage is that since one core is given the whole access to main memory during memory phase, this avoids the interleaving effect of other requests coming from other cores. This can benefit platforms with hardware arbitration policy that is unfair. For example, with a fair arbiter such as **round-robin (RR)**, the inflated memory time (due to contention) for one memory request can be bounded by multiplying the memory latency with m, the number of cores in the system, assuming each memory request is preceded by $m-1$ other requests in the worst case. However, with an unfair arbiter, the inflated memory time can exceed m. For example, **dynamic random access memory (DRAM)** controllers are designed to utilize the memory bandwidth; thus, they may favor some requests that are ready to be served over others using polices such as **first-ready first-come first-serve (FR-FCFS)**.

1.3 Dissertation Theme and Problem Statement

The main focus of this dissertation is scheduling real-time applications on multicore processors by taking memory time into account. These three aspects are depicted in Figure 1.2. In particular, *multiprocessor scheduling* means spatial and temporal assignment of tasks on multiprocessor, *real-time* introduces the notion of time and predictable execution, and *memory* considers the memory time.

The problem of this dissertation can be stated as follows. The classical multiprocessor real-time scheduling theory focuses on processor cores and ignores memory. However, the *contention* for access to main memory can significantly change the execution time of real-time tasks on multicore processors. A real-time task, like any other computer program, can be abstracted as a stream of memory and CPU instructions. Similar to CPU instructions, we argue that these memory instructions (memory requests) have to be scheduled to avoid the contention conflicts and enhance their execution by hiding their latency. To achieve these goals, the real-time tasks have to be split into two phases: memory and computation.

Chapter 2

Background and Related Work

This chapter provides the essential background material and reviews the related work. The multiprocessor system we consider in this dissertation is the **symmetric shared-memory multiprocessor (SMP)**. Here, all processors have equal access to one shared main memory and can exchange data through this shared memory. Shared memory design has gained wide acceptance due to its simplified programming and natural transition from uniprocessors. Furthermore, as Moore's prediction is expected to continue in the next few years, the trend in current technology is to put more processors on the same chip. These processors inside the same chip are called cores. Therefore, the term multiprocessor also includes multicore or **chip multiprocessor (CMP)**.

There are two topics that are directly related to the subject of this dissertation: memory contention and real-time scheduling. We recall that the goal of this dissertation is to solve memory contention problem through scheduling. Thus, we first overview the memory subsystem of SMP architectures. Then, we discuss the memory contention problem. Lastly, we overview the theory of multiprocessor real-time scheduling.

2.1 Memory Subsystem

A single high-performance processor can generate two data memory requests and fetch four instructions per clock cycle [70]. For example, Intel Core i7 with 3 GHz clock rate can generate 6 billion 64-bit data requests per second, in addition to fetching 12 billion 32-bit instructions per second; this is a total peak bandwidth of 96 GB/s. In contrast, the peak bandwidth of DDR3 memory is 17 GB/s (only 18 %). The situation in multiprocessor

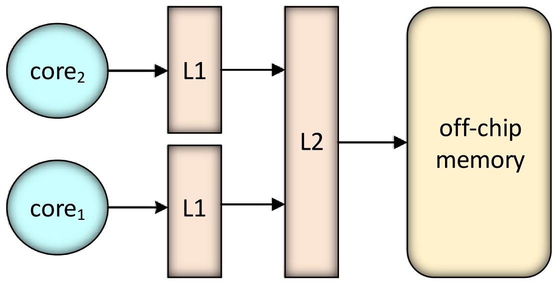


Figure 2.1: The levels of memory hierarchy.

systems is even worse. In fact, the number of memory requests per second grows as the number of processors increases. Theoretically, four Intel Core i7 can generate a peak bandwidth of 384 GB/s which is far more than what current DRAM technology can supply.

In order to mitigate the huge performance gap between multiprocessor and main memory, different techniques are often used such as multilevel caches. While these techniques enhance the average-case performance, they introduce unpredictable behavior. It is more challenging to support real-time applications on systems with the presence of caches. In what follows, we discuss the hierarchy levels in a typical memory subsystem as shown in Figure 2.1. In particular, we will focus on the relevant issues that may influence the predictability of timing behavior. We note that the cache levels can be larger than two in some platforms but we restrict our discussion in this section to two levels: one private and one shared.

2.1.1 Cache Memory

Predicating cache-related delays is a typical component of WCET analysis [138]. Several well-developed cache analysis techniques have been proposed for single-core processors. These techniques analyze the interference due to intra-task and intra-core cache conflicts. The latter is known as **cache related preemption delay (CRPD)**. The CRPD focuses on cache reload overhead due to preemption while the intra-task analysis focuses on the cache conflicts within the same task assuming non-preemptive execution.

In existing multicore processors [73], the last-level cache is typically shared by multiple cores. This design has several merits such as increasing the cache utilization, reducing the complexity of cache coherency and facilitating a fast communication medium between cores. However, it is extremely difficult to accurately determine the cache miss rate because the cache content depends on the size, organization and replacement strategy of the cache in addition to the order of accesses. Shared caches in multicore processors are similar to caches in single core processors in that they all have inter/intra-task interference. In addition, when multiple cores share a cache, they can evict each other cache lines, resulting in a problem known as inter-core interference.

Unfortunately, single-core cache timing analysis techniques are not applicable for multicore with shared caches. Inter-core interference is caused by tasks that can run in parallel and this requires analyzing all system's tasks. The analysis of non-shared caches has been already considered as a complex process and extending it to shared caches is even harder. In fact, the researchers in the community of WCET analysis [131] seem to agree that "*it will be extremely difficult, if not impossible, to develop analysis methods that can accurately capture the contention between multiple cores in a shared cache*".

Despite this challenge, few works have been proposed to address the problem of shared caches. These techniques are applicable for simple architectures and statically scheduled tasks. The first work that studies the analysis of shared caches in multicore processors is proposed in [141]. This work assumes a system with two tasks simultaneously running on two cores with direct-mapped shared instruction cache. Later, cache conflict graphs were used to capture the potential inter-core conflicts [147]. The work in [85] improves upon [141] by exploiting the lifetime information of tasks and bypassing the disjoint tasks (tasks that cannot overlap at run-time) from the analysis. This work assumes a task model where all tasks are synchronized. Clearly, for systems with dynamic scheduling, it will be extremely difficult to identify the disjoint tasks. Other research [66] proposes to bypass the shared cache for single-usage cache lines to avoid inter-core conflicts and therefore improve the timing analysis. For systems where tasks are allowed to migrate between cores, **cache related migration delay (CRMD)** has been studied in [67]. This work estimates the number of cache lines that can be reused from the L2 shared cache when a task migrates from one core to another. Due to the lack of analysis techniques for multicore platforms with multiple tasks scheduled dynamically, an empirical study has been proposed in [23] to evaluate the impact of **cache-related preemption and migration delays (CPMD)**. An interesting result is that delays related to preemption and migration do not differ significantly on a heavily loaded system.

Furthermore, cache memories in single core processors have three types of misses: compulsory, conflict and capacity. Beside these three types of misses, private caches in multicore processors introduce another type: coherency misses. They pose another challenge because shared variables, that may simultaneously exist in several caches, are automatically updated by the cache coherency hardware in a hidden way (without explicit instructions). If frequent shared memory accesses occur, the WCET can be largely affected [63]. In general, private caches introduce two problems in multicore processors [29] that could affect the execution of real-time tasks: (1) access serialization (atomic) to shared cache lines. When a core writes to a shared cache line, the write is not considered complete until all shared cache lines are invalidated. Similarly, before a core reads a dirty cache line written by another core, the cache line has to be written back to main memory and the source core has to change the cache line state into shared. (2) A large number of coherency messages are sent across the interconnection network to keep the shared data between multiple processors coherent. This could saturate the interconnection network and cause further delays that could reduce the parallel processing gain and more importantly the time predictability for real-time systems. Depending on the application, shared data is intrinsic and cannot be avoided. To the best of our knowledge, no existing static WCET analysis technique is able to account for the effects of coherency misses.

Another line of research has suggested to randomize the cache and employ a stochastic analysis to determine the cache miss rate [115]. In particular, **probabilistic timing analysis (PTA)** has been proposed as an alternative to conventional timing analysis that can be highly pessimistic for the worst-case [34]. PTA provides **probabilistic WCET (pWCET)** estimates that can be exceeded with a given probability. That is, a pWCET with an associated low probability, say 10^{-15}, means that the probability for the execution time to exceed this pWCET is 10^{-15}. However, PTA techniques require the execution times to have a probability of occurrence that is independent and identically distributed. These two features are essential to allow using random variables and apply statistical methods for analyzing the system. At the cache level, caches with LRU replacement policy cannot be used because the result of each memory access is dependent on the previous accesses. A fully-associative cache with random replacement is one example that can be used for PTA. Along this line of research, PTA has been applied to a single level cache [77], multilevel caches [78] and for CRPD [48]. Recently, PTA was used for shared caches to estimate the inter-core cache conflicts [130].

In contrast to timing analysis techniques where caches are used without restrictions, the approach of managed caches has the advantage to avoid complex analysis methods for estimating the cache behavior [62]. Indeed, the time predictable architecture in [106] proposes a statically-partitioned L2 cache to avoid the inter-core cache conflicts. In addition, managed caches can be used in situations where the static analysis cannot be used, for example, the case where the cache replacement policy is not documented. On the other hand, while managing the cache space provides a timing isolation between tasks, the reduced cache space may impact the task execution time [8]. The basic idea of managed caches is to give each task a portion of cache space so that no other task is allowed to have access. The cache memory can be seen as a two dimensional array in which the columns are the cache ways, and the rows are the cache sets. The shared cache can be partitioned in sets (rows), ways (columns) or individual cache lines.

Page coloring is a technique used to map virtual pages to physical pages (frames) by taking into account the cache structure. It can be employed to implement cache partitioning in software by re-arranging the physical addresses [135]. *Cache locking* is a mechanism that hooks some cache lines from being evicted until explicitly unlocked [94]. It has the advantage of controlling the partition at cache line granularity. Cache locking has two different schemes: either *static* or *dynamic*. With a static scheme, locked cache lines remain unchanged throughout execution, i.e., locked once. On the other hand, dynamic locking allows cache lines to be reloaded on the fly. The limitations of static locking manifest when an application has multiple hot regions and limited cache space. With dynamic locking, this limited cache space can be utilized multiple times.

These cache portions can be assigned to individual tasks or processors. Processor-based partitioning allows tasks to enjoy more cache space; in contrast, task-based partitioning avoids cache reloading at preemption. When combined with cache locking, many different schemes can be obtained. Suhendra and Mitra [131] explore these different schemes and evaluate their effects on the worst-case performance. They conclude that processor-based partitioning is better than task-based partitioning independent of the locking scheme.

As we stated above, there is no analysis method available to account for cache coherency delays. Thus, caches are not used for shared data in hard real-time systems [114]. Recently, the work in [113] proposes **on-demand cache coherent (ODC^2)**, a hardware approach that relies on software support to guarantee coherent accesses to data inside critical regions. Thus, ODC^2 allows applications to use caches for shared data.

2.1.2 Scratchpad memory

An alternative to cache memory is **scratchpad memory (SPM)**. The advantages include reduced power consumption and predictable behavior [137]. The SPM is a special static RAM placed close to the processor (on-chip, similar to L1 cache). The address space of the SPM is mapped into a predefined memory address of the processor. Unlike cache memory, the SPM has to be explicitly managed. In other words, the memory blocks have to be moved in software from main memory and copied into the SPM before being used. Thus, SPM is highly predictable in the sense that it has one access latency compared to caches with two different latencies for cache hit and miss.

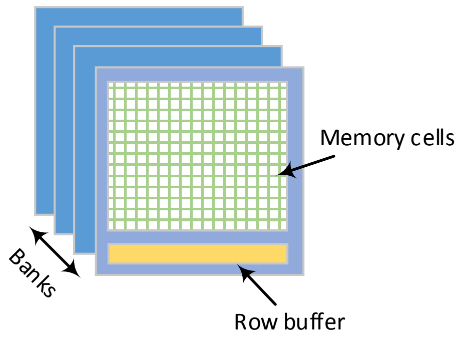


Figure 2.2: Multibank DRAM.

There has been a significant amount of work in literature that proposes solutions to dynamically manage the SPM for one task [50, 90, 13, 52, 134] by reusing the available space over task's execution. These works propose solutions for managing task's code and data including stack and heap. Since SPM is not transparent with respect to address translation, management schemes have to impose constraints on analyzable code; in particular, memory aliases must be statically resolved, since otherwise the management scheme risks loading the same data into two different positions in the SPM.

In this dissertation, the focus is co-scheduling memory and computation phases in which the memory phase is the time to load a task from main memory to local memory. We use the term *local memory* to include either cache memory or SPM. As we discuss above, the literature is rich of software and hardware techniques that can be employed to let cache memory behave like SPM without conflict misses.

2.1.3 Off-chip Memory

The off-chip memory is often made of cost-effective technology such as DRAM. It is a 3-dimensional array of memory cells organized as banks, rows and columns as in Figure 2.2. DRAM accesses are controlled by a *memory controller* which can be seen as a mediator between processor cores or last-level cache on one side and the DRAM chip on the other side. It translates read/write memory requests into corresponding commands and schedules them while satisfying the timing constraints of a DRAM chip. Each memory bank has its own row buffer which acts as a cache for the DRAM. Before data can be transferred, the target row has to be opened (moved to the row buffer). A memory request that hits the row buffer is faster than a memory request that needs to close the current row buffer and open a new one. Thus, DRAM access latency is variable and depends on the previous accesses. It is a hardware component that holds a state information (like cache memory) which may affect the application timing behavior. In addition, an arbitration policy like FR-FCFS is often used to utilize DRAM bandwidth by favoring requests that are ready to be served over other requests. Furthermore, DRAM is divided into banks to increase efficiency by interleaving memory requests. In other words, while data is transferred for a bank, the other bank can be activated. In fact, this parallelism can be exploited to enhance the application execution performance. However, one has to trade-off between performance and predictability. If memory pages of one core are mapped into the same bank, an isolation between different cores can be achieved. On the other hand, an application can gain higher memory bandwidth when memory pages are interleaved between different banks.

Due to the dynamic nature of DRAM controllers, they are ill-suited for hard real-time systems. Thus, several designs of predictable memory controllers have been proposed as in [2, 61, 68]. These designs change the internals of memory controller such as the page policy and the command scheduler to produce tighter bounds. In this dissertation, we look at off-chip memory as a black-box with constant access latency similar to most of the related work we discuss in the next section. However, as we mention in Section 1.2, the memory phase of co-scheduling approach has the advantage to exploit the DRAM structure to greatly reduce the access latency without the involved complexity of other work to derive bounds on memory latency.

2.2 Memory Bus Contention

The memory bandwidth in multicore processors is shared and therefore subject to *contention*. At the hardware level, the memory requests are mediated by an *arbiter*. Based on the arbitration policy, the access latency of one request can be delayed by other requests. The research community has recognized memory contention as a significant challenge that may affect both performance and predictability. An experimental evaluation on the accuracy of some task scheduling algorithms is done in [128]. The results show that these scheduling algorithms show a very poor accuracy where the real execution time is most often multiple of the estimated one. The problem stems from the assumption that all processors are fully connected in a way that communications can proceed concurrently without contention and without the processor involvement in the communication. The same authors as in [128] suggest that the task model should be modified to include the

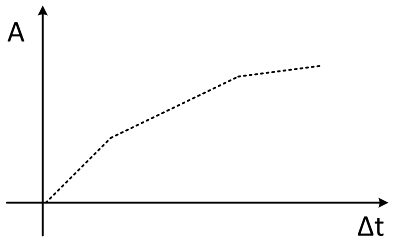


Figure 2.3: An arrival curve specifies the maximum number of memory requests that can be generated within a window of time Δt.

processor involvement in the communication and reflect the fact that the communication subsystem is not contention-free [129]. Furthermore, the work in [109] shows that the task execution time due to memory contention can increase linearly with number of processors.

The topic of memory *bus contention* has received a significant attention in recent years. [106] proposes RR arbitration for memory bus. In this case, the memory request of one task can be delayed by at most $m - 1$ other requests of other cores. Thus, the inflated memory time of one request can be bounded as $m \times \sigma$ where σ is the memory latency of single request without interference. In [86], the authors propose a memory wheel where each task is given a window of time to access memory. We note that RR, unlike memory wheel, is work-conserving and there is no wasted time. Consider a memory with access latency of $\sigma = 20$ cycles and a system with $m = 4$ cores. With memory wheel, a memory request can take $19 + 4 \times 20 = 99$ cycles if this request arrives one clock cycle after the beginning of its wheel window. In contrast, with RR arbitration, the memory time of a request, regardless of its arrival time, is always bounded by $4 \times 20 = 80$ cycles.

The work in [109] models each task as a set of super-blocks and derives an arrival curve for these super-blocks. Then, it introduces an analysis method to compute an upper bound on memory contention delay by assuming RR and **first-come first-serve (FCFS)** arbitration at the hardware level. The concept of arrival curves is also used in [125] to model the memory load of each processor. As shown in Figure 2.3, the arrival curve $A(\Delta t)$ is a cumulative function that specifies the maximum number of memory requests that can arrive within a window of time. Thus, many conflicts are guaranteed to be eliminated with large request distances (time windows). A measurement-based method is proposed in [43] to capture the arrival pattern of memory requests. This work assumes a partitioned non-preemptive scheduling at the task-level and unspecified work-conserving arbitration at the hardware level.

Recently, a framework to analyze memory bus contention is proposed in [44]. The problem is split into two steps: hardware arbitration dependent and independent. In the first step, a bus availability model is derived using worst-case and best-case start times. In the second step, task request-profile is used to model the traffic generated by tasks. To tighten the traffic distribution, the task request-profile is split into equal-size temporal regions where each region is characterized by the maximum number of memory requests. Using task request-profile and bus availability model, an algorithm is proposed to compute the maximum memory bus contention. [44] idles processors if tasks execute for less than their WCET to ensure that the number of memory requests within a time window is not higher at run-time than the computed one at design time. In contrast, we allow tasks in our design to execute for less than their WCET; thus, the system can accommodate soft real-time tasks to utilize the over-provisioning of hard real-time tasks.

Another line of research uses TDMA arbitration [120, 126, 36]. In this case, the bus is divided into time slots and these slots are assigned to cores. Thus, each core is associated with slots with known start and finish times. This distribution of time slots to cores is called, *bus schedule*. Access to main memory is allowed for one core in its assigned slot. The memory or bus arbiter stores the bus schedule in a lookup table and grants access to cores accordingly. If a core requests memory at a time slot that belongs to another core, it will be delayed until its next slot. Clearly, this scheme determines exactly when a core will be granted the bus, thus, it allows analyzing the memory time for each core separately without the interference effect of other cores. Given a bus schedule, the WCET of one task can be determined using static WCET analysis. This requires the locations of memory requests in time to be known. In other words, there should be a guaranteed alignment between memory requests and bus schedule. A drawback of this approach is that the memory locations of a task can only be specified for a particular hardware architecture by using its micro-architectural model to account for the timing effects of underlying components such as the pipeline. A thorough discussion of this topic can be found in [37].

The work in [127] also uses TDMA to arbitrate for main memory. However, instead of knowing the exact locations of memory requests, the task is modeled as a sequence of super-blocks and each super-block is divided into three phases: acquisition phase, execution phase and replication phase. Access to main memory is limited to an acquisition phase at the beginning and a replication phase at the end. This work then proposes an analysis framework to compute the **worst case response time (WCRT)** of each task for a given TDMA schedule.

Other research considers priority based arbitration. The work in [3] proposes credit-based arbitration scheme. Similarly, the work in [103] proposes an arbitration scheme based on weighted fair queuing theory at the granularity of memory requests. However, these two schemes involve large overhead when implemented in hardware.

Yun *et al.* [146] propose a memory bandwidth reservation system called Memguard that works at the operating system level and can reserve memory bandwidth for a specific core. Furthermore, a **hardware performance counter (HPC)** is used to measure the resource usage and enforce it using a run-time monitor. The authors also use a resource reclaiming approach to dynamically adjust the bandwidth upon request to maximize memory bandwidth utilization. Memguard [146] is a software technique proposed to provide bandwidth partitioning without changing the bus arbiter at the hardware level. However, it relies on hardware arbiters to provide the fine-grained arbitration. Therefore, it can be seen as a coarse-grained bandwidth partitioning for a group of memory requests originated from one core. A similar hardware-implemented approach is proposed in [54]. Memguard for partitioned multicore processor has been analyzed in [142]. This work considers sequential tasks and assumes the resource budget for each core is given. The main focus is to compute the response time of real-time tasks on regulated cores.

The work in [104] proposes an analysis method to compute an inflated WCET due to the contention for access to main memory. This approach requires the knowledge of memory demand for each task in a quantified value such as the total number of accesses. However, this analysis is only applicable for time-triggered execution in which cores are synchronized and tasks periodically repeat an off-line schedule. Most of previous discussed works have a similar assumption about the execution pattern of tasks. It is not clear, however, how these approaches can leverage the massive progress in multiprocessor scheduling theory. In what follows, we discuss an emerging technique that explicitly treats memory as a resource similar to processor cores. Thus, memory can be scheduled using the same algorithms proposed for processor cores.

2.2.1 The Co-Scheduling Approach

The work in [108] proposes a co-scheduling approach that can avoid memory contention from a software level. In this approach, each task is split into two phases called memory and computation. During the memory phase, the task's code and data is loaded from main memory to the local memory of processor core, and during the computation phase, the task executes out of the local memory without access to main memory. The co-scheduling approach exactly aligns with the **software managed multicore (SMM)** architectures. SMMs are promising alternatives for real-time systems due to their scalability, power efficiency and predictability [76]. In SMM, each core can only access an SPM while main memory accesses are explicitly done through the use of DMA. Good examples for SMM architecture are Cell processor [92] and FlexPRET [149]. In contrast, traditional multicore platforms with coherent caches are very hard to adopt for real-time systems because they make the analysis very difficult and often result in pessimistic bounds.

While [108] is introduced to avoid contention between single core and I/O devices, the work in [143] uses this model for multicore with partitioned scheduling. A TDMA arbitration is applied to avoid the contention between cores for access to main memory. To utilize memory slots, the priority levels of memory phases are increased over computation phases during the allotted time. In this case, the separation of memory and computation phases has the advantage to control memory phases and allow them to preempt computation phases. A simulation-based evaluation for partitioned scheduling is conducted in [15] using different scheduling policies.

In [99], the authors focus on the theoretical aspect of this model and identify the critical instant (the worst-case activation pattern that leads to WCRT of task under analysis) assuming fully preemptive scheduling of both memory and computation phases on single core. By knowing the critical instant, an exact response time analysis is derived. While we adopt this deterministic model to avoid memory contention, the main motivation in [99] is to exploit the burst-feature of some DMA engines to speedup the memory transfer time. Recently, [144] applies the co-scheduling approach to avoid memory contention in multicore processor. In this work, a global preemptive scheduling is considered in which each task is assumed to have a dedicated partition in each local memory to avoid re-loading the task content after each preemption point. In addition, the memory phases are designed to have priorities higher than the computation phases. This latter point is designed to ensure that memory phases are not interfered from computation phases.

The explicit nature of the co-scheduling approach has been utilized indirectly by other work. For example, tasks migrate between cores in global scheduling and this causes CRMD. A technique has been proposed to warm up the cache before the task execute on the new processor [124]. The transfer of cache lines is initiated explicitly by the scheduler from the source core to the target core. It is contrasted with the traditional pull mechanism where the target cache has to implicitly pull the data from the source cache or the shared cache. Another example is the work in [94] which proposes a framework to manage the cache using cache locking, a technique used to make caches behave like scratchpads [112]. An important step in this framework requires knowing the memory footprint of the task to do prefetching and then locking.

2.2.2 Limitations Imposed by the Co-Scheduling Approach

A real implementation of the co-scheduling approach has been demonstrated on many available **commercial off-the-shelf (COTS)** platforms. In particular, our recent work [119] ports the full automotive suite of EEMBC benchmarks [1] to execute according to the co-scheduling approach on MPC5777M, a quad-core **micro-controller unit (MCU)** from Freescale [2]. This MCU includes four cores: two E200Z710 application cores operating at 300 MHz, one E200Z425 I/O core and one additional core for delayed lockstep operation. Each core features private SPM for instruction and data of sizes 16 KB and 64 KB, respectively, and all cores share a single SRAM of size 404 KB. Since tasks have to be loaded into the local memory before execution, Erika OS [3] has been modified to load the task's code and data sections from SRAM into SPM. Similarly, the work in [53] ports a simplified version of a flight management system to execute on TMS320C6678, a COTS architecture from Texas Instruments [4]. In addition, the work in [30] explores the applicability of the co-scheduling approach on an embedded multicore heterogeneous platform [5]. This platform has a 4-core host processor and a cluster of eight **digital signal processing (DSP)** cores serving as an accelerator.

Even though the co-scheduling approach has been implemented on several fully COTS platforms, we discuss in this section some limitations that may arise in different settings. In particular, the discussion is split into four limitations, and we discuss how each one can be addressed.

(1) Local memory size: since code and data of each task must be explicitly loaded into a local memory before being executed, tasks must fit inside the local memory of one core. We note that there are existing COTS platforms with relatively large local memories that can fit many existing real-time applications. For example, Cell Processor has 256KB SPM for each core. To put things into prospective, FreeRTOS which is a popular OS in the real-time systems domain, has around 8-10KB memory footprint (based on the underlying hardware and the compiler used). Also, the work in [136] has transformed many applications from EEMBC benchmarks to execute according to the co-scheduling approach with a relatively small size of SPM (16KB). In fact, a large number of real-life embedded applications including those for control have a small memory requirement [56, 132]. If, however, the code and data of one task cannot fit in the available local memory space, we

[1]http://www.eembc.org/
[2]Acquired by NXP Semiconductors in December 2015
[3]http://erika.tuxfamily.org/drupal/
[4]http://www.ti.com/product/TMS320C6678
[5]http://www.ti.com/product/66AK2H12

can split the task into multiple segments such that each segment can fit inside the available local memory as discussed in Section 3.7.

(2) Determining the working set: the co-scheduling approach relies on obtaining the set of memory blocks that is needed by each task for load and unload phases. The task memory footprint can be represented as a set of memory blocks, including both code and data, where each block could be a cache line or memory page. In general, the memory blocks can be determined by means of either compiler-driven approaches [134, 98], using program annotations [108] or using measurement-based methods [94]. In particular, the authors in [94] propose a measurement-based method that can be used to determine the memory blocks of load and unload phases at the level of virtual memory pages. They basically profile the task and convert the absolute addresses of memory accesses to relative values with respect to the execution-independent memory regions. Hence, load and unload code can be written based on these relative addresses.

We acknowledge that the working set for some tasks can be very hard to perfectly determine before run-time. In this case, the co-scheduling approach can be partially applied, say for 80-90% of memory accesses and the remaining 10-20% can be analyzed assuming contention for access to main memory. In particular, the recent work in [95] introduces an automated profiling tool that can modify arbitrarily C programs to behave according to the co-scheduling approach by inserting prefetch statements before the function under analysis. The results show that both overhead and missed accesses are small for a number of non-trivial real-time benchmarks. For example, 95% of cache misses are reduced for a JPEG image encoding benchmark. In addition, the work in [35] proposes MadT, a tool that detects data memory accesses of general purpose applications. It translates the virtual addresses to their symbolic variable names, a relevant feature that can be used to implement load and unload phases. In addition, the results in [108] based on the automotive program group of MiBench and DES benchmarks indicate that the amount of loaded data is only slightly higher than the amount actually used.

(3) Handling I/O data: since tasks can load data from main memory before they execute, this data can also be memory mapped I/O. However, I/O that reads or writes to main memory directly using DMA has to be controlled to avoid the interference with application cores. The work in [14] shows how this can be done through hardware modifications. Alternatively, some COTS platforms, like MPC5777M, have a dedicated bus to handle I/O transactions. In this case, traffic traversing the dedicated I/O bus has to be processed by an I/O core and buffered before reaching the application cores, similar to our implementation in [119].

(4) Cache related issues: local memory can be either cache or SPM. For caches, it is particularly important to facilitate load and unload phases. In particular, the platform API should provide cache prefetching primitives, e.g., P4080 platform [58] has a stashing mechanism that allows a DMA component to put data directly into the cache; otherwise, `LOAD` instructions can be used to move data from main memory. In addition, the cache should employ a write-back policy and have primitives to invalidate some cache lines and write them back to main memory. For example, Intel platforms provide `CLFLUSH` instruction [71].

Furthermore, during load phase, self-eviction should be avoided, in the sense that loading a cache line should not evict another cache line loaded within the same phase. To illustrate how memory blocks can be allocated without conflicts, we can think of cache memory as a two dimensional array in which the columns are the cache ways and the rows are the cache sets. In systems with virtual memory, the memory blocks can be allocated without conflict along cache ways by re-arranging the physical addresses (page coloring). However, in caches with non-deterministic replacement policy, only one cache way can be utilized. Fortunately, some cache controllers have a *lockdown* feature that can be implemented at the granularity of either a single line or way. This feature allows memory blocks to be hooked on a particular cache way without being overwritten until they are explicitly unlocked. In fact, the *Colored Lockdown* technique proposed in [94] can be used to avoid conflicts in loading task's code or data. The key idea of this technology is to combine coloring and cache locking to deterministically control cache allocation of memory blocks. With this combined method, the whole cache can be utilized to allocate memory blocks without conflicts. Similarly, the work in [38] proposes *column caching*, a hardware modification to the replacement unit of the cache in order to limit replacement to the column (way) specified by a bit-vector stored in page table entries.

The multicore processor can also have a **last-level cache (LLC)** that is shared between cores as common in today's architectures [73]. Spatial isolation techniques as discussed in Chapter 2 can be used to partition the shared cache among different cores. Here, we assume there is no timing interference between cores for access to shared cache. This can be achieved using *bankization* [106], a technique to partition the cache into banks and each core is assigned a private set of banks.

2.2.3 Memory Space Allocation

The memory used by real-time applications is usually statically allocated to avoid non-deterministic memory management systems. This used to be a sufficient solution for some real-time applications; however, modern real-time systems are increasingly becoming complex and demanding for more flexible memory allocation. In fact, building a time-predictable memory manager is a challenging task. The problem is that the memory state changes as data come and leave. Thus, memory allocation time is not always fixed. Half-fit [105] was the first dynamic memory manager that allocates memory space in a constant time, but the cost can be wasting half of the memory space. Another work [41] proposes a scheme called **compact-fit (CF)**. It provides a bounded response time (either constant or linear to the size of allocated data) for allocating data regardless of the current state of the memory. Since our main focus is the arbitration time for access to main memory (the temporal aspect), the timing interference due to memory space management is out of this dissertation's scope.

2.3 Multiprocessor Real-time Scheduling

Uniprocessor real-time scheduling has received a significant amount of consideration in the past [31]. The computational model has been extended from a simple theoretical model to include more realistic models found in today's systems. On the other hand, multiprocessor real-time scheduling has not received the same amount of consideration as in uniprocessor. The theory of multiprocessor real-time scheduling studies the scheduling of real-time workloads on multiprocessor systems. This topic has seen big advances over the past years and is still growing in a fast pace. We will discuss in this section the most closely related aspects, and refer the reader to a recent book on this topic by Baruah *et al.* [20]. In multiprocessor real-time scheduling, one has to distinguish two scheduling paradigms: (1) sequential tasks and (2) parallel tasks. The real-time research community until recently has focused on the first paradigm in which the real-time system is modeled as being composed of a finite number of sequential tasks. The work in [46] surveyed the literature of different techniques for scheduling sequential tasks on multiprocessor systems. Although scheduling parallel tasks without deadlines has been addressed by parallel-computing researchers for a long time [1], there has been a recent attention to schedule parallel tasks in real-time systems. This has been observed in [122],"*the growing importance of parallel task models for real-time applications poses new challenges to real-time scheduling theory that has mostly focused on sequential task models.*" Modeling modern real-time systems as being composed of parallel applications is crucial because the timing constraints of computation-heavy real-time applications such as synthetic vision and object tracking [79] cannot be feasibly executed on single core processor. We will provide a brief survey of these works in Section 2.3.5.

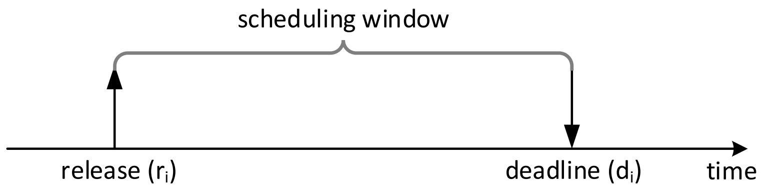


Figure 2.4: A job needs to execute for amount of time equal to its WCET between its release time and deadline.

2.3.1 Task Model

The real-time workload comprises units of work known as *jobs*. These jobs are generated by a finite collection of independent recurrent tasks $\Gamma = \{T_1, \cdots, T_n\}$. Each job is characterized by three values: release time r_i, absolute deadline d_i and WCET as depicted in Figure 2.4. Over the years, different models have been proposed for hard real-time tasks. One widely-used model is the 3-parameter model. As the name indicates, each task is characterized by WCET e_i, period p_i and relative deadline D_i. This model extends the classical **Liu and Layland (LL)** model [87] which has only two parameters: WCET and period. Adding relative deadline as a third parameter allows to model tasks that occur infrequently (large period) but with urgent deadline (small deadline). Based on the relationship between the relative deadline and the period, a 3-parameter task system may be classified as follows.

- An implicit-deadline where the relative deadline is equal to the period. This is exactly the same as LL model.
- A constrained-deadline where the relative deadline is no larger than the period.
- An arbitrary-deadline where there is no restriction on the values of the relative deadline and the period.

In addition, a *sporadic* task model specifies a lower bound on the arrival of successive jobs of one task. In contrast, a *periodic* task model specifies an exact separation of successive jobs.

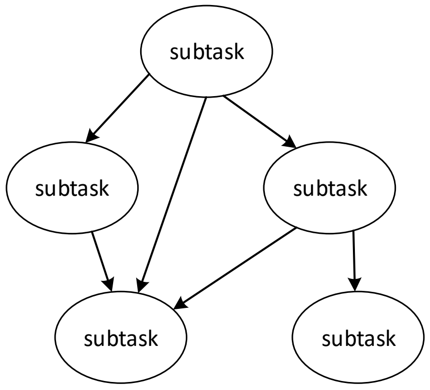


Figure 2.5: DAG task.

Generally, the 3-parameter task model does not allow parallelism within one task. However, the trend in current technology demands models that are capable of exposing intra-task parallelism. The **directed acyclic graph (DAG)** task model [21] has been proposed to capture such parallelism where each task is characterized by a DAG $G_i = (V_i, E_i)$ as shown in Figure 2.5. Each node $v \in V_i$ of the DAG represents a sequential subtask characterized by WCET. In addition, each directed edge $(v, w) \in E_i$ of the DAG represents a precedence constraint. That is, subtask v must finish before w can start. Therefore, the sporadic DAG task model can be characterized by a DAG G_i, a relative deadline D_i and a period p_i. Unlike the sequential task model where each task is described by a single WCET e_i, the DAG task model describes a task as a collection of subtasks with order relation in their execution. Similar to sequential tasks, DAG tasks generate a sequence (possibly infinite) of jobs. Upon job release, all sub-jobs (dag nodes) become available for execution subject to the precedence constraints.

2.3.2 Hard Real-time

Hard real-time systems do not tolerate any deadline misses. In contrast, soft real-time systems allow for some deadline misses to occur or deadlines can be missed by no more than a certain amount of time (bounded tardiness). We will focus on hard real-time scheduling; thus, the large body of research on soft real-time scheduling falls outside the scope of this dissertation.

2.3.3 Preemption and Migration

Scheduling can be classified based on *preemption*. In *preemptive* scheduling, the scheduler is allowed to stop one job (perhaps to execute another job) and resume its execution at later time. In contrast, after a job starts its execution, *non-preemptive* scheduling never stops this job until completion. *Limited-preemptive* scheduling is proposed as a hybrid approach between preemptive and non-preemptive. A recent survey [32] discusses different approaches for limited-preemptive scheduling. An approach directly related to this dissertation is called *fixed preemption points*. That is, a job is divided into non-preemtive segments and preemption can only occur at the boundaries of these segments. There are two models of preemption in this approach: lazy and eager [97, 133]. In the lazy model, a job is preempted if it is the lowest priority job running in the system. The eager model, on the other hand, preempts the first lower-priority task to reach a preemption point. For the reason we mention at the beginning of Chapter 3, we will primarily focus on non-preemptive scheduling and adopt fixed preemption points approach in Section 3.7.

Scheduling can be further classified based on *migration*. In *global* scheduling, there is no restriction on which processor a job can use to execute. In contrast, *partitioned* scheduling restricts each task to execute on one particular processor. Again, *clustered* scheduling is proposed as a hybrid approach between global and partitioned. In this approach, the processors are partitioned into clusters, and task migration is only allowed within one cluster, i.e., global within one cluster. We will primarily focus on global scheduling for sequential tasks in Chapters 3 and 4.

2.3.4 Static and Dynamic

The simplest scheduling scheme is called *cyclic-executive* [33]. A real-time clock is used to activate tasks at the right time. The advantage of this static approach is that the run-time overheads such as context-switching can be determined at compile time. Other approaches are dynamic and based on priority. At run-time, the scheduling algorithm allocates the available processors to the highest priority jobs. Based on how priority is assigned, scheduling can be classified into three types:

- **Fixed task priority (FTP)**, each task is assigned a unique priority and its jobs inherit this priority. The most popular algorithm of this type is **rate monotonic (RM)** in which priority is assigned based on period, tasks with smaller periods are given higher priority. Our focus will be on this type.

- **Fixed job priority (FJP)**, jobs of the same task may have different priorities. However, the priority will never change once assigned. The **earliest deadline first (EDF)** is an example of this type in which priority is assigned based on deadlines, jobs with earlier deadline are given higher priority.

- **Dynamic priority (DP)**, there is no restriction on priority assignment which can change at task or job level. The **proportional fair (pfair)** is an example of this type.

2.3.5 Schedulability Test

In building hard real-time systems, it must be guaranteed before run-time that all deadlines will be met by the scheduling algorithm. This guarantee is made by a *schedulability test*. The output of this test determines whether the task set is schedulable or not. Real-time tasks are often characterized by worst-case values rather than exact; thus, it is expected (at run-time) that tasks behave better than the worst-case bounds. The notion of *sustainability* is a desirable property (fundamental from an engineering point of view) of the schedulability test to ensure that the task set is still schedulable when tasks execute for less than their WCET.

The *utilization bound*, defined below, can be used as a schedulability test. The task utilization is denoted as $u_i = e_i/p_i$ and $Ut = \sum_{\forall T_i} u_i$.

Definition 1. The utilization bound for a given scheduling algorithm A is the largest utilization value U_A such that all task sets with total utilization $Ut \leq U_A$ are always schedulable by A.

The utilization bound can also be used as a metric to compare different scheduling algorithms. That is, if $U_A > U_B$, then algorithm A is better than algorithm B. However, the utilization bound of FTP global scheduling is very poor. It can reach $1 + \epsilon$, where ϵ is an arbitrarily small number, regardless of the number of processors due to a phenomenon commonly called *Dhall effect* [51]. Thus, another metric called *speedup factor* is often used.

Definition 2. If an optimal scheduler can schedule a task set on m unit-speed processors, then algorithm A with *speedup factor* b can schedule this task set on m b-speed processors.

Since the optimal scheduler for FTP global scheduling is not known [57], the speedup factor can be used as a metric to compare algorithms rather than a schedulability test.

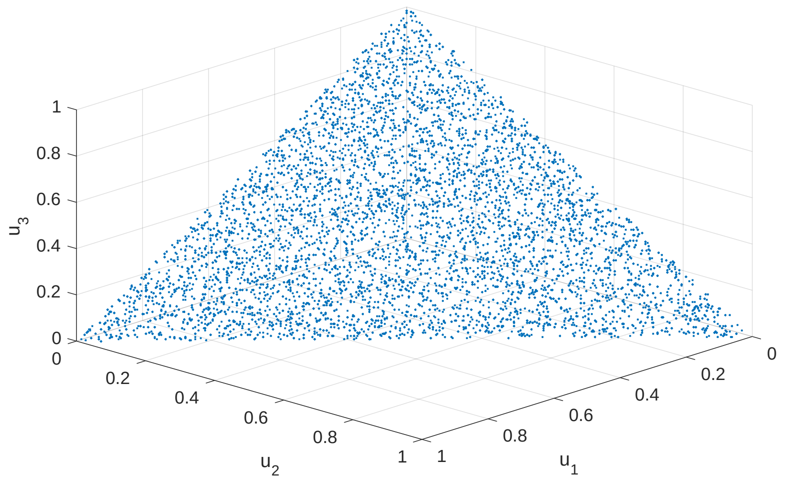


Figure 2.6: 5000 task sets of UUnifast(3,1).

The per-task schedulability test, on the other hand, can be used to test whether or not a task set is schedulable. Two types of tests are available [45]: **deadline analysis (DA)** and **response-time analysis (RTA)**. With DA, the latest time a task can execute is when it completes within its deadline while the latest time a task can execute with RTA is when it completes within its WCRT. As noted in [45], RTA dominates DA and the latter can be easily extended to RTA using an iterative method. In this dissertation, we use DA to compare our proposed algorithms against others rather than using speedup factors or utilization bounds.

To compare between different algorithms with DA test, the *acceptance ratio* metric is often used. It is the ratio between the number of schedulable task sets over the total number of task sets. An algorithm such as UUnifast(n,Ut) [28] can be used to randomly generate task sets such that each task set contains n tasks and total utilization Ut. In Figure 2.6, we plot the results of 5000 task sets for $n = 3$ and $Ut = 1$. As we can see, utilization values (triplets) are uniformly distributed.

Sequential Tasks

Since we consider in this dissertation global non-preemptive scheduling for sequential tasks, we will restrict our discussion to this type. We refer the reader to [46] for a survey of preemptive multiprocessor scheduling including both partitioned and global approaches. Non-preemptive global scheduling was first addressed in 2006 by Baruah [22]. He proposed a sufficient schedulability test for global non-preemptive scheduling of periodic tasks which can be easily extended to include sporadic tasks. However, this test may reject a task set with low utilization if it contains a task whose execution time is longer than the relative deadline of any other task because of non-preemptive execution. Even though this is true for uniprocessor scheduling, the case in multiprocessor is different because a high-priority task can run on different processor if other processors are blocked by low-priority tasks with large execution times. Guan *et al.* [65] provided a schedulability analysis for global non-preemptive scheduling under earliest deadline first (EDF). Their analysis is based on a novel approach by Baker [16] and techniques introduced in [26]. They also used ideas from [17] to limit the amount of carry-in workload. The work in [64] improved upon the previous work [65] by providing an analysis which is of polynomial computational complexity instead of pseudo-polynomial. They also focused on FTP scheduling as opposed to EDF.

A schedulability test for EDF under non-preemptive global scheduling is also proposed in [82]. In this work, the task model is generalized to include both hard and soft real-time tasks with pre-defined tardiness bounds and arbitrary relative deadlines. Moreover, the work in [47] introduced a global fixed priority scheduling with deferred preemption. It is again a more general model that includes both preemptive and non-preemptive scheduling. In this model, each task is characterized by a parameter called **final non-preemptive region (FNR)**. By increasing the FNR length to include all of the task's execution time, we can have a non-preemptive scheduling.

The state-of-the-art schedulability analysis of FTP non-preemptive global scheduling is found in [64]. This work combines several techniques developed over the years to derive an efficient analysis. A summary of this analysis is provided in Section 3.3 to serve as a foundation to understand Chapters 3 and 4.

Parallel Tasks

With respect to parallel tasks, Goossens and Berten [59] discuss an interesting classification of parallel tasks: rigid, moldable and malleable. A rigid task can only execute when a fixed number of processors are available; otherwise, the task will not execute. A moldable task is similar to rigid task, but the number of processors is specified at job level, i.e., different jobs of the same task can be assigned different number of processors. In contrast, the malleable task has no restriction and can execute on any number of cores. The traditional Gang scheduling aligns with the definition of rigid tasks. In [74], EDF is applied for gang scheduling and [59] applies FTP scheduling. Most of the work in parallel scheduling assumes malleable tasks.

Lakshmanan *et al.* [80] introduce *fork-join* model for parallel tasks in which the task alternates between sequential and parallel segments. The authors propose a transformation method that impose an artificial deadline for each subtask. Once these deadlines are determined, the schedulability analysis becomes equivalent to scheduling independent sequential tasks. [80] has been generalized in [121, 102] to lift some restrictions on the task model such as the number of subtasks within parallel segments. Instead of transformation method, the work in [39] proposes global EDF scheduling and uses the DA test. The concept of *critical interference* is introduced to capture the interference of parallel subtasks. This concept is adopted in [93] for FTP global scheduling, and RTA is used to tighten the schedulability conditions. Similarly, RTA is used in [11] but for FTP partitioned scheduling.

The work in [21] proposes DAG task model and provides a schedulability analysis for single parallel task. Later, the works in [83, 24] consider the same model but for multiple parallel tasks. A response-time analysis for DAG tasks is proposed in [100]. This work follows similar techniques as in [25] for sequential tasks. However, a novel idea is proposed to capture the workload of parallel tasks. We recall that parallel tasks can have different degree of parallelism and can utilize more than one core at a given time. Thus, this imposes a challenge to bound their workload compared to sequential tasks.

In [83], the authors propose *federated scheduling* to schedule parallel tasks. Unlike global scheduling where tasks (including their subtasks) share one global queue, tasks in federated scheduling are assigned dedicated cores. Since we use this approach in Chapter 5, a detailed discussion of federated scheduling is provided in Section 5.2.

Chapter 3

Global Scheduling of 3-phase Sequential Tasks

The co-scheduling approach [108] was initially proposed with two phases, load and computation. In this chapter, we propose to extend this approach with a third phase to unload modified data back to main memory. We then discuss how to schedule 3-phase tasks on a multicore processor. We focus on non-preemptive FTP global scheduling in which the ready tasks are inserted in one global queue and each task is assigned a unique priority. We refer to this global scheduler as **global predictable execution model (gPREM)**.

As depicted in Figure 3.1, tasks are divided into three phases: (1) *load* phase, (2) *computation* phase and (3) *unload* phase. During the load phase, task's code and data are loaded from main memory to the local memory of the assigned core. During the computation phase, the task executes out of the core's local memory without main memory stalls. The modified data is then written back to main memory during the unload phase. Based on this 3-phase execution model, the scheduler can enforce a *contention-less* memory access by scheduling only one memory phase (load or unload) at any time. Other tasks are allowed to execute but they must be in their computation phases. This execution model

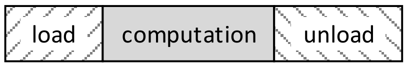


Figure 3.1: 3-phase execution model.

unload
load
computation
memory
computation

(a) Combined unload and load phases. (b) Two phases.

Figure 3.2: Simplified 3-phase task.

provides a highly predictable execution time for both memory and computation phases. Specifically, the memory phases suffer no delay from other tasks and computation phases execute without memory stalls.

We described the 3-phase model for a single task. However, real-time systems are often multitasking in which more than one task execute concurrently. Since 3-phase tasks have two memory phases (load and unload), having to dynamically schedule both of them complicates the schedulability analysis. Instead, we propose to combine the unload phase of one job with the load phase of next job executed on the same core as shown in Figure 3.2(a). We refer to these combined phases as the *memory phase*. Suppose J_b is the next job to execute after J_a on the same core. Then, the unload phase of J_a is executed non-preemptively with load and computation phases of J_b. In other words, 3-phase tasks now have two phases: memory and computation as shown in Figure 3.2(b). We will be using the diamond pattern to represent memory phases from now on.

To schedule 3-phase tasks efficiently, we propose to execute them non-preemptively. Intuitively, if we allow preemption, we need to reload the code and data at each preemption point. Otherwise, we need to assign each task a private partition in each local memory to avoid conflicts between tasks; however, this choice may not be practical for systems with large number of tasks. In industry practice, non-preemptive scheduling is often preferable due to its lower run time overhead. In addition, the non-preemptive blocking time due to low-priority tasks is less severe in multicore than single core because high-priority tasks can still execute in parallel on other cores [64]. In fact, neither preemptive nor non-preemptive global fixed-priority scheduling dominates the other, i.e., there are some task sets that are schedulable under preemptive that are not schedulable under non-preemptive and vice-versa.

3.1 Contention-based Execution

The co-scheduling approach inherently uses local memory to execute tasks. In other systems, local memories are used to reduce the number of requests to main memory by exploiting the locality of reference. However, estimating the WCET of real-time tasks in such platforms is challenging. With local memories, memory requests have two different latencies for hit and miss; hence, the execution time of one task will depend on the distribution of these memory requests. In addition, preemptive scheduling can increase task's execution time by 33% due to CRPD [49]. To solely focus on memory contention problem, we will use the contention-based non-preemptive global scheduling (gCONT) as our base system for comparison purposes. With respect to main memory, each task is characterized by a total number of memory requests. In gPREM, memory requests occur in one phase while memory requests in gCONT are spread across the execution of the task and can occur at any time. That is, the total number of memory requests of one task under gCONT and gPREM are the same, assuming the local memory is managed under gCONT to avoid conflict misses.

A bound on memory latency for gCONT can be easily obtained for a fair hardware arbiter such RR. In this case, the memory portion of each task is inflated by m, the number of cores in the system, by assuming each memory request is preceded in the worst-case by $m-1$ requests from other cores. In other words, each task can utilize $1/m$ of memory bandwidth. We note that memory requests in current architectures are not serviced in RR nor they have the same access latency. Instead, current memory controllers are designed for average-case performance where utilizing the memory bandwidth is the main objective. To achieve such goal, current memory controllers exhibit a dynamic behavior where the latency of single memory request varies based on the history of previous requests. As a result, a complicated analysis for each memory controller is needed to derive an upper bound on memory access latency [75, 140]. In this case, the memory inflation can exceed m; however, we favor gCONT by assuming RR (a fair arbiter) and main memory is treated as a black box with constant access latency. As we mentioned earlier, these complexities are avoided in gPREM because it does not rely on hardware arbiters and tasks during memory phases access memory without interleaving effect of other tasks.

3.2 System Model

We consider a set of n sporadic 3-phase tasks, $\Gamma = \{T_1, \dots, T_n\}$, to be globally scheduled on a processor of m homogeneous cores $\{core_1, \dots, core_m\}$. Each core has a private local

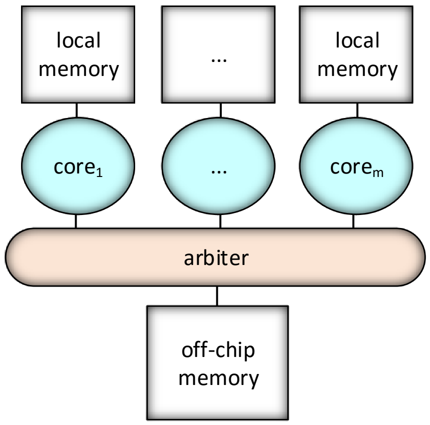


Figure 3.3: Abstracted hardware architecture.

memory, and all cores share a single off-chip main memory as shown in Figure 3.3. We use J_i to denote any job of T_i with r_i as its release time and d_i as its absolute deadline. The *sporadic* task model has three parameters for each task T_i. That is, a relative deadline D_i, worst-case execution time e_i and period or inter-arrival time p_i. We consider a constrained deadline model in which $D_i \leq p_i$. The release times of jobs are assumed to be unknown at compile time and their actual execution time can be less than the worst-case value. Since we explicitly consider memory as a system resource, each task is characterized by three WCET values corresponding to each phase of 3-phase model: (1) e_i^x, computation time, (2) e_i^v, memory load time and (3) e_i^w, memory unload time. We use e_i^m to denote the memory time including both load and unload phases. We note that $e_i^m = e_j^v + e_i^w$ since the memory phase of J_i includes the unload phase of previous job J_j and the load phase of J_i. Since the previous job in dynamic scheduling is generally unknown as core allocation and execution ordering are determined at run time, e_i^m cannot be determined at compile-time. We will address how to build a safe bound on e_i^m in Section 3.6.3. We let $e_i = e_i^m + e_i^x$ as an upper bound on the total execution time including both memory and computation phases. This assumption holds for timing compositional architectures or systems where the memory time is additive to computation time, i.e., there is no anomalies assuming $e_i \leq e_i^m + e_i^x$.

Each task has a unique priority indicated by its index such that T_a has higher priority than T_b if $a < b$. We use $hp(k)$ to denote the set of tasks with priorities *higher* than T_k, $lp(k)$ to denote the set of tasks with priorities *lower* than T_k, and $lep(k) = \{T_k \cup lp(k)\}$. The

Table 3.1: Summary of notations.

T_i	one task
Γ	task system
m	number of system's cores
n	number of system's tasks
e_i	execution time of T_i including memory and computation
e_i^v	load time
e_i^w	unload time
e_i^m	memory time including load and unload
e_i^x	computation time
D_i	task's deadline
p_i	task's period
u_i^m	memory utilization of T_i
u_i^x	computation utilization of T_i
Um	total memory utilization of Γ
Ux	total computation utilization of Γ
Ut	total utilization of Γ including memory and computation

memory and computation utilization of each task is denoted as $u_i^m = (e_i^v + e_i^w)/p_i$ and $u_i^x = e_i^x/p_i$, respectively. Similarly, the total memory utilization of all tasks is denoted by Um, the total computation utilization is denoted by Ux and the total utilization (including both memory and computation phases) is denoted by Ut. All these notations are summarized in Table 3.1 and will be used in subsequent chapters.

In addition, all time values are assumed to be non-negative integers and expressed as cycles of the most precise clock in the system. The length of an interval $[a, b]$ is $b - a + 1$. Lastly, we use the notation $(x)_0$ meaning $max(0, x)$ to simplify expressions.

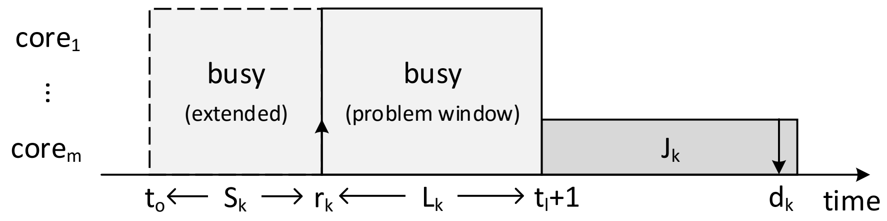


Figure 3.4: The necessary condition for J_k to miss its deadline.

3.3 Deadline Analysis

The schedulability analysis in this chapter is based on deadline analysis. Hence, we dedicate this section to go through this analysis in detail. To simplify the discussion, we consider traditional tasks with computation phases only. The analysis in Section 3.6 will consider 3-phase tasks. The deadline analysis is based on a basic argument that for any work-conserving scheduler, a job J_k will meet its deadline unless it is pushed by the *interference* of other tasks.

Definition 3. The interference refers to a time interval in which J_k is not able to execute because all processor cores are busy executing other tasks.

The job J_k is called a *problem job*, and the time interval $[r_k, t_l]$ is called a *problem window*, where t_l is the latest time for J_k to execute and finish by its deadline, i.e., $t_l = d_k - e_k$. We use L_k to denote the problem window length, i.e., $L_k = t_l - r_k + 1$. Since we consider a non-preemptive execution, a job will meet its deadline if it acquires a processor core at or before t_l. Based on this argument, a sufficient schedulability condition can be easily established. That is, if the interference of other tasks is less than the problem window length, the problem job is guaranteed to meet its deadline. In Figure 3.4, we show a task system generating an interference that prevent J_k from running at or before t_l. We then have to verify that the same condition holds for each task.

In multiprocessor scheduling, the maximum interference of a task system over an interval of time is not always obtained when all tasks are released simultaneously, i.e., the critical instant of uniprocessor. Moreover, finding the release times which result in the maximum interference is an NP-hard problem [91, 22]. However, due to the NP-hardness nature of the problem, we note that the contribution of any task in any interval of time is

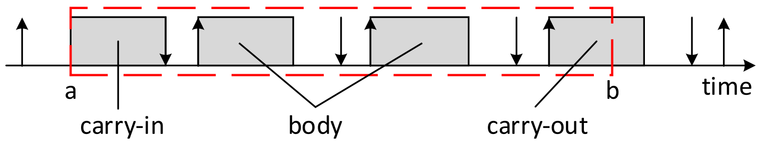


Figure 3.5: The workload of a task inside a window of time $[a, b]$.

never greater than the worst-case workload of the task in the same interval [26]. Thus, we can use the worst-case workload of all tasks T_i in the problem window of task T_k to derive an upper-bound on the interference.

The workload of T_i in a window of time $[a, b]$ is composed of three parts as shown in Figure 3.5.

1. *Carry-in*: the contribution of at most one job with release time before a and deadline after a.
2. *Body*: the contribution of jobs with both release time and deadline inside the window.
3. *Carry-out*: the contribution of at most one job with release time inside the window and deadline after b.

According to these definitions, a job with release time before a and deadline after b is considered as a carry-in job. We also assume the worst-case activation such that carry-in jobs start as late as possible and carry-out jobs start as early as possible.

In addition, assuming all tasks can have carry-in jobs has a large impact on the amount of workload and consequently, the interference. Thus, we consider a busy downward extension of the problem window in which there are at most m tasks that can have carry-in jobs. The extended problem window has an earlier starting point t_o and shares the same endpoint t_l as shown in Figure 3.4. We now define t_o as follows.

Definition 4. We let t_o to denote the earliest time instant before r_k such that from t_o to r_k either all cores are busy with tasks from $hp(k)$ or there is at least one pending task from $hp(k)$. If such time does not exist, then we let $t_o = r_k$.

Based on this definition, we have the following lemma.

Lemma 1. *The time interval $[t_o, r_k)$ is busy.*

Proof. Assume a time interval inside $[t_o, r_k)$ which is not busy, i.e, there is at least one core idle. Definition 4 states that inside $[t_o, r_k)$ either all cores are busy with tasks from $hp(k)$ or there is at least one pending task from $hp(k)$. Since FTP global scheduling is work-conserving, both cases contradict the above assumption. □

The following lemma bounds the number of tasks that can have carry-in jobs inside the problem window that starts at t_o.

Lemma 2. *There are at most m tasks that can have carry-in jobs of which at most $m-1$ tasks from $hp(k)$.*

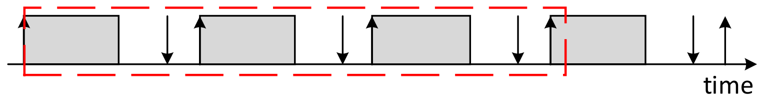
Proof. We use $t_o - 1$ to denote the time instant before t_o. The complement of Definition 4 states that at $t_o - 1$, not all cores are busy with tasks from $hp(k)$ and there is no task pending from $hp(k)$. This limit the number of tasks in $hp(k)$ executing at $t_o - 1$ to $m - 1$. Furthermore, since there is no $hp(k)$ task pending at $t_o - 1$, it follows that only tasks in $hp(k)$ that are executing at $t_o - 1$ can have carry-in jobs. Similarly, no task in $lep(k)$ can start executing at or after t_o as per Definition 4. Since it is not possible to execute more than m tasks at the same time, the lemma follows. □

We note that even though the extended problem window that starts at t_o has the advantage of limiting the amount of carry-in jobs, finding the start point t_o is of pseudo-polynomial time complexity [17], given that the total utilization (Ut) is strictly less than m. The authors of [64] observed that choosing a window of length L_k and starting at t_o is sufficient for the schedulability test. Intuitively, computing bounds on the amount of work inside a small time interval (L_k) is tighter than a large time interval $(S_k + L_k)$ as the amount of workload gets amortized over larger intervals [17].

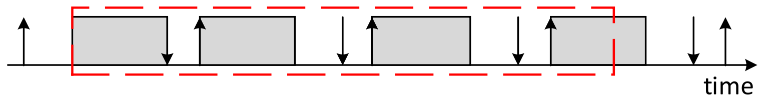
We use $W_k^{CI}(T_i)$ and $W_k^{NC}(T_i)$ to denote the workload of T_i in the problem window of T_k with and without carry-in, respectively. Obviously, $lep(k)$ tasks can only have one job as carry-in workload; thus, $W_k^{CI}(T_i) = \min(e_i, L_k)$ and $W_k^{NC}(T_i) = 0$. In contrast, $hp(k)$ tasks can be activated and executed inside the problem window. Thus, their workload can be computed from Figure 3.6 as follows.

$$W_k^{NC}(T_i) = \left\lfloor \frac{L_k}{p_i} \right\rfloor e_i + \min(e_i, L_k \bmod p_i) \tag{3.1}$$

$$W_k^{CI}(T_i) = \min\left(\left\lfloor \frac{(L_k - e_i)_0}{p_i} \right\rfloor e_i + e_i + \beta, L_k \right) \tag{3.2}$$



(a) $W_k^{NC}(T_i)$



(b) $W_k^{CI}(T_i)$

Figure 3.6: The workload without and with carry-in.

where

$$\beta = \min(e_i, (L_k - e_i)_0 \bmod p_i - (p_i - D_i)). \tag{3.3}$$

We note that $W_k^{CI}(T_i) \geq W_k^{NC}(T_i)$ by at most one complete job. Thus, we denote their difference by

$$W_k^{diff}(T_i) = W_k^{CI}(T_i) - W_k^{NC}(T_i). \tag{3.4}$$

Since m tasks can have carry-in jobs as in Lemma 2, we take the largest m among all tasks.

$$W_k^{diff}(\Gamma) = \sum_{largest(m)} W_k^{diff}(T_i) \tag{3.5}$$

The total workload of all tasks in the problem window of T_k can be bounded as follows.

$$W_k(\Gamma) = W_k^{diff}(\Gamma) + \sum_{T_i \in \Gamma} W_k^{NC}(T_i) \tag{3.6}$$

While $W_k(\Gamma)$ bounds the volume over two dimensions, the horizontal length (interference) can be trivially bounded as:

$$I_k(\Gamma) = \frac{W_k(\Gamma)}{m}. \tag{3.7}$$

Finally, the schedulability test for each task can be established by checking $I_k(\Gamma) < L_k$. This condition has to be checked for each task to declare whether the task set is schedulable or not.

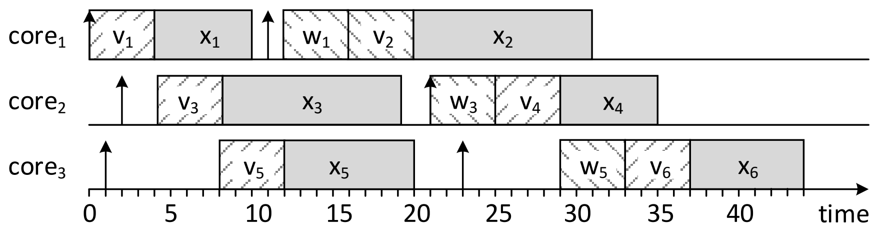


Figure 3.7: An example of gPREM schedule for six jobs on three cores.

3.4 Scheduling Example of gPREM

In this section, we use an example to illustrate gPREM scheduling. Figure 3.7 shows a schedule example of six jobs scheduled on three cores. The up arrows indicate the release time of jobs and the deadlines are omitted for simplicity. These six jobs are from different tasks and the job priority is indicated by its index such that J_a has higher priority than J_b if $a < b$. We use the following notation for task's three phases: V_i for load phase, X_i for computation phase and W_i for unload phase.

J_1 is released at time 0. V_1 is scheduled until time 4. X_1 is immediately executed after its memory phase as we assume a non-preemptive execution. While V_1 is executing, both J_3 and J_5 have been released at time 1 and 2, respectively. Since our scheduler allows only one job to be in memory phase, J_3 and J_5 have to wait until time 4, the end time of V_1. J_3 is chosen to execute before J_5 because it has higher priority. J_2 is released at time 11 but is blocked by V_5 for one time unit. W_1 is merged with V_2 and executed non-preemptively with X_2. From time 0 to time 20, the memory is fully utilized with no idle time. From time 20 to time 21, the memory is idle because there is no pending task inside the ready queue. At time 21, J_4 is released and immediately scheduled to execute its memory phase on $core_2$. Again, V_4 is merged with W_3, the unload phase of the previous scheduled job on the same core. J_6 is released at time 23, but is blocked by J_4 until time 29.

We now highlight two important points from previous example. First, J_5 was pending after its release at time 1 and did not acquire a processor core. Thus, J_5 did not block J_3 as in traditional non-preemptive scheduling. The scheduler is invoked at time 4, the end of J_1 memory phase. At this time, both J_5 and J_3 were pending inside the ready queue. Thus, J_3 was chosen to execute before J_5. This point will help us in Section 3.6 to provide a tighter interference bound. Second, assume J_3 and J_4 are from the same task, i.e., they share the same data. At time 21, the release time of J_4, both $core_2$ and $core_3$ are

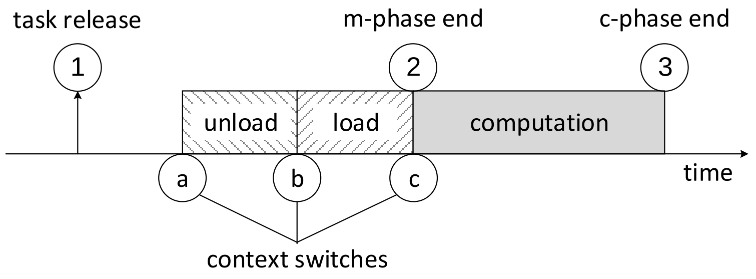


Figure 3.8: Scheduling decisions and context switches.

available to execute J_4. If we choose $core_3$ instead of $core_2$ to execute J_4, there will be data inconsistency for J_4 because the modified data of J_3 is still in the local memory of $core_2$ where J_3 is executed, and cannot be seen by J_4. Since we combine the unload phases with load phases, another job should be scheduled on the same core to carry out the unload phase. To mitigate such problem, we propose to design the scheduler such that if there are more than one core available to execute a job, the scheduler should choose the recent used core by the previous job of the same task. Hence, we guarantee that the modified data is seen by the next job of the same task.

3.5 Scheduler Design of gPREM

In this section, we show how to design gPREM scheduler to be compatible with the data structures of FreeRTOS [1]. The scheduler maintains a global queue in which ready tasks are ordered according to fixed priorities. A task is *inserted* into the ready queue after its release. The scheduler is implemented as an **interrupt service routine (ISR)** triggered by three events 1, 2 and 3 as shown in Figure 3.8. Upon activation, the scheduler checks two conditions: (1) at least one core is idle and (2) the memory is idle because each task starts with a memory phase and only one memory phase is allowed to execute at any time. If these two conditions are not satisfied, the scheduler exists and will be triggered again by a later event. If, however, both conditions hold, the scheduler *extracts* from the top of the ready queue the highest priority task, then invokes the dispatcher (by sending an inter-core interrupt) to do a context switch on the selected core.

FreeRTOS uses **task control block (TCB)** data structure to represent each task.

[1]http://www.freertos.org/

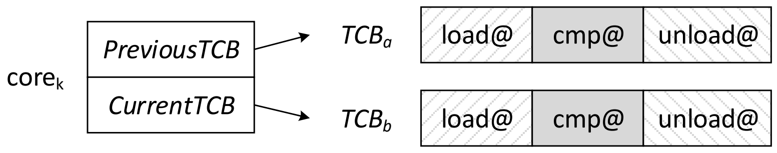


Figure 3.9: Task's TCB and core's pointers.

The TCB contains, among other things, the memory address corresponding to the first instruction of each task. Since each task in our system is composed of three phases, the TCB has to contain three addresses corresponding to each phase as shown in Figure 3.9. These different addresses are used by the dispatcher to properly context switch tasks. Moreover, since we combine the unload phase of previous job to the unload phase of next job to be executed on the same core, each core has to maintain two pointers as shown in Figure 3.9. These two changes are minimal to FreeRTOS data structures. In what follows, we explain how these pointers and data structures are manipulated by the dispatcher.

We show in Figure 3.8 three points a, b and c at which the dispatcher is invoked to do a context switch. At point a, the dispatcher is invoked by the scheduler as explained above to context switch a new job. The dispatcher at this point will first adjust the core's pointers $PreviousTCB$ to point to the TCB of previous job and $CurrentTCB$ to the TCB of current job (the highest priority). After the unload phase of previous job (point b), the dispatcher will be invoked again to do another context switch, $\textsc{Context-Switch}(CurrentTCB.load)$. At point c, the dispatcher will do a context switch for the computation phase of current job, $\textsc{Context-Switch}(CurrentTCB.cmp)$. This cycle continues as long as there are pending tasks.

3.6 Schedulability Analysis

In traditional scheduling, a task becomes pending after its release if all cores are busy executing other tasks. Thus, a time window is defined busy if all processor cores are busy. In gPREM, however, a task can be pending even though there is an idle core because tasks start with memory phase, and no more than one memory phase can be active at any time. Thus, we propose to redefine the term busy for a window of time as follows.

Definition 5 (PREM-busy)**.** A time window $[a, b]$ is PREM-busy if $\forall t \in [a, b]$ either the memory is busy or all cores are busy.

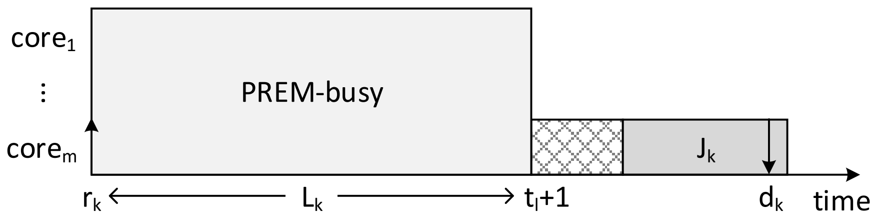


Figure 3.10: A necessary condition for a problem job J_k to miss its deadline.

Based on this definition, a necessary condition for J_k to miss its deadline is that the problem window $[r_k, t_l]$ has to be PREM-busy so that a 3-phase job is not able to execute. In Figure 3.10, we show a task system generating an interference that prevents J_k from running at or before t_l. We note that $t_l = d_k - e_k$ and $e_k = e_i^w + e_k^v + e_k^x$. As we mentioned earlier, e_i^w cannot be determined at compile time; thus, we assume e_i^w to be the largest unload phase. This is only for the problem job to determine the problem window length, we derive a better bound for the interfering tasks in Section 3.6.3.

Unlike traditional scheduling, the workload of each 3-phase task is not simply the sum of the execution time of its individual jobs because jobs have two phases, and memory phases must run exclusively. The idea behind our analysis is to collect all interfering jobs inside the problem window. Then, we derive from this collection of jobs a global bound on the total workload.

Giving an upper bound on the total workload is the subject of the following three sections. First, we show how to limit the amount of carry-in workload in Section 3.6.1. Second, we discuss in Section 3.6.2 how to compute $W_k^{CI}(\Gamma)$, the carry-in workload. Third, we show in Section 3.6.3 how to bound $W_k^{NC}(\Gamma)$, the workload without carry-in. We finally present the schedulability condition for gPREM in Section 3.6.4.

3.6.1 Carry-in Workload Limit

In order to limit the amount of work carried into the problem window, we consider a PREM-busy downward extension of the problem window in which there are at most m tasks that can have a carry-in jobs. The extended problem window has an earlier starting point t_o and shares the same endpoint t_l. The definition of t_o is the same as Definition 4. Based on this extension, we have the following lemma.

Lemma 3. *The time interval $[t_o, r_k)$ is PREM-busy.*

Proof. Assume a time interval inside $[t_o, r_k)$ which is not PREM-busy, i.e., both memory and at least one core are idle. Definition 4 states that inside $[t_o, r_k)$ either all cores are busy with tasks from $hp(k)$ or there is at least one pending task from $hp(k)$. Since gPREM is work-conserving, both cases contradict the above assumption. □

Since 3-phase tasks have memory and computation phases, we distinguish two types of carry-in. (1) Memory carry-in is a job with both memory and computation phases while (2) computation carry-in is a job with just a computation phase. With this definition, an important result is stated in the following lemma.

Lemma 4. *Only one task from Γ can have memory carry-in.*

Proof. As in proof of Lemma 2, we note that at $t_o - 1$ not all cores are busy with tasks from $hp(k)$ and there is no pending task from $hp(k)$. Therefore, it follows from Definition 4 that at t_o either all cores become busy with jobs from $hp(k)$ or one job from $hp(k)$ becomes pending. In either case, a new task from $hp(k)$ must be released exactly at t_o for either condition to become true. We now consider each case separately. (1) The release of a job from $hp(k)$ satisfies the first condition, i.e., it becomes the m^{th} task that make all cores busy. Since each task starts with a memory phase and only one memory phase can be active on the system at any time, it follows that all tasks that are executing at $t_o - 1$ must have completed their memory phases by t_o. Hence, there is no memory phase carry-in in this case. (2) A job from $hp(k)$ becomes pending after its release. We note that a job can be pending due to either all cores are busy with computation phases or one core is busy with memory phase. Since only one memory phase can be active at any time, as tasks are dispatched to their cores one after another, only one memory phase from $hp(k) \cup lep(k)$ can block this job. Similarly to the previous case, all other tasks that are executing at $t_o - 1$ must have completed their memory phases before t_o. Hence, at most one job can have memory carry-in, concluding the proof. □

Based on Lemma 4, we can have at most one memory carry-in from either $hp(k)$ or $lep(k)$. In the case where we do have a memory carry-in, we propose to extend t_o further downward such that it always start at the beginning of a memory phase t_e as in Figure 3.11. In this way, tasks can only have computation carry-in. However, if a task in $lep(k)$ was executing its memory phase at t_o, then t_e will correspond to the beginning of its execution; hence, in the worst case a single task in $lep(k)$ can have a body job within the problem window. The following lemma states that this extension of t_o does not change the property of the time interval $[t_e, r_k)$.

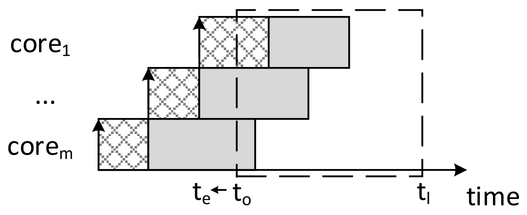


Figure 3.11: Only one task can have memory carry-in at t_o.

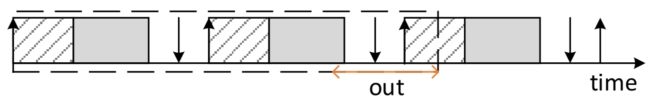
Lemma 5. *Extending t_o to start at the beginning of a memory phase t_e keeps the time interval $[t_e, r_k)$ PREM-busy.*

Proof. We recall from Definition 5 that an interval is PREM-busy if either one core is busy with a memory phase or all cores are busy with computation phases. Thus, extending t_o with memory phase keeps the time interval $[t_e, t_o)$ PREM-busy. □

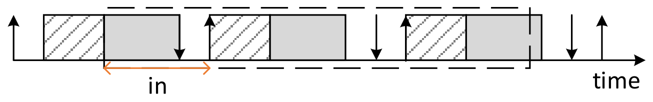
Finally, the following theorem summarizes the worst-case carry-in situation for the extension to t_e.

Theorem 1. *In the worst-case for gPREM, the carry-in workload at time t_e is limited by $m-1$ computation phases from $hp(k) \cup lep(k)$ tasks and one body job from $lep(k)$ tasks.*

Proof. Since tasks from $hp(k)$ can only have at most $m-1$ carry-in jobs at time t_o according to Lemma 2, the m$^{\text{th}}$ task is always from $lep(k)$. It is easy to see that Lemma 2 also applies to time t_e, since no job (outside of the one that possibly starts a memory phase at t_e) can start executing in $[t_e, t_o)$. Furthermore, no task can have memory carry-in, however, a single body job of a task in $lep(k)$ can now execute inside the window $[t_e, r_k)$, in which case the total number of carry-in jobs must be limited to $m-1$. Thus, the worst-case situation is to have $m-1$ computation carry-in jobs from tasks in either $hp(k)$ or $lep(k)$, plus either a computation carry-in from $lep(k)$ or a body job of a task in $lep(k)$. Since a body job includes both memory and computation, the latter case is the worst one, concluding the proof. □



(a) Without carry-in.



(b) With computation carry-in.

Figure 3.12: The workload in a window of interest.

3.6.2 Bounding $W_k^{CI}(\Gamma)$

Due to the non-preemptive execution, $m-1$ tasks of $lep(k)$ can have computation carry-in as in Theorem 1. The computation carry-in of $T_i \in lep(k)$ is computed as follows.

$$W_k^{CI}(T_i) = \min(e_i^x, L_k). \tag{3.8}$$

We limit the task's workload by L_k because tasks are sequential, and they cannot contribute for more than the problem window size.

In contrast, $hp(k)$ tasks can be activated and executed inside the problem window. To safely account for the workload of $hp(k)$ tasks, we always assume the worst-case activation such that carry-in jobs start as late as possible and carry-out jobs start as early as possible. As in Theorem 1, $m-1$ tasks of $hp(k)$ can have carry-in workload and all other $hp(k)$ tasks can have workload without carry-in.

We show the two types of workload in Figure 3.12. For no carry-in workload, the worst-case is to start the memory phase at t_e as in Figure 3.12(a). On the other hand, the worst-case is to start the computation phase at t_e for the workload with carry-in as in Figure 3.12(b). We recall that no task can have memory carry-in as in Theorem 1. In traditional scheduling, the carry-in workload is always larger than the workload without carry-in. Therefore, the carry-in workload can be computed as the difference between them [17]. In contrast, gPREM has the following two properties:

Property 1. The total amount of memory phases included in the problem window for the no carry-in case is no smaller than the carry-in case.

Property 2. The total amount of computation phases included in the problem window for the carry-in case is at most e_i^x larger than the no carry-in case.

The Property 1 immediately holds since in the no carry-in case a memory phase is started right at the beginning of the problem window, and the carry-in case can only introduce computation phase. Similarly, for Property 2, the computation phase starts immediately at the beginning of the window in the carry-in case, and this introduced computation phase pushes out memory and computation phases from the other end in the no carry-in case as marked by the horizontal left-right arrows in Figure 3.12.

Unfortunately, this observation complicates the analysis because the workload of memory and computation phases are non-comparable. As a result, we propose to consider one additional computation phase as the carry-in workload and computed as in (3.8). As in Theorem 1, the carry-in workload is limited by $m - 1$ computation phases from $hp(k) \cup lep(k)$. Thus, the total carry-in workload is computed as follows.

$$W_k^{CI}(\Gamma) = \sum_{largest(m-1)} W_k^{CI}(T_i). \tag{3.9}$$

3.6.3 Bounding $W_k^{NC}(\Gamma)$

Unlike the carry-in workload with only computation phases, the workload without carry-in can have both memory and computation phases. Thus, their workload is not simply their sum due to the restriction that no two memory phases can be active at the same time. Instead, we represent the workload of individual tasks $W_k^{NC}(T_i)$ as a set of jobs and each job is split into load, computation and unload phases. We populate $W_k^{NC}(T_i)$ for $hp(k)$ tasks by bounding the number of jobs inside the problem window as:

- $\lfloor \frac{L_k}{p_i} \rfloor$ body jobs.
- One carry-out job of size $\min(e_i^v + e_i^x, L_k \bmod p_i)$.

Even though Figure 3.12(a) shows load and computation phases only, the unload phases are implicitly induced by computation phases, i.e., we add an unload phase for each computation phase. Since the first m unload phases are carried in from jobs outside the problem window, we should consider the largest m unload phases to be combined with the first m load phases.

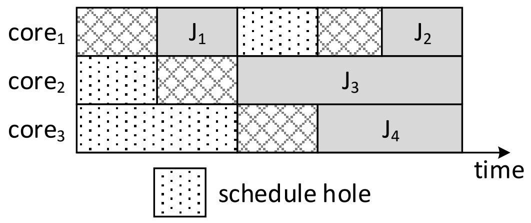


Figure 3.13: gPREM schedule with *schedule holes*.

We recall that one body job from $lep(k)$ tasks can execute inside the problem window as in Theorem 1. Since we need to consider only one job from $lep(k)$ tasks, and jobs have different lengths of load and computation phases, we cannot simply compare jobs and choose the maximum because the workload of memory and computation phases are not comparable. Instead, we propose to try all $lep(k)$ tasks one by one and take the one that leads to maximum workload. Alternatively, we can construct one representative job with largest load phase and largest computation phase to account for the worst-case.

We propose to construct from collected jobs a sequence of load phases (π_i), a sequence of computation phases (λ_i) and a sequence of unload phases (δ_i). We use the sequence (μ_i) to represent combined load and unload phases. We will explain how to construct these memory phases out of load and unload phases later in this section. Now, we let α to denote the length of both (μ_i) and (λ_i). It is clear that body jobs have equal number of memory and computation phases. However, for carry-out jobs, we might have a memory phase without computation. To guarantee α is equal for both (μ_i) and (λ_i), we set the corresponding computation phase to zero.

Definition 6. A schedule *hole* is a period of time whereby a core is idle because the memory is occupied by another core.

As we can see from Figure 3.13, gPREM schedule contains *holes* beside memory and computation phases. Schedule holes only exist when there is a memory phase. With this observation, we now define $W_k^{NC}(\Gamma)$ as in traditional scheduling, i.e., the total sum.

$$W_k^{NC}(\Gamma) = \sum_{i=1}^{\alpha} \mu_i + \sum_{i=1}^{\alpha} \lambda_i + holes. \tag{3.10}$$

Observe that $\sum_{i=1}^{\alpha} \mu_i$ and $\sum_{i=1}^{\alpha} \lambda_i$ above are completely defined for a given task set. However, *holes* is variable and dependent on jobs execution ordering.

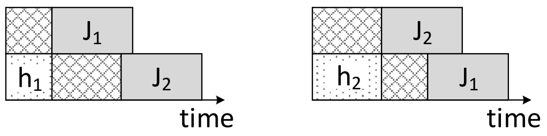


Figure 3.14: Based on the order of execution, the size of schedule hole changes ($h_1 < h_2$).

Definition 7. A schedule *overlap* is the amount of time by which computation phases run simultaneously with memory phases.

The overlap time in gPREM schedule is an important measure as we will see shortly. Note that schedule overlap is a dual definition of schedule hole in the sense that increasing one by a given amount will decrease the other by the same amount. We use both of them to simplify our arguments because in some cases it is more intuitive to use one than the other. Holes are related to overlap as follows.

$$holes = (m-1) \cdot \sum_{i=1}^{\alpha} \mu_i - overlap. \tag{3.11}$$

By substituting (3.11) in (3.10), we obtain:

$$W_k^{NC}(\Gamma) = m \cdot \sum_{i=1}^{\alpha} \mu_i + \sum_{i=1}^{\alpha} \lambda_i - overlap. \tag{3.12}$$

With gCONT, the contention global scheduler, the memory portion of the execution time is inflated by m. Thus, the total workload of gCONT is:

$$W_k^{cont}(\Gamma) = m \cdot \sum_{i=1}^{\alpha} \mu_i + \sum_{i=1}^{\alpha} \lambda_i. \tag{3.13}$$

From (3.12) and (3.13), we have:

$$W_k^{NC}(\Gamma) = W_k^{cont}(\Gamma) - overlap \tag{3.14}$$

Equation 3.14 shows an important result for gPREM when compared to gCONT. That is, as the amount of overlap increases, the advantage of gPREM accordingly increases.

Hence, our goal is to determine the amount of overlap, or alternatively the amount of holes as in Equation 3.10. However, with a set of α jobs, there are $\alpha!$ possible orderings of

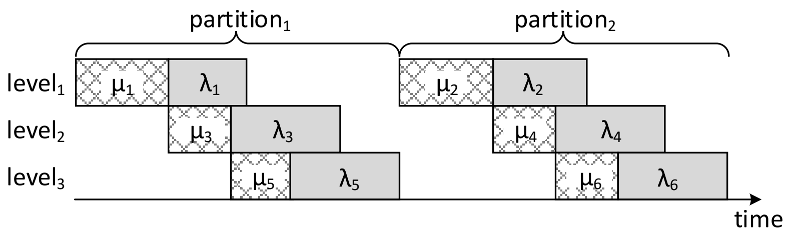


Figure 3.15: An example for computing a lower bound on *overlap*.

jobs. It is easy to see that each ordering leads to different workload because the amount of schedule holes changes, see Figure 3.14 for an example of two jobs released at the same time but executed in different order. In order to avoid such combinatorial complexity, we split memory phases from their computation phases. To obtain a lower-bound on the amount of overlap, we propose to order such computation and memory phases as follows.

We first sort load phases (π_i) such that $\pi_i \geq \pi_{i+1}$, and we sort (λ_i) such that $\lambda_i \leq \lambda_{i+1}$. We also sort unload phases (δ_i) such that $\delta_i \geq \delta_{i+1}$. We then let $\mu_i = \pi_i + \delta_i$, i.e., we combine largest unload phases with largest load phases to maximize the amount of holes, and we let $\rho = \alpha/m$. We assume for simplicity ρ is integer. However, this procedure can be easily extended for cases where α is not exact multiple of m by adding zeros to the tail of sorted (μ_i) and (λ_i) as this will not change the amount of holes or overlap.

Since the maximum amount of holes a memory phase μ_i can create is $(m-1) \times \mu_i$, we order the first ρ longest memory phases such that they have the maximum amount of holes. We then let the second ρ longest memory phases to overlap with the first ρ shortest computation phases and so on. This procedure divides the schedule horizontally into ρ partitions and m levels as in Figure 3.15. Before we show how to compute the amount of overlap for this schedule, we first consider a simple example.

Example 1. Figure 3.15 shows an example of 6 jobs and 3 cores. In this example, we have two levels to consider as memory phases of first level have no overlap. First, we consider the overlap of level 1 computation with level 2 memory, and level 2 computation with level 3 memory. This amount of overlap can be computed as: $\min(\lambda_1, \mu_3) + \min(\lambda_3, \mu_5)$ for the first partition and $\min(\lambda_2, \mu_4) + \min(\lambda_4, \mu_6)$ for the second partition. Second, we consider the overlap of level 1 computation with level 3 memory. This amount of overlap can be computed as: $\min((\lambda_1 - \mu_3)_0, \mu_5)$ for the first partition and $\min((\lambda_2 - \mu_4)_0, \mu_6)$ for the

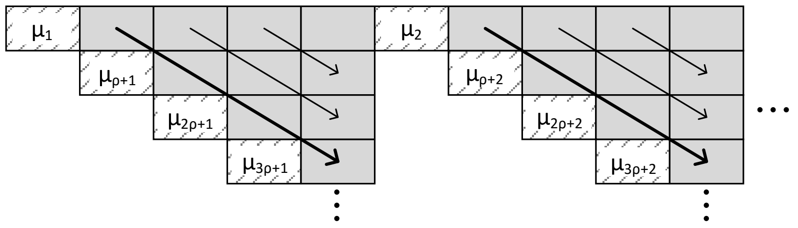


Figure 3.16: The way *overlap′* is computed.

second partition. We need to subtract the memory phase portion μ_3 and μ_4 because we accounted for them before.

The following equation generalizes the above example to compute a lower-bound on the amount of overlap for n jobs and m cores.

$$\begin{aligned} overlap' = & \sum_{j=1}^{(m-1)\rho} \min(\lambda_j, \mu_{\rho+j}) + \\ & \sum_{j=1}^{(m-2)\rho} \min((\lambda_j - \mu_{\rho+j})_0, \mu_{2\rho+j}) + \\ & \vdots \\ & \sum_{j=1}^{\rho} \min((\lambda_j - (\cdots + \mu_{(m-2)\rho+j}))_0, \mu_{(m-1)\rho+j}). \end{aligned} \tag{3.15}$$

The first term in (3.15) computes the overlap between one memory level and the immediate computation level above. The second term computes the overlap between one memory level and the computation of two levels above, and so on. See Figure 3.16 for an illustrative diagram where each level of diagonal arrows represents one term in above equation. For instance, the bold-line arrows represent the first term in (3.15). We note that in the first term, we take the minimum between the computation and the memory phase $\min(\lambda_j, \mu_{\rho+j})$. For the second term, we need to subtract the memory phase portion $\mu_{\rho+j}$ because it is already been accounted for in the first term. We continue this process by subtracting the overlap computed from previous terms. The following Lemma states that the above procedure indeed computes a lower-bound on the amount of overlap.

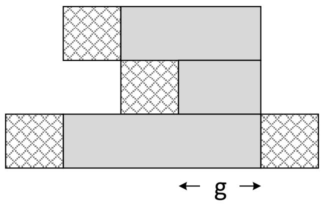


Figure 3.17: The gaps between memory phases increase the amount of overlap.

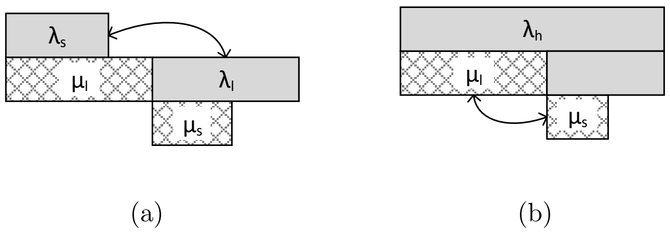


Figure 3.18: An illustrative diagrams to help proving lemma 6.

Lemma 6. *The order of memory and computation phases as in Figure 3.15 is the worst-case that leads to a minimum amount of overlap or alternatively, a maximum amount of holes.*

Proof. We will prove this lemma in three steps.

(1) The situation presented in Figure 3.15, where each partition comprises a continuous chain of memory phases, leads to a minimum amount of overlap. We argue that introducing gaps between memory phases increases the amount of overlap. We recall that the problem window has to be PREM-busy in order for the problem job to miss its deadline. Therefore, all processor cores must be busy in computation phases within the introduced gap to hold the condition. Figure 3.17 shows that the gap g leads to an increased amount of overlap compared to the case where there is no gap between memory phases of the same partition as in Figure 3.15.

(2) Letting shortest computation phases to overlap with longest memory phases leads to a minimum amount of overlap. We use Figure 3.18(a) to guide our proof. We have two memory phases $\mu_l > \mu_s$, and two computation phases $\lambda_l < \lambda_s$. By contradiction, we assume if we exchange λ_s with λ_l, the amount of overlap is reduced. We let $a = min(\lambda_s, \mu_l)$ and $b = min(\lambda_l, \mu_s)$ before the exchange. Hence, the amount of overlap is $a + b$.

There are four cases after the exchange based on the two terms a and b: (1) both remain the same. (2) a increases and b remains the same (3) a remains the same and b decreases. (4) a increases and b decreases. Note that b can not increase and a can not decrease after the exchange because $\lambda_s < \lambda_l$. Obviously, the first two cases will not reduce the amount of overlap. Thus, we only need to consider the last two cases.

Case(3): To satisfy this case, we should have $\lambda_s > \mu_l$ and $\lambda_s < \mu_s$, but we obtain $\mu_s > \mu_l$ which is a contradiction.

Case(4): This case is more elaborate than the previous one. In order to satisfy this case, we should have $\lambda_s < \mu_l$ and $\lambda_s < \mu_s$. Now, we consider two sub-cases. (i) $\mu_s < \lambda_l$: based on these assumptions, the overlap before the exchange is $\lambda_s + \mu_s$ and after the exchange is $\mu_l + \lambda_s$. Clearly, this sub-case increases the overlap. (ii) $\lambda_l < \mu_s$: as in previous sub-case, the overlap before the exchange is $\lambda_s + \lambda_l$ and after the exchange is $\lambda_l + \lambda_s$ which keeps the amount of overlap unchanged. After we examine all possible cases, we can conclude that our initial argument is true. That is, when we overlap shortest computation phases with longest memory phases, we obtain a lower bound on the amount of overlap.

(3) The order of memory phases within each partition leads to a minimum amount of overlap. As a proof sketch, we consider the case as in Figure 3.18(b) in which λ_h spans over both memory phases. In this case, the overlap is $\mu_l + 2\mu_s$. If we change the order by starting with shortest memory, the overlap becomes $\mu_s + 2\mu_l$ which clearly leads to an increased amount of overlap. □

Finally, the following theorem summarizes the results for total workload.

Theorem 2. *$W_k(\Gamma) = W_k^{NC}(\Gamma) + W_k^{CI}(\Gamma)$ is an upper-bound on the workload of all tasks in Γ within T_k problem window including memory, computation and holes.*

Proof. Equation 3.12 is an upper-bound on $W_k^{NC}(\Gamma)$ if overlap is substituted by *overlap*$'$ which is a lower-bound on overlap as in Lemma 6. In addition, an upper-bound on $W_k^{CI}(\Gamma)$ is computed as in Equation 3.9. Since the carry-in workload has no memory phases and therefore no schedule holes, their sum is an upper bound on the total workload $W_k(\Gamma)$. □

3.6.4 Schedulability Condition

So far, we specified the necessary condition for a 3-phase job to miss its deadline. That is, the problem window has to be continuously PREM-busy to prevent the problem job from running. As we defined early, an interval of time is PREM-busy if one core is busy with a memory phase or all cores are busy with computation phases. The complement of this definition states that during a non PREM-busy time interval, at least one core is idle and all other cores execute computation phases. In this case, the problem job can acquire this idle core and meet its deadline.

In what follows, we give the schedulability condition for gPREM by starting with the following lemma.

Lemma 7. *If J_k misses its deadline and the start of the problem window is t_e as defined in Section 3.6.1, then $W_k(\Gamma) \geq m \times L_k$.*

Proof. Assume by contradiction that J_k misses its deadline and $W_k(\Gamma) < m \times L_k$. The time interval $[r_k, t_l]$ has to be PREM-busy in order for J_k to miss its deadline; otherwise, J_k could have executed and finished before its deadline as gPREM is work-conserving. In addition, we proved in Lemma 3 and Lemma 5 that the time interval $[t_e, r_k)$ is PREM-busy. As a result, the time interval $[t_e, t_l]$ has to be PREM-busy for J_k to miss its deadline. Theorem 2 states that $W_k(\Gamma)$ is an upper bound on the workload of all tasks including computation, memory and holes. Since $W_k(\Gamma) < m \times L_k$, then there must exist a time in the window of length L_k starting at t_e that is not PREM-busy. Furthermore, since the size of $[t_e, t_l]$ is at least L_k (given that $t_e \leq r_k$), it follows that there must exist a time in $[t_e, t_l]$ which is not PREM-busy. This creates a contradiction, hence the lemma follows. □

Theorem 3. *If $W_k(\Gamma) < m \times L_k$ for all $T_k \in \Gamma$ then the system is schedulable.*

Proof. It follows from the contrapositive of Lemma 7 that is if $W_k(\Gamma) < m \times L_k$ for any task, then this task must meet its deadline. Thus, if this condition holds for all tasks in Γ, the system is schedulable. □

The computational complexity of our analysis can be derived as follows. We recall that the first step in our analysis is to sort memory and computation phases in opposite directions. Since each task has three types of jobs: carry-in, body and carry-out, sorting these phases can be done in $O\big(n \cdot \log(n)\big)$. Furthermore, computing the overlap in (3.15) involves $m\alpha$ of additions and min / max operations. We note that the number of phases α depends on the number of jobs rather than the number of tasks n. That is, for a given window of time, the number of jobs is dependent on the number of tasks as well as their period relations. Since these operations and sorting are done in two steps, their complexity can be combined as $O\big(n \cdot \log(n)\big) + O(m\alpha)$. In general, our analysis is designed to run off-line; hence, we deem such complexity acceptable.

Our analysis is also *sustainable* in the sense that if tasks execute for less than their WCET, the computed workload is still an upper bound. This can be easily proven by observing that the amount of overlap either remains constant or reduces if the actual computation or memory phases are executed for less than their worst-case values.

3.7 Task Splitting

We assumed so far that the code and data of each task can fit inside the local memory of any core. However, this assumption can be restrictive for large real-time applications. In this section, we turn our discussion to the case in which the task size is larger than the local memory. To mitigate such limitation, we propose to split each task into multiple segments such that each segment can fit inside the local memory. With task splitting approach, each segment has three phases as in single segment case, and each segment is executed non-preemptively. Unlike single segment task, the multi-segment task can be preempted between its segments in places we refer to as *preemption points*. Furthermore, the first segment is released like a normal task and the subsequent segment is released after the finish time of previous segment. Jobs of the same task are assumed to produce the same number of segments and each segment inherits the priority of the producing task.

Task-splitting is quite popular in practice and supported by several commercial tools. For instance, it is adopted in time-triggered architectures to mitigate the allocation problem of tasks to cycles [33]. The recent work in [110] shows how to break a program into non-preemptive segments through compiler analysis. Moreover, there has been a significant amount of work in literature that proposes techniques to manage the local memory of one task [50, 90, 13] for code and data, including stack and heap. Our focus in this section, however, is how to provide a timing guarantee for these globally scheduled jobs rather than showing how to divide a task into multiple segments.

The task model is similar to before but now each job J_i is split into s_i non-preemptive segments. We use $J_{i,j}$ to denote the j^{th} segment of J_i. The total execution time for each segment, including both memory and computation phases, is denoted by $e_{i,j}$. Similar to single segment case, we let t_l refers to the latest time for J_{k,s_k} to execute and finish before the deadline, i.e., $t_l = d_k - e_{k,s_k}$. We use L_k to denote the length of $[r_k, t_l]$ window. We now seek the necessary condition for J_{k,s_k} to miss the deadline. With task splitting, segments are released one after another, i.e., no two segments can be released at the same time. This is important to avoid intra-task interference. Thus, the interference on J_k can be defined as the cumulative length of all intervals in which the cores are PREM-busy executing segments other than the segments of J_k.

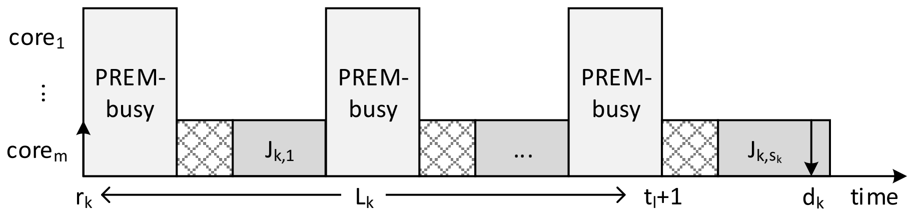


Figure 3.19: A necessary condition for J_k to miss its deadline.

The analysis of multi-segment case is different than single segment case in many aspects. We split our discussion into four sub-sections to address each aspect.

3.7.1 The Problem Window of Multi-segment Job

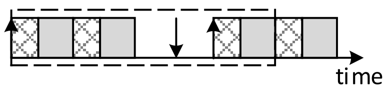
We demonstrate in Figure 3.19 the problem window for a job with multiple segments, $\{J_{k,1}, \ldots, J_{k,s_k}\}$. Here, the PREM-busy intervals are not necessarily contiguous as in single segment case. The necessary condition for J_k to miss the deadline can be stated as follows.

$$W_k(\Gamma) \geq m \times Z_k, \tag{3.16}$$

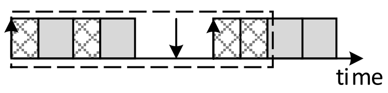
where $Z_k = L_k - \sum_{j=1}^{s_k-1} e_{k,j}$. For single segment case, $s_k = 1$ and $L_k = Z_k$. In contrast, $L_k > Z_k$ for multi-segment case. We note that L_k is the problem window length to collect interfering jobs and Z_k is used in the schedulability test as in (3.16).

3.7.2 The Workload of Multi-segment Carry-out Job

A sustainable analysis requires that if tasks run for less than their WCET, the computed upper bound should still be valid. In our case, however, the workload of multi-segment 3-phase tasks can be larger if the computation phases are run for less than their WCET. For example, consider the carry-out job in Figure 3.20(a), and assume the computation phase of the first segment is executed for zero time. This effectively allows the memory phase of the second segment to execute inside the problem window. Thus, the amount of workload can increase because the workload of memory phases are larger than the computation phases as per the following lemma.



(a) Without compacting memory phases.



(b) With compacting memory phases.

Figure 3.20: The workload of multi-segment carry-out job.

Lemma 8. *The workload of J_a with $(\mu + \Delta)$ memory and $(\lambda - \Delta)$ computation is greater than or equal the workload of J_b with μ memory and λ computation.*

Proof. Assume the system workload is Σ before we consider J_a or J_b. The workload of J_a is $(\mu+\Delta) \times m + (\lambda - \Delta) - a$ where a is the amount of overlap that J_a introduces to system workload. Similarly, the workload of J_b is $\mu \times m + \lambda - b$ where b is the overlap introduced by J_b. The lemma claims the following:

$$\Sigma + (\mu + \Delta) \times m + (\lambda - \Delta) - a \geq \Sigma + \mu \times m + \lambda - b.$$

After simplifying the terms, we have:

$$a \leq b + \Delta \times (m - 1) \leq b.$$

It remains to prove that $a \leq b$ is true. We recall that the procedure for extracting a lower bound on the amount of overlap sorts computation phases from short to long, and the ρ longest computation phases are discarded from consideration, i.e., they do not contribute to the overlap. Since $\lambda \geq (\lambda - \Delta)$, λ may be discarded and $\lambda - \Delta$ may be completely overlapped in the worst case. Now, we consider the case in which we add J_a to the workload. Since $\lambda - \Delta$ is overlapped, this causes another computation phase λ^* with length of $\lambda - \Delta$ or larger to be discarded. On the other hand, for the case in which we add J_b, this computation phase λ^* would replace λ which we assume to be discarded. Thus, a will never exceed b. □

To avoid this problem of carry-out job, we propose to compact the memory phases of all segments at the beginning, then we compact the computation phases afterwards as shown in Figure 3.20(b). It is easy to see that, for a given window of time, the amount of memory in Figure 3.20(b) is always larger than or equal the one in Figure 3.20(a). Hence, this bound on the workload of carry-out jobs is safe as per Lemma 8.

In what follows, we discuss the amount of workload that can execute in the problem window of J_k. Since with task splitting $lp(k)$ tasks can interfere between the segments of J_k, we distinguish two types of intervals: the beginning of the window and between segments (the middle intervals).

3.7.3 The Workload at the Beginning Interval

In this section, we show how to compute $W_k^{CI}(T_i)$ for $hp(k)$ and $lep(k)$ tasks. We observe that the extended problem window that starts at t_e has the advantage to limit the number of carry-in workload to $m-1$ computation phases from $lep(k) \cup hp(k)$. However, with multi-segment tasks, this only limits the first segment of $hp(k)$ tasks, and the subsequent $s_i - 1$ segments including both memory and computation phases can still execute inside the problem window. In single segment case, the carry-in workload is made of computation phases and was easily computed as in (3.9). In contrast, the carry-in workload of $hp(k)$ tasks in multi-segment case is made of memory and computation phases. Since tasks have different lengths of memory and computation phases, the maximum workload of $m-1$ jobs can be determined by trying $\binom{n}{m-1}$ possible combinations. We recall that the amount of overlap, which determines the amount of workload, is variable and its value is dependent on the relative lengths of memory and computation phases of all jobs.

Alternatively, we propose the following approach to mitigate the above combinatorial complexity. We recall that the procedure described in Section 3.6 tries to extract the amount of overlap to reduce the total workload. Here, instead, we propose to assume each memory phase of carry-in jobs does not overlap with computation phases which is clearly an upper bound on its maximum workload. In other words, each memory phase μ_i is assumed to have $m \times \mu_i$ amount of workload. Since the segments of carry-in jobs will execute with the segments of no-carry-in jobs, the amount of overlap may decrease. To ensure a safe lower bound on the amount of overlap as in (3.15), we add to the sequence of computation phases (λ_i) another list of computation phases constructed as follows. From $hp(k)$ tasks, we collect the smallest S of $e_{i,j}^x$, where $S = \sum_{largest(m-1)} s_i$, an upper bound on the number of computation phases that can execute inside the problem window.

Similar to single segment case, we first determine the interval of time in which a carry-in job can execute. However, the difference here is that we need to account for both memory and computation workload. In addition, similar to carry-out jobs, we need to account for memory workload first to ensure sustainability. Let $a_k^{CI}(T_i)$ and $a_k^{NC}(T_i)$ denote the amount of workload with and without carry-in, respectively, in which each job is combined into one phase by adding the length of computation and memory phases. Similarly, let $b_k^{CI}(T_i)$

and $b_k^{NC}(T_i)$ denote the amount of workload with and without carry-in, respectively, in which each job has memory phases only. In addition, let $c_k^{CI}(T_i) = a_k^{CI}(T_i) - b_k^{CI}(T_i)$ and $c_k^{NC}(T_i) = a_k^{NC}(T_i) - b_k^{NC}(T_i)$. We use $\Delta_k^b(T_i)$ and $\Delta_k^c(T_i)$ to denote the difference between the workload with carry-in and without carry-in for each case, respectively. Now, we have for each $T_i \in hp(k)$

$$W_k^{CI}(T_i) = m \times \Delta_k^b(T_i) + \Delta_k^c(T_i). \tag{3.17}$$

For $lep(k)$ tasks, we let $Q_i = \max(e_{i,j}^x)$ be the maximum computation phase among the segments of T_i. Thus, we have $W_k^{CI}(T_i) = \min(L_k, Q_i)$. Finally, the total carry-in workload is computed as in (3.9). In summary, the workload at the beginning interval is (1) one segment from $lep(k)$ including both memory and computation and (2) the maximum $m-1$ of either computation phases from $lep(k)$ or s_i computation phases and $s_i - 1$ memory phases from $hp(k)$. We note that this amount of workload reduces to single segment case if we let $s_i = 1$.

3.7.4 The Workload at the Middle Intervals

Our analysis assumes that when J_k executes, the remaining $m-1$ cores are idle as per Definition 3. Thus, $m-1$ tasks of $lp(k)$ can acquire the other cores during the execution of J_k and push some workload into the problem window of J_k due to the non-preemptive execution. The following Lemma summarizes the workload of $lp(k)$ tasks between each two segments.

Lemma 9. *Between each two segments of J_k, $lp(k)$ tasks can interfere with one segment including both memory and computation phases and $m-2$ segments with only computation phases.*

Proof. We show in Figure 3.21 an example of two consecutive segments: $J_{k,x}$ and $J_{k,x+1}$. The $lp(k)$ tasks can interfere with at most one segment with memory and computation phases. This can result from a task released just one time unit before the end of $J_{k,x}$ computation phase. Since $J_{k,x+1}$ is released immediately after $J_{k,x}$, no $lp(k)$ task can start its memory after $J_{k,x}$. In addition, other $lp(k)$ jobs can acquire the $m-2$ cores during the execution of $J_{k,x}$ and push their computation phases as in Figure 3.21. Although $core_m$ is idle immediately after $J_{k,x}$, $hp(k)$ jobs can execute and delay the next segment, $J_{k,x+1}$. $\square$

We observe that the memory phase of one $lp(k)$ segment is guaranteed to overlap with $m-2$ computation phases of $lp(k)$ as in Figure 3.21. Thus, the total workload for one

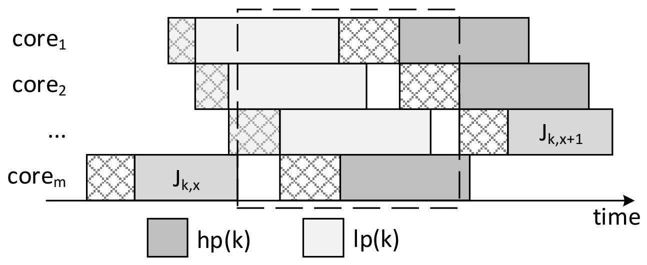


Figure 3.21: The workload of $lp(k)$ tasks in a middle interval.

middle interval can be computed as follows:

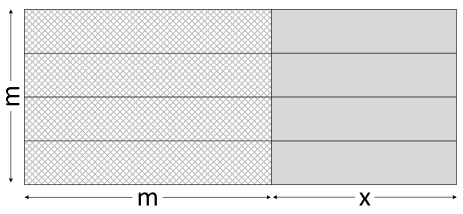
$$\begin{aligned} A_k &= m \times \mu_k^{max} + \max(Q^k) \\ &+ (\sum_{j=2}^{m-1} \max_j(Q^k) - (m-2) \times \mu_k^{max})_0, \end{aligned}$$

where μ_k^{max} is the maximum memory phase in $lp(k)$ segments, $Q^k = \{Q_{k+1}, \cdots, Q_n\}$ is the set of largest computation phases of $lp(k)$ segments and $\max_j$ is the j$^{\text{th}}$ largest value. For a job with s_k segments, there are $s_k - 1$ middle intervals. Thus, the total workload of $lp(k)$ tasks is $(s_k - 1) \times A_k$.

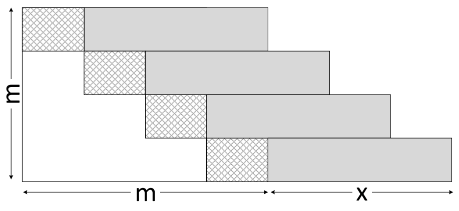
Based on above discussion, we observe the following differences for multi-segment case over single segment.

- Since we split tasks based on the size of local memory, the task's phases are now shorter and close in length.
- Tasks can have memory carry-in.
- The size of problem window is larger.

Clearly, the first point favors the multi-segment case while the last two points favor the single segment case. The reason for the first point is that our analysis combines the largest memory phases with shortest computation phases. With these phases being close in length, the pessimism in the analysis decreases. In addition, our analysis takes the maximum carry-in workload. With large tasks being split into smaller segments, this reduces the amount of carry-in workload. For example, consider two tasks T_a and T_b, and for simplicity, assume



(a) gCONT



(b) gPREM

Figure 3.22: The workload of gCONT and gPREM.

they have only computation phases of length 90 and 30 time units, respectively. Now, assume that T_a is split into 3 segments of length 30 each and T_b is not split. The amount of carry-in workload of T_a in the problem window of T_b is 90 without split and 30 after split.

3.8 Evaluation

In this section, we evaluate gPREM against gCONT. We use the most recent schedulability test [64] for gCONT. Before we show the evaluation results, we intuitively discuss the difference between gPREM and gCONT.

3.8.1 An Intuitive View

In this section, we show an example of the workload under gPREM and gCONT. Assume a given m jobs with equal memory phases of one time unit and equal computation phases of $x = m - 1$ time units. We seek to compute the total workload of these m jobs under each scheduling scheme. In Figure 3.22(a), we show gCONT schedule in which memory requests are assumed to occur simultaneously across all cores. Therefore, the total workload is computed as:

$$W^{cont} = m^2 + m \times x.$$

In contrast, Figure 3.22(b) shows gPREM schedule. In this case, the upper half of memory phases is overlapped with computation phases. Hence, the total workload for gPREM is computed as:

$$W^{prem} = m^2 + \frac{x \times (x+1)}{2}.$$

Table 3.2: Benchmarks

benchmark	code (byte)	data (byte)	CPU (cycle)
a2time	3108	5420	100497
rspeed	1956	6864	55688
canrdr	2724	8030	47280
corner-turn	2032	8192	16726
transitive	2080	3024	102898

The second term represents the series $x+(x-1)+(x-2)+\ldots+1$ which corresponds to the staircase of computation phases that do not overlap with memory phases. The speedup factor is the following ratio:

$$S = \frac{W^{cont}}{W^{prem}} = \frac{m^2 + m \times x}{m^2 + \frac{x\times(x+1)}{2}}.$$

If we substitute x by $m-1$, we obtain

$$S = \frac{4m^2 - 2m}{3m^2 - m}.$$

Thus, the maximum speedup with large number of cores can reach up to $\lim_{m\to\infty} S = 4/3$. This is a speedup limit for this particular example in which the memory phases are equal and the computation phases are large enough to overlap with memory phases.

3.8.2 Results

The evaluation is based on real data taken from the automotive suite of EEMBC [111] and DIS benchmark suite [101]. The set of benchmarks are listed in Table 3.2. The selection is aimed to represent several applications used in embedded real-time domain and memory intensive applications are chosen to better stress the memory. The size in bytes of code and data for each benchmark is reported in the first two columns. In addition, each benchmark is executed on a NIOS-II processor after loading its code and data into the local memory of the processor, i.e., without main memory stalls. This computation time is reported in clock cycles in the third column.

In our experiments, the rate monotonic policy is used to assign priorities to tasks. We construct task sets from the tasks in Table 3.2 as follows. For a given value of Ut, we randomly choose a task, then we keep adding tasks until either the value of Ut is reached or $Um = 1$. In addition, we generate 1000 task sets and report the average value for each utilization point. The experiments are repeated 100 times and the error bars represent the first decile and the ninth decile. We use the following parameters to characterize the generated task sets.

1. The **total utilization** Ut is varied from $0.025m$ to $0.975m$ in steps of size $0.025m$.
2. The task **periods** p_i are chosen from uniformly distributed values within $[200, 1000] \times 1000$ clock cycles.
3. Since we report the memory demand of each task in bytes, we use **memory latency** as a parameter to control the length of memory phases. We set the default value of memory latency as 1 cycle per byte.
4. The **local memory size** is used as a parameter to control the number of segments for each task. We note that the tasks of automotive suite of EEMBC benchmarks process streams of data; thus, their computation times increase with the size of input data. Such property allows us to re-size tasks (their memory and computation phases) based on the size of local memory. We let σ_i to be the ratio between memory to computation phase for each benchmark. For a given local memory size, we split each benchmark into segments of same size that fit into this local memory. Then, we let the ratio between memory to computation of these segments equals to σ_i.
5. The **number of cores** m is set to 8 as a default value.

In the first set of experiments, we use the *acceptance ratio* metric to evaluate the schedulability of gPREM against gCONT. We also include in this experiment gNOCT, a non-preemptive global scheduler in which there is no contention for access to main memory, i.e., the memory portion is not inflated as in gCONT. We note that this scheduler is not realistic but we include it to show the importance of considering memory contention. The results show that gPREM has higher schedulability than gCONT. In addition, the schedulability difference between gCONT and gNOCT is high, indicating that memory contention indeed has a large impact. Even though gPREM shows better schedulability than gCONT, we cannot claim that gPREM always dominates gCONT. We recall that the unload phase of problem job is assumed to be the largest in the system since it cannot be determined at compile-time. Thus, this unload phase can cause the memory phase of one job to be larger than the inflated memory under gCONT.

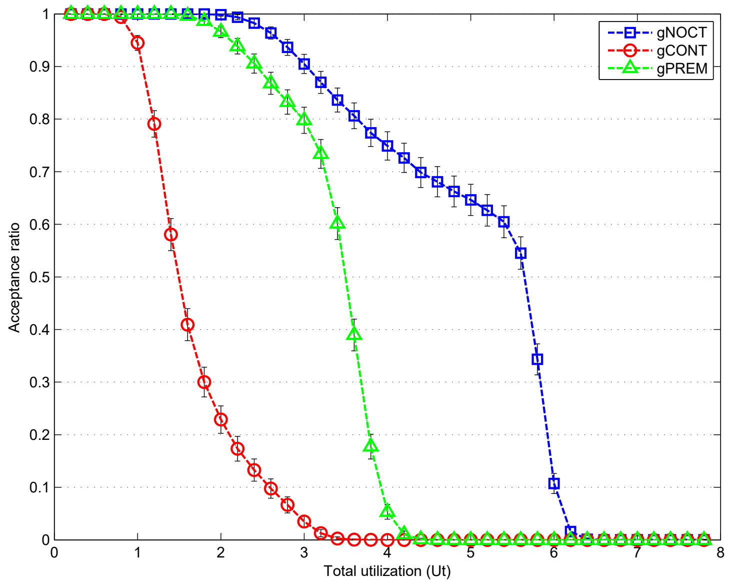


Figure 3.23: The schedulability of gPREM against gCONT and gNOCT.

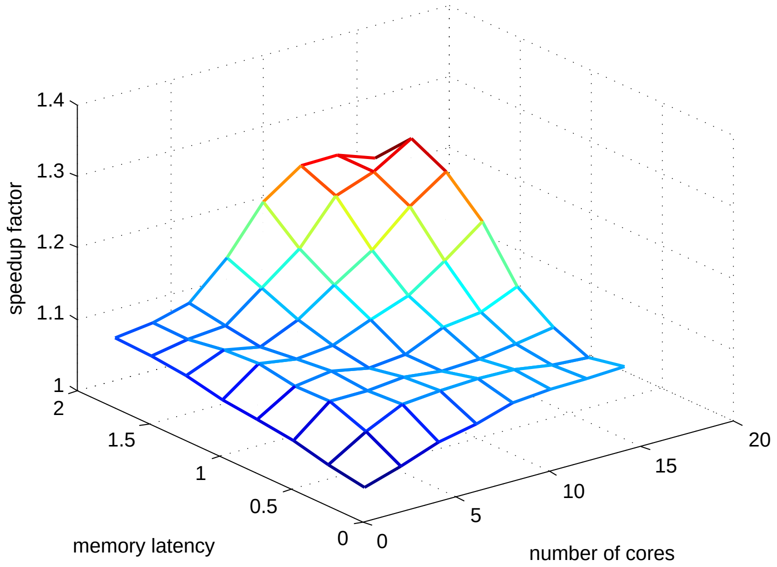


Figure 3.24: The speedup factor by varying m and memory latency.

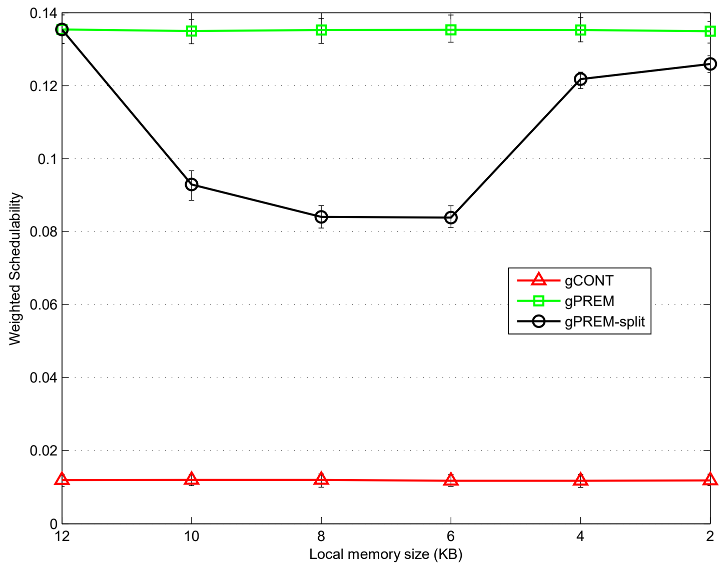


Figure 3.25: Weighted schedulability as a function of local memory size.

In the second set of experiments, we choose a task with middle priority and compute its response time under gPREM and gCONT. A middle priority task is chosen to account for both types of workloads: $hp(k)$ and $lep(k)$. Unlike the previous experiment, we quantify the performance gain using the speedup factor. It is the ratio between the response times of this middle priority task under gCONT and gPREM. Figure 3.24 shows the results by varying m and memory latency. For each m value, we set $Ut = m$. We note that increasing Ut or memory latency has the same effect, increasing Um until reaching the maximum value of 1. The maximum speedup factor of 1.25 is achieved when both Um and Ut is high, i.e., when there is a balanced load of memory and computation.

In the third set of experiments, we evaluate the multi-segment case (gPREM-split). We use weighted schedulability [23] as the evaluation metric. We can think of weighted schedulability as the area under the schedulability curve, i.e., each point in Figure 3.25 represents the area under the curve in Figure 3.23. We vary the local memory size in the x-axis. The results in Figure 3.25 show that the schedulability of muti-segment case is lower than single segment case but still higher than gCONT. In previous section, we hypothesized three points of differences between single and multi-segment tasks. These points explain the curve of multi-segment case. The general trend is that the schedulability difference between single and multi-segment tasks decreases as we increase the local memory size or, alternatively, decrease the number of segments. However, we notice at the beginning with small local memory sizes, the multi-segment case has an advantage (the first point of the differences). As we increase the local memory size, this advantage diminishes.

3.9 Summary

We propose gPREM, a new approach to globally schedule 3-phase tasks on multicore processor. A new concept of *schedule hole* is introduced to account for the workload generated by memory phases. We showed that by overlapping computation phases with memory phases, we can hide some of memory latency and therefore enhance the system performance.

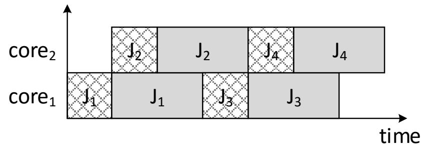
Chapter 4

DMA Global Scheduling of 3-phase Sequential Tasks

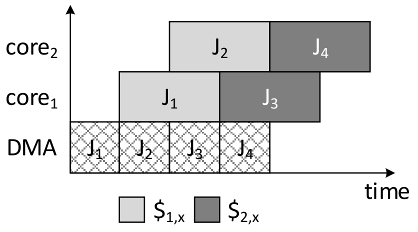
With gPREM, the processor cores are stalled while loading from or unloading to main memory. To mitigate such limitation and increase the system performance, we propose to use a DMA component to execute the memory phases. In addition, the local memory of each core is divided into two partitions so that while the DMA is loading one task into one partition of local memory, another task can be concurrently executed out of the second partition without memory stalls. In this way, the system performance is increased by overlapping memory transfers with processor execution for tasks within the same core.

We refer to this memory efficient global scheduler as gDMA. With m cores, there are $2 \times m$ identical local memory partitions. We refer to the first partition in the local memory of $core_x$ as $\$_{1,x}$ and the second partition as $\$_{2,x}$. We further assume that each local memory is double-ported so that the processor core and the DMA can simultaneously access different partitions of the local memory without causing timing interference (contention). In fact, MPC5777M platform supports stall-free operations between core and DMA. We have verified this by experimentation [119] and found that both the core and the DMA do not suffer any delay when they access the SPM simultaneously.

In Figure 4.1, we show an example schedule of four jobs and two cores for gPREM and gDMA. In Figure 4.1(a), $core_1$ is stalled while executing the memory phase of J_3. Similarly, $core_2$ is stalled while executing the memory phase of J_4. In contrast, there is no stalling in Figure 4.1(b) for gDMA where the computation phases are executed immediately after each other. In this case, the memory phases of J_3 and J_4 are completely hidden by overlapping them with computation phases of both cores. We note that gPREM can only



(a) gPREM schedule.



(b) gDMA schedule.

Figure 4.1: The memory phases of J_3 and J_4 are completely hidden.

overlap memory phases with computation phases of other cores. In contrast, gDMA can overlap memory phases with computation phases of the same core as well as other cores.

In this chapter, we derive a novel schedulability analysis for gDMA. We note that the system has two types of active components: processor cores and DMA in which cores execute computation phases and DMA executes memory phases. The nature of co-scheduling DMA and processor cores requires us to largely re-define the existing concepts of workload and interference in global real-time scheduling. In particular, we introduce a new concept of *scheduling interval* to account for the workload generated by both memory phases and computation phases.

We use $J_a \rightarrow J_b$ to denote that J_b is the next job to run immediately after J_a on the same core. We use $t_s(J_a)$ to denote the start time of the computation phase of J_a, and $proc(J_a)$ to denote the core where J_a is executed. Beside the notations we introduce in this chapter, we will use the same notations as in Table 3.1 of Chapter 3.

4.1 Scheduling Algorithm

Similar to gPREM, we propose to combine the unload phase of one job to the load phase of the next job to be executed out of the same partition. Suppose that $J_a \rightarrow J_b \rightarrow J_c$. We note that the consecutive execution of jobs on the same core alternate between its local memory partitions. Hence, J_c is the next job to execute after J_a out of the same partition. Then, when J_c is scheduled to be executed on the DMA, the DMA executes the unload phase of J_a non-preemptively with the load phase of J_c. For simplicity, we refer to the combined unload and load phases as the *memory phase* for J_c. In addition, both memory

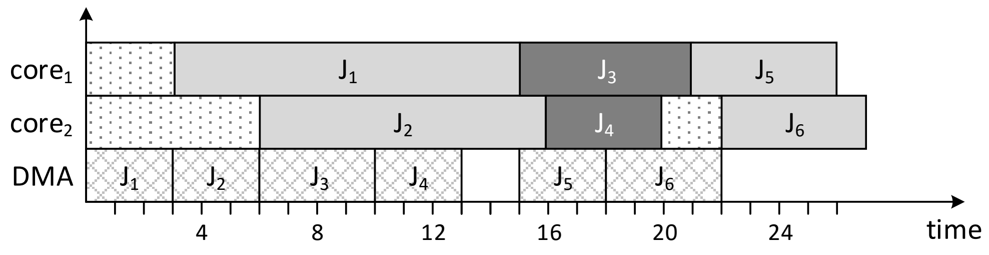


Figure 4.2: An example of gDMA schedule of 6 jobs on 2 cores.

phases and computation phases are executed non-preemptively. We note that the memory phase and the computation phase of one task are not necessarily executed continuously as in gPREM because after loading a task, the core might be busy executing non-preemptively another task out of the other partition. However, after loading a job into a local memory partition, its content is locked until the finish time of its computation phase.

Example 2. Figure 4.2 shows an example for scheduling 6 jobs on 2 cores. These six jobs are from different tasks and the job priority is indicated by its index such that J_a has higher priority than J_b if $a < b$. In addition, these jobs are assumed to be released at the same time, at time 0. Since gDMA is a fixed-priority scheduler, the highest priority job J_1 is chosen first. The scheduler chooses $core_1$ to execute J_1. The DMA is instructed to unload the previous job J_x from $\$_{1,1}$ back to main memory. Then, the DMA is instructed to load J_1 into $\$_{1,1}$. After that, J_1 is able to run on $core_1$ with no memory stalls. While $core_1$ is executing J_1 out of $\$_{1,1}$, the scheduler at time 3 chooses J_2 to be executed on $core_2$. Here, the DMA is running in parallel with J_1 by unloading $\$_{1,2}$ and loading it with J_2. At time 6, the scheduler chooses the free partition of $core_1$ to execute J_3. Similarly, J_4 is chosen at time 10 to execute on the free partition of $core_2$.

At time 13, all four partitions are loaded. Hence, the memory phase of J_5 has to wait until time 15, the finish time of the computation phase of J_1 which indicates that $core_1$ has again a free partition. Thus, the scheduler at time 15 chooses J_5 to be executed on $core_1$. Finally, J_6 is scheduled at time 18 to execute on $core_2$. We note that even though $core_2$ has finished execution at time 20, J_6 has to wait until time 22 because its memory phase is delayed. This delay induced a schedule *hole* between J_4 and J_6. We define a schedule hole as the time at which the core is idle waiting for a task to be loaded.

As shown in Figure 4.2, the memory phases are largely overlapped with computation

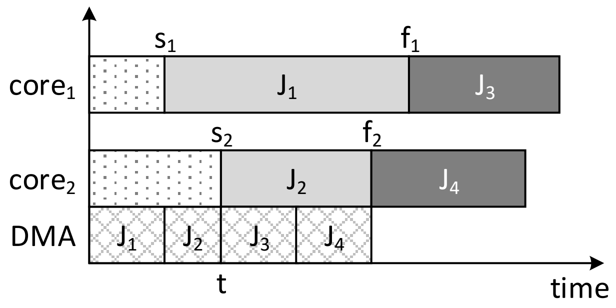


Figure 4.3: The cores are chosen based on minimum s_k.

phases with a few induced schedule holes. This hiding of memory phases gives gDMA a better schedulability over gPREM and gCONT.

In what follows, we discuss how gDMA chooses cores to schedule tasks. In Figure 4.3, we show a schedule of 4 jobs and 2 cores. The time pointers s_1 and s_2 indicate the start time of last scheduled jobs on $core_1$ and $core_2$, respectively. Similarly, the time pointers f_1 and f_2 indicate the finish time of last scheduled job on each core. Now, consider the time t at which each core has a free partition. gDMA chooses $core_1$ to schedule J_3 rather than $core_2$ because $core_1$ has earlier start time of last scheduled job i.e., $s_1 < s_2$. We design gDMA to choose cores based on start time of their last scheduled jobs rather than the finish time to bound the amount of holes between computation phases as we will discuss in Section 4.3.2.

4.2 Scheduler Design of gDMA

gDMA maintains a global queue Q_r in which ready tasks are ordered according to fixed priorities. Whenever a task is released, it is *inserted* in this global queue. The dispatcher *extracts* from the top of the queue the highest priority task and executes it on the DMA, given that the DMA is idle and there is at least one available partition; otherwise, the job remains inside the global queue. Furthermore, the scheduler is implemented as an ISR triggered by certain events. In gDMA, we have the following three events: (1) task release, (2) memory phase completion and (3) computation phase completion. DMA-Dispatcher procedure below is triggered at time t, corresponding to one of these three events, to schedule a new task on the DMA. In addition, after events (2) and (3), if a new task has already been loaded, the core will update s_j to the current time, do a context switch and then execute this task.

For example, consider Figure 4.2 again. At time 3, the completion time of the memory phase of J_1, $core_1$ will update $s_1 = 3$ then do a context switch to execute J_1 and at the same time J_2 will be scheduled to execute on the DMA. At time 15, the completion time of the computation phase of J_1, $core_1$ will update $s_1 = 15$ then do a context switch to execute J_3 since it has already been loaded in $\$_{2,1}$ and at the same time J_5 will be scheduled to execute on the DMA because the completion of J_1 computation phase indicates a free partition.

```
DMA-DISPATCHER
1  $i$ = SELECT-TASK($Q_r$)
2  $(l, j)$ = SELECT-CORE($\{s_k\}$)
```

For simplicity, we assume at time t when DMA-DISPATCHER procedure is invoked that (1) the DMA is idle, (2) at least one partition is available and (3) there is at least one ready task. Otherwise, the procedure will exit, as we assume a non-preemptive execution, and will be triggered again by a later event. Basically, we have: $\{s_k\}$, a set of m time pointers to indicate the start time of the last scheduled job on each core. SELECT-TASK procedure in Line 1 returns the index (i) of the highest priority task out of Q_r. Similarly, SELECT-CORE procedure in Line 2 returns the index (l) of the core that has the minimum s_j, with ties broken arbitrarily, among all cores with free partition, and the index (j) of the last scheduled job on the selected partition. We need this j to schedule the unload phase of previous job with the load phase of current job. For scheduling the first job in the system, we propose to initialize the previous job pointer on each core to a dummy task that simply invokes the dispatcher again. This is more efficient than checking for the special case each time the dispatcher is invoked. Since ties are broken arbitrarily by SELECT-CORE, this procedure works for scheduling the first job if we set $s_j = 0$ for all cores.

4.3 Schedulability Analysis

Since each job in gDMA executes on both the DMA and a processor core, one has to consider both in computing the interference for a given job. However, our analysis works constructively by considering only the computation phases on processor cores. We use

$W_k(J_i)$ to denote the workload of individual job J_i in the problem window of J_k including both the computation time and the induced schedule hole as shown in Figure 4.2. We use $W_k(\Gamma)$ to denote the workload of all tasks inside the problem window of J_k. Furthermore, the interference is denoted by $I_k(\Gamma)$ and is computed based on $W_k(\Gamma)$. Since we only consider the computation phases, we define the time window $[r_k, t_l]$ where $t_l = d_k - e_k^x$ as the new problem window, and we use $L_k = t_l - r_k + 1$ to denote its length. If the problem job starts its computation phase by time t_l, then it completes before its deadline as we assume non-preemptive computation phases.

In bounding the interference, there are two main differences between gDMA and traditional global scheduling theory: (1) long memory phases can induce a " schedule hole" between successive computation phases on the same core. Hence, the interference must include such schedule holes. (2) In traditional global scheduling system, a problem job J_k would start on the earliest processor that becomes idle. Hence, the worst case interference for J_k can be bounded by $W_k(\Gamma)/m$. Since in gDMA we have to bind a job to a core partition once we start its memory phase, such bound does not hold anymore.

The rest of this section proceeds as follows. In Section 4.3.1, we show how to bound the interfering jobs inside the problem window of J_k, for a given task system Γ. The bound on the workload of individual jobs $W_k(J_i)$ is discussed in Section 4.3.2. We then show in Section 4.3.3 how to derive a global bound on $W_k(\Gamma)$, the workload of all tasks. In Section 4.3.4, we compute the bound on the interference based on $W_k(\Gamma)$. Finally, we present the schedulability condition for gDMA in Section 4.3.5.

4.3.1 Bounding the Interfering Jobs

In this section, we first bound the interfering jobs inside the problem window including carry-in, body and carry-out jobs. Then, we populate out of these jobs three sets: X, LD and UD, the set of computation, load and unload phases, respectively. We will use these sets in Section 4.3.3 to derive a global bound on $W_k(\Gamma)$.

A job J_i can interfere inside the problem window of J_k in two different ways: (1) interference caused by non-preemptive scheduling, and (2) interference caused by priority order. Tasks in $lep(k)$ can only have carry-in interference because they cannot start executing after the beginning of the problem window. In contrast, tasks in $hp(k)$ can be activated inside the problem window. Thus, they can have carry-in, body and carry-out jobs. However, assuming all tasks in $hp(k)$ can have carry-in jobs is quite pessimistic. Hence, we start this section by redefining the problem window in order to bound the number of carry-in jobs from $hp(k)$. Then, we show how to bound the interfering jobs that can execute inside the problem window.

Carry-in Limit

As in Chapter 3, we propose to extend the problem window such that it has an earlier starting point t_o and has the same end point t_l. The definition of t_o is the same as Definition 4. In addition, we have the following definition of pending job.

Definition 8. We say that a job is pending if it has been released but it has not started its DMA yet.

The following lemma bounds the number of tasks that can have carry-in inside the problem window that starts at t_o.

Lemma 10. *There are at most $2m$ tasks from $hp(k) \cup lep(k)$ that can have carry-in jobs.*

Proof. We use $t_o - 1$ to denote the time instant before t_o. The complement of Definition 4 states that at $t_o - 1$ there is no pending task from $hp(k)$; otherwise, we could have extended the window. Based on this and the fact that J_k is released at r_k, t_o always corresponds to a release of a task in $hp(k)$. Since we have $2m$ partitions, two for each core, only $2m$ tasks from $hp(k)$ could have been released at or before t_o without being pending or having completed. It follows that only these tasks can have carry-in jobs. Similarly, no task in $lep(k)$ can start on the DMA at or after t_o because there is always a pending job from $hp(k)$ in $[t_o, t_l]$. Thus, only $2m$ tasks executing at least one time unit before t_o can have carry-in jobs. Since it is not possible to load more than $2m$ partitions at one time, the lemma follows. $\square$

Since we assume tasks have constrained deadlines, i.e., no two releases of one task can be active at the same time, these $2m$ jobs must be from different tasks. In what follows we distinguish two types of carry-in: memory and computation in which computation carry-in means a job that has only computation phase, and memory carry-in means a job that has both memory and computation phases.

Lemma 11. *Only one task from $hp(k) \cup lep(k)$ can have memory carry-in.*

Proof. Tasks in $lep(k)$ cannot start on the DMA at or after t_o; hence, they can only have one memory carry-in. On the other hand, tasks in $hp(k)$ that have not started its DMA at

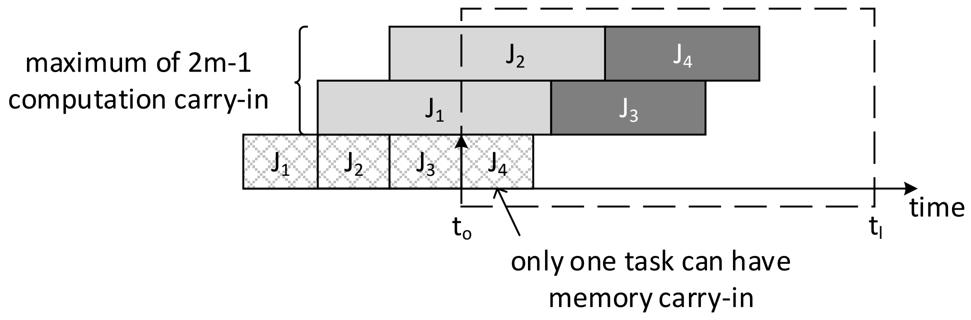


Figure 4.4: Carry-in limit.

or before t_o cannot have memory carry-in because they would be pending and the window would be extended. Since we have a single DMA, only one memory phase can be active at any time either from $hp(k)$ or $lep(k)$. □

We illustrate in Figure 4.4 the worst case carry-in situation for gDMA in which t_o aligns with a beginning of a memory phase and $2m-1$ partitions have been loaded beforehand. We note that a memory phase of a task in $lep(k)$ has to start one time unit before t_o in order to have a memory carry-in. Since memory carry-in is always greater than computation carry-in, we know based on Lemma 10 and Lemma 11 that the worst case situation is to have $2m-1$ tasks with computation carry-in and one task with memory carry-in.

Bounding Jobs for $hp(k)$

In Figure 4.5, we show the worst case activation for a task in $hp(k)$ with three different scenarios. We note that since unload phases are determined at run-time by the scheduler, we only show load and computation phases when bounding the interfering jobs of each task. After each computation phase, the modified data has to be written-back to main memory. Thus, each computation phase introduces an unload phase to the workload of the problem window. We construct memory phases out of these load and unload phases in Section 4.3.3. In addition, the unload phases of the first $2m$ jobs to be executed inside the problem window are carried-in from outside the window. Thus, we need to include the largest $2m$ unload phases of tasks in $hp(k) \cup lep(k)$ to account for these unload phases.

For the case where there is no carry-in, the worst activation is to release the task at t_o. On the other hand, for the case where there is a carry-in job, the worst activation is to execute the carry-in job as late as possible such that its computation phase aligns with t_o

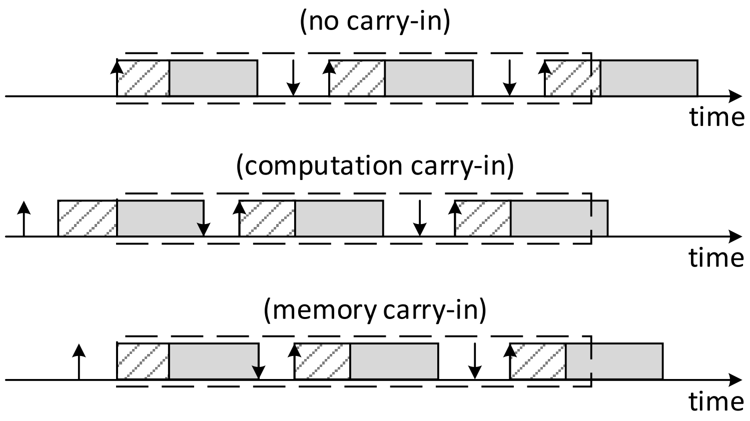


Figure 4.5: The interfering jobs of tasks $\in hp(k)$ with no carry-in, computation carry-in and memory carry-in.

for tasks with computation carry-in, and the memory phase aligns with t_o for tasks with memory carry-in. We observe that the amount of memory and computation are always greater for a task with memory carry-in than the same task with no carry-in. However, the amount of memory and computation can be less for a task with computation carry-in than the same task with no carry-in. To understand how this could happen, see Figure 4.5. The computation carry-in (the middle one) is obtained by introducing a computation phase of a job executed just before the deadline. This introduced computation pushed out a memory phase from the other end. Even though the computation phase has increased, the memory phase has decreased. Unfortunately, this observation complicates the analysis because it is very hard to determine which case will lead to the worst interference because tasks interact differently as we show in Section 4.3.3. As a result, we will consider the task with no carry-in plus an extra computation phase to account for tasks with computation carry-in since a task with carry-in can have at most one extra job compared to no carry-in case.

We note that the execution of jobs after the end of the problem window has no effect to the analysis; hence, we use min function below to only account for jobs within the window. Now, we bound the interfering jobs for a task with no carry-in inside a problem window of J_k as:

- $\lfloor \frac{L_k}{p_i} \rfloor$ body jobs.

- One carry-out job of size $\min(e_i^v + e_i^x, L_k \bmod p_i)$.

Similarly, we compute the interfering jobs for a task with memory carry-in as:

- $\lfloor \frac{L_k + (D_i - (e_i^v + e_i^x))}{p_i} \rfloor$ body jobs.
- One carry-out job of size $\min(e_i^v + e_i^x, L_k + (D_i - (e_i^v + e_i^x)) \bmod p_i)$.

Even though Figure 4.5 shows load and computation phases only, the unload phases are implicitly induced by computation phases, i.e., we add an unload phase for each computation phase. In summary, we populate X, LD and UD to include computation, load and unloaded phases, respectively, from:

- One task in $hp(k) \cup lep(k)$ with memory carry-in.
- All $hp(k)$ tasks with no carry-in.
- An extra $2m-1$ largest computation phases computed as $\min(L_k, e_i^x)$ for $T_i \in hp(k) \cup lep(k)$.

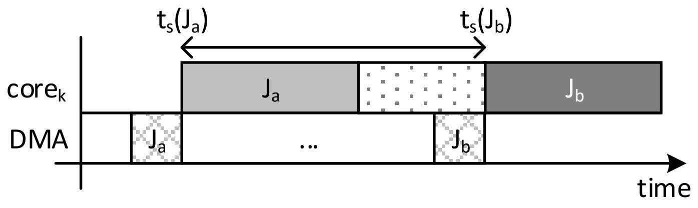
Since it is difficult to choose which task with memory carry-in will lead to the worst case as tasks have different ratios of computation and memory phases, we propose to re-calculate the total interference n times (for each task with memory carry-in) and then take the maximum interference. Finally, we need to include the load phase of the problem job as we only consider the computation phase to define the problem window size.

4.3.2 Bounding the Individual Workload $W_k(J_i)$

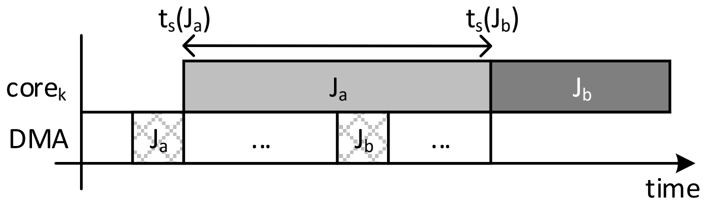
We capture the workload of a job on any core, including both computation and holes, by introducing the concept of a *scheduling interval*.

Definition 9 (Scheduling Interval)**.** Assume J_a is running inside the problem window of J_k and $J_a \rightarrow J_b$. We call $[t_s(J_a), t_s(J_b))$ the *scheduling interval* for J_a, and we let $W_k(J_a)$ to denote its length.

Figure 4.6 shows two examples of scheduling intervals. In Figure 4.6(a), the scheduling interval of J_a contains a schedule hole because the memory phase of J_b is delayed by memory phases from other tasks and therefore $t_s(J_b)$ is also delayed. In contrast, the scheduling



(a) A scheduling interval with schedule holes.



(b) A scheduling interval without schedule holes.

Figure 4.6: Scheduling interval examples.

interval of J_a in Figure 4.6(b) is followed immediately by the scheduling interval of J_b with no schedule holes because the memory phase of J_b has finished (completely overlapped) within the scheduling interval of J_a. As we can see, the hole size for each scheduling interval is variable and is dependent on the execution ordering of other tasks. A key idea behind our analysis is that we ignore the relative ordering of memory and computation phases within the problem window. Instead, we create a bound on the length of each scheduling interval by determining the maximum number of memory phases that can execute within a scheduling interval, a concept we call a *memory sequence*. We will show how to create an ordering of memory and computation phases that maximizes the total length of scheduling intervals in Section 4.3.3.

Definition 10 (Memory Sequence)**.** A *memory sequence* is the consecutive execution of any m memory phases on the DMA. The length ρ of the memory sequence is the sum of the length of the m memory phases.

The following two lemmas will help us in proving Lemma 14 and Theorem 4.

Lemma 12. *At $t_s(J_a)$, there is always a free partition in $proc(J_a)$.*

Proof. Let t_f be the finish time of the memory phase of J_a, and assume J_a is loaded into $\$_{1,k}$ where $k = proc(J_a)$. Since we assume non-preemptive memory phases, we have the following two cases at time t_f. (1) The partition $\$_{2,k}$ is free. In this case, $t_s(J_a) = t_f$ because there is no job executing out of $\$_{2,k}$. (2) $\$_{2,k}$ is already loaded before the memory phase of J_a, i.e., both partitions are full at t_f. As we assume a non-preemptive computation phases, the computation phase of $\$_{2,k}$ should end at $t_s(J_a)$. The end of a computation phase indicates a free partition which concludes the proof. □

Lemma 13. *Assume $J_a \to J_b \to J_c$. Let t_f be the end time of the memory phase of J_b and t_s be the start time of the memory phase of J_c. Then, the computation phase of J_b must start in the interval $[t_f, t_s]$.*

Proof. We consider two cases. (1) If the computation phase of J_a finishes by t_f, then J_b starts immediately at t_f because its memory phase is already loaded. (2) If J_a is still running at t_f, it follows that both partitions must be full at t_f as J_b is also loaded. Since the memory phase of J_c starts at t_s, at least one partition must be freed by t_s. Hence, the computation phase of J_a must finish by t_s and J_b immediately starts because its memory phase is already loaded at t_f. In either case, J_b starts computing in $[t_f, t_s]$, proving the lemma. □

The following Lemma explains why gDMA is designed to select a core with earlier start time rather than finish time as shown in Figure 4.3. It basically gives the guarantee in which our analysis relies on.

Lemma 14. *gDMA will never schedule more than m memory phases inside any scheduling interval with schedule holes.*

Proof. It suffices to prove that no two memory phases targeting a same core can execute inside any scheduling interval with schedule holes. This implies that the maximum number of memory phases inside any scheduling interval with schedule holes is m.

Let $J_a \to J_b$ and assume they run on $core_l$. Consider another core $core_p$ and assume by contradiction that two memory phases targeting $core_p$ are executed inside the scheduling interval $[t_s(J_a), t_s(J_b))$ in which the memory phase of J_b executed last as we assume a scheduling interval with schedule hole. Let t_f and t_s be the finish and the start time of these two memory phases, respectively. By Lemma 13, a computation phase must have started on $core_p$ in the interval $[t_f, t_s]$. The core $core_l$ has a free partition at $t_s(J_a)$ as per Lemma 12, and this partition remains free at t_s because the memory phase of next job J_b is executed last. Since $t_s(J_a) < t_f \leq t_s < t_s(J_b)$, our scheduler at time t_s would target $core_l$ rather than $core_p$. This creates a contradiction, hence the lemma holds. □

The following theorem states the bound of each scheduling interval.

Theorem 4. *The length of $W_k(J_a)$, the scheduling interval of J_a in the problem window of J_k, is upper bounded by the maximum between e_a^x and the length of any memory sequence.*

Proof. Before we start the proof, we note that within the problem window of J_k, there is always at least one pending task. This is by the definition of t_o in the interval $[t_o, r_k)$ and the fact that J_k is pending after r_k. Now, consider two jobs such that $J_a \to J_b$ and both run on $core_l$. Since computation phases are run non-preemptively, the next computation job J_b on the same core cannot start before $t_s(J_a) + e_a^x$. Hence, e_a^x is a bound to the length of the scheduling interval of J_a. Now, let us assume that $t_s(J_b)$ is strictly greater than the finishing time of J_a, i.e., there is a schedule hole between J_a and J_b. Since $t_s(J_b)$ is by definition the earliest time that the next job on $core_l$ can start computing, and $core_l$ is idle immediately before $t_s(J_b)$, it holds that the memory phase of J_b must finish exactly at $t_s(J_b)$. From Lemma 12, we know that $core_l$ must have a free partition starting at $t_s(J_a)$. Hence, it follows that the memory phase of J_b would be started at time $t_s(J_a)$, unless the DMA is continuously busy executing other memory phases. In this case, there must be a continuous memory phases executing on the DMA in the interval $[t_s(J_a), t_s(J_b))$ and we know from Lemma 14 that at most m memory phases, a memory sequence, can execute inside any scheduling interval with holes which conclude the proof. □

As an example, consider again Figure 4.6. It is easy to see that $W_k(J_a)$ in Figure 4.6(b) is equal to the computation phase e_a^x while $W_k(J_a)$ in Figure 4.6(a) is equal to the memory sequence.

4.3.3 Bounding the Total Workload $W_k(\Gamma)$

In the previous section, we showed how to bound $W_k(J_i)$, the workload of an individual job. However, we only characterized the workload as the maximum between a computation phase and a memory sequence as in Theorem 4. It is clear that $W_k(J_i)$ depends on the computation phase of J_i and a possible hole induced by a memory sequence from other jobs. Therefore, $W_k(J_i)$ should be considered globally to determine a safe bound on $W_k(\Gamma)$.

In Section 4.3.1, we populate three sets LD, X and UD from the interfering jobs. However, we assume for now a given set of computation phases and memory sequences. We will show later how to construct from LD and UD a set of memory sequences. We can derive a bound on $W_k(\Gamma)$ as in the following lemma.

Lemma 15. *After sorting computation phases $X = \{\lambda_1, \dots, \lambda_\alpha\}$, where α is the cardinality of X, such that $\lambda_i \le \lambda_{i+1}$, and sorting memory sequences such that $\rho_i \ge \rho_{i+1}$, the following is a valid bound on $W_k(\Gamma)$:*

$$\sum_{i=1}^{\alpha} \max(\lambda_i, \rho_i).$$

Proof. Based on Theorem 4, the length of each scheduling interval is upper bounded by the length of the corresponding computation phase or a memory sequence; hence, we take $\max(\lambda_i, \rho_i)$.

By contradiction, assume that there exists a hypothetical configuration of pairs different than the one in the hypothesis that leads to a strictly higher bound. Since computation and memory sequence lengths are ordered in opposite directions in the hypothesis, it follows that in the hypothetical configuration there must instead exist two pairs (λ_s, ρ_s) and (λ_l, ρ_l), such that $\lambda_s < \lambda_l$ and $\rho_s < \rho_l$ (i.e., ordered in the same direction). We now show that "swapping" λ_s with λ_l leads to a new configuration with a $W_k(\Gamma)$ bound no less than the previous one. Since we can always obtain the configuration in the hypothesis with a finite number of such "swaps" (for example, using bubble sort), this creates a contradiction.

We let $a = \max(\lambda_s, \rho_s)$ and $b = \max(\lambda_l, \rho_l)$, and the bound is $a + b$. We have four cases after swapping λ_s with λ_l based on the two terms a and b and given that $\lambda_s < \lambda_l$: (1) both remain the same. (2) a increases and b remains the same. (3) a remains the same and b decreases. (4) a increases and b decreases. Clearly, (1) and (2) will not decrease the bound. Thus, we only need to consider (3) and (4).

Case(3): To satisfy this case, we should have $\rho_s > \lambda_l$ (a remains the same) and $\rho_l < \lambda_l$ (b decreases), but we obtain $\rho_s > \rho_l$ which is a contradiction.

Case(4): This case is more elaborate than the previous one. In order to satisfy this case, we should have $\lambda_l > \rho_s$ (a increases) and $\lambda_l > \rho_l$ (b decreases). Now, we consider two sub-cases. (i) $\lambda_s \ge \rho_s$: based on these assumptions, the bound before the exchange is $\lambda_s + \lambda_l$ and after the exchange is $\lambda_l + \max(\lambda_s, \rho_l)$ which is larger or equal. (ii) $\lambda_s < \rho_s$: the bound before the exchange is $\rho_s + \lambda_l$ and after the exchange is $\lambda_l + \max(\lambda_s, \rho_l) = \lambda_l + \rho_l$, which is larger. After we examine all possible cases, we can conclude that our initial argument is true. □

Lemma 15 assumes a given set of memory sequences. However, we only have from interfering jobs a set of load LD and unload UD phases. As per Definition 10, a memory sequence contains m memory phases. Thus, we first discuss how to construct memory phases out of LD and UD. Then, we show how to construct memory sequences out of these memory phases.

Since the scheduler dynamically combines unload and load phases into one memory phase, the merged unload phase is generally unknown. We could safely assume for each job that its load phase is merged with the longest unload phase in the system, but this is highly pessimistic. Therefore, we instead propose to construct memory phases as in the following lemma.

Lemma 16. *The set of memory phases $\{\mu_1, \mu_2, \ldots\}$ can be constructed by combining the longest load phases from LD with longest unload phases from UD to increase schedule holes as much as possible.*

Proof. Based on the ordering used in Lemma 15, there must exist a worst case configuration for some value $1 \leq j \leq \alpha$, where in the first j pairs the length of memory sequences is larger or equal, and in the remaining $\alpha - j$ pairs the length of the computation phases is larger or equal. Hence, the bound on $W_k(\Gamma)$ can be obtained by summing the largest j memory sequences and the largest $\alpha - j$ computation phases. By adding largest unload phases to largest load phases, we maximize the first j memory sequences. As a consequence, $W_k(\Gamma)$ is also maximized. □

The longest memory sequence is clearly $\rho_{max} = \sum_{i=1}^{m} \mu_i$ as each memory sequence contains m memory phases. We could safely assume that all computation phases are overlapped with the longest memory sequence. However, the following lemma adds a constraint that improves the bound on $W_k(\Gamma)$.

Lemma 17. *Let $J_a \rightarrow J_b$ and they both run on $core_l$. In addition, assume $W_k(J_a)$ and $W_k(J_b)$ are two scheduling intervals with holes. Then, any memory phase executing in the problem window can contribute to either $W_k(J_a)$ or $W_k(J_b)$.*

Proof. Since the scheduling interval $[t_s(J_a), t_s(J_b))$ of J_a contains holes, i.e., it is bounded by a memory sequence, a memory phase targeting core $core_l$ must finish exactly at $t_s(J_b)$. Hence, such memory phase and all previous memory phases can contribute only to $W_k(J_a)$. On the other hand, all following memory phases can contribute only to $W_k(J_b)$. □

Based on Lemma 17, each memory phase can contribute to at most one memory sequence on each core, i.e., m memory sequences in total. Therefore, we propose to construct memory sequences ρ_i out of memory phases μ_i as follows: $\rho_i = m \times \mu_i$. This guarantees that each memory phase appears at most m times in $W_k(\Gamma)$. Furthermore, following the

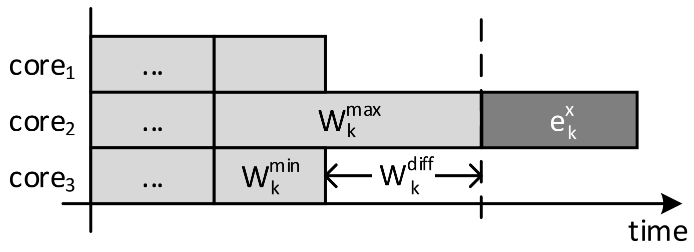


Figure 4.7: The computation phase of the problem job executes after W_k^{max}.

proof of Lemma 16, by combining the largest memory sequences together, we maximize the sum of the first j memory sequences as computed in Lemma 15; hence, the bound on $W_k(\Gamma)$ is also maximized.

4.3.4 Bounding the Interference on a Problem Job

In traditional global scheduling, the interference on the problem job J_k can be bounded by $W_k(\Gamma)/m$ assuming the scheduler is work-conserving. In gDMA, however, J_k can be scheduled on a core with later time even though there is an earlier time available on another core, see how J_3 is scheduled in Figure 4.3. The following lemma further characterize the interference on J_k.

Lemma 18. *Let $J_a \to J_k$ where J_k is the problem job. In addition, let J_b be the last scheduled job on any core such that $proc(J_b) \neq proc(J_k)$. Then, it must hold that $t_s(J_a) \leq t_s(J_b)$.*

Proof. Let t be the time at which DMA-Dispatcher is invoked to schedule J_k. In order for J_k to be scheduled on $proc(J_a)$, $s_{proc(J_a)}$ has to be at time t the minimum (or at least equal since we assume that ties are broken arbitrarily) among all cores that have a free partition according to our scheduler rules. In other words, all other cores should have scheduled jobs with start times greater than or equal to $s_{proc(J_a)}$ before J_k can be scheduled on $proc(J_a)$. □

We let $W_k^{max}(W_k^{min})$ be an upper bound on the maximum (respectively, lower bound on the minimum) length of any scheduling intervals in the problem window of J_k, and

$W_k^{diff} = W_k^{max} - W_k^{min}$. In Figure 4.7, we show the worst case pattern in which each core runs a sequence of scheduling intervals, and J_k is scheduled after the maximum scheduling interval. Based on Lemma 18, the maximum scheduling interval can finish at most W_k^{diff} time units after the earliest finishing time of last scheduling interval on any other core. The sum of all scheduling intervals on all cores is upper bounded by $W_k(\Gamma)$, and a bound on the earliest finishing time of other cores can be derived as:

$$\frac{W_k(\Gamma) - W_k^{diff}}{m}. \tag{4.1}$$

By adding W_k^{diff} as in Figure 4.7 and rephrasing the terms, we obtain the upper bound on $I_k(\Gamma)$, the interference on J_k, as:

$$\frac{W_k(\Gamma)}{m} + (\frac{m-1}{m})W_k^{diff}. \tag{4.2}$$

We compute W_k^{max} as $\max(\rho_{max}, \lambda_\alpha)$ and W_k^{min} as λ_1. Since tasks execute non-preemptively on each core, the length of each scheduling interval must be at least equal to the length of a computation phase, and taking the shortest computation phase within the problem window constitutes a safe lower bound on W_k^{min}. However, decreasing W_k^{min} could render a task unschedulable. Hence, we can assume either tasks idle until their worst-case execution time or we consider a lower bound of 0 on W_k^{min}. In the evaluation, we assumed the latter case. In addition, by assuming the latter case, our schedulability analysis is *sustainable* in the sense that if tasks execute for less than their WCET our bound is still safe. To explain this, since we take the maximum between DMA phases and computation phases when computing the bound, if tasks execute for less, this bound of interference is still an upper bound.

4.3.5 Schedulability Condition

Since the bound on $W_k(\Gamma)$ requires the knowledge of all interfering jobs, it follows that the bound on interference of the problem job derived in Equation 4.2 depends on the size of the problem window. We note that even though the extended problem window that starts at t_o has the advantage of limiting the amount of carry-in, finding t_o is of pseudo-polynomial time complexity, given that the total utilization is strictly less than m [17]. However, the authors of [64] observed that choosing a window of length L_k and starting at t_o is sufficient for the schedulability analysis. Since $t_o \leq r_k$, the length of $[t_o, t_l] \geq [r_k, t_l]$. Intuitively, computing bounds on the amount of work inside a small time interval is tighter than a large interval as the amount of work gets amortized over larger intervals [17]. Based on such intuition, we prove our schedulability condition as follows.

Lemma 19. *If J_k misses its deadline, then $I_k(\Gamma) \geq L_k$.*

Proof. Assume J_k misses its deadline and $I_k(\Gamma) < L_k$. Since we assume a work-conserving scheduler, the time interval $[r_k, t_l]$ has to be busy in order for J_k to miss its deadline. Otherwise, J_k could have executed and finished before the deadline. In addition, $[t_o, r_k)$ has to be busy as per Definition 4. As a result, the time interval $[t_o, t_l]$ has to be busy for J_k to miss its deadline. Equation 4.2 gives a bound on $I_k(\Gamma)$, the interval of time where all cores are busy including both computation and holes. Since $I_k(\Gamma) < L_k$, there must exist a time in $[t_o, t_l]$ which is not busy. This creates a contradiction; hence, the lemma follows. □

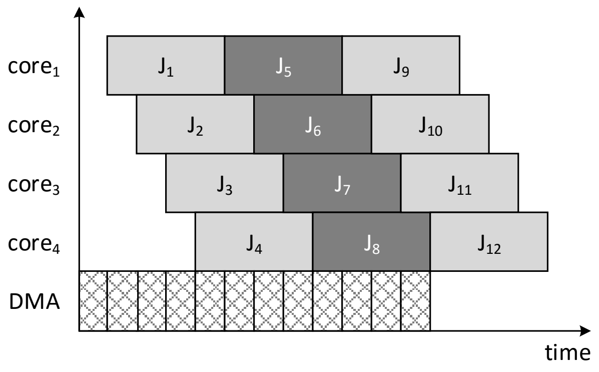
Theorem 5. *If $\forall T_k \in \Gamma : I_k(\Gamma) < L_k$ then the system is schedulable.*

Proof. It follows from the contrapositive of Lemma 19 that J_k will meet its deadline if $I_k(\Gamma) < L_k$. If this hold for all tasks in Γ, the system is schedulable. □

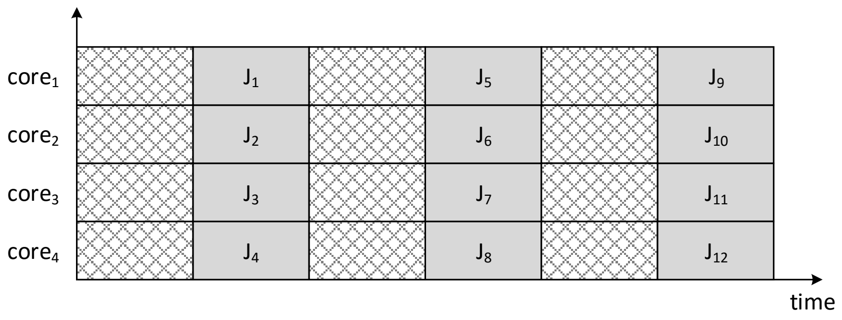
Furthermore, the computational complexity of our analysis is mainly dominated by computing α max operations as in Lemma 15. Since the number α depends on the number of tasks $T_i \in hp(k)$ as well as the period ratios p_i/p_k, the computational complexity is dependent on the number of jobs in a window of size L_k rather than the number of tasks. The analysis in general is meant to be run off-line; hence, we deem such complexity acceptable.

4.4 Evaluation

Before we show the results, we intuitively discuss how gDMA is able to utilize processor cores better than gCONT. Assume a system loaded with 12 tasks. Each task takes 4 time units to compute on a processor core and one time unit for DMA to unload/load into a local partition. gDMA is able to utilize the processor cores 100% by overlapping tasks' execution over the cores as long as the DMA operations are small compared to cores execution as shown in Figure 4.8(a). On the other hand, processor utilization in gCONT is affected by memory stall times. In the worst case, all cores can access main memory at the same time. In this case, each core takes $1 \times m$ time units to access main memory instead

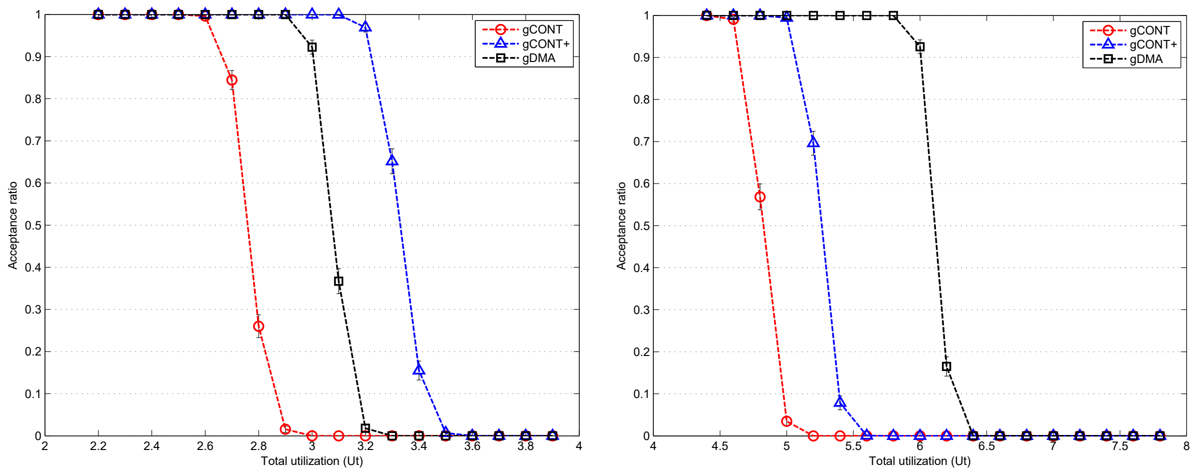


(a) gDMA schedule showing 100% utilization.



(b) gCONT schedule showing 50% utilization.

Figure 4.8: A comparison between gDMA and gCONT.



(a) $m = 4$ (b) $m = 8$

Figure 4.9: The schedulability of gDMA, gCONT and gCONT+.

of one time unit. In particular, with $m = 4$, each task takes 4 time units for memory and 4 time units for computation. As shown in Figure 4.8(b), the processor utilization is

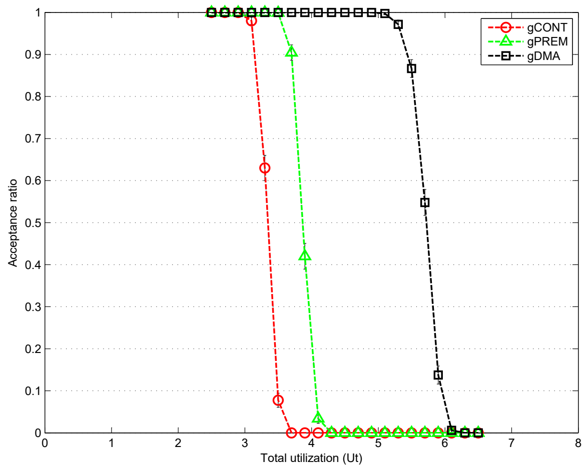
$$\frac{12 \times 4}{(12 \times 4) + (12 \times 4)} = 50\%$$

Based on this intuition, gDMA should perform significantly better than gCONT. However, gDMA schedule exhibits more priority inversion than gCONT, which can worsen the WCRT of high priority tasks. In addition, the computation times of real tasks are not always large enough to completely hide the memory time of DMA operations.

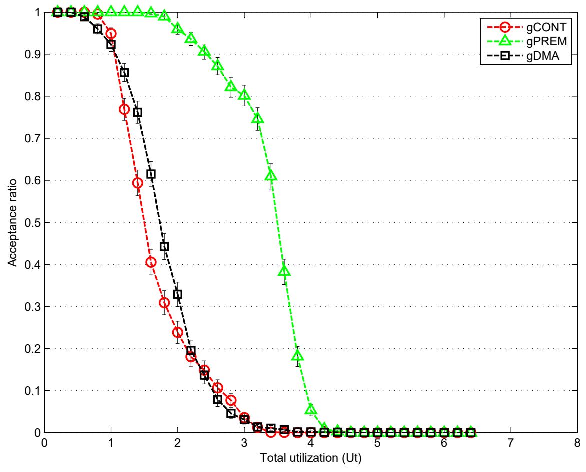
4.4.1 Results

We use the same experimental setup as in Chapter 3. Since gDMA utilizes a DMA component which is an extra hardware component, we compare against gCONT and gCONT with one extra core denoted as gCONT+. In this case, we favor gCONT+ because a processor core is more complex than a DMA component.

Figure 4.9 shows the results in terms of the percentage of schedulable task sets (acceptance ratio). We set the periods to be within $[1000, 5000] \times 1000$ clock cycles. Figure 4.9(a) shows that gDMA has schedulability that is better than gCONT and worse than gCONT+.



(a) Large periods.



(b) Small periods.

Figure 4.10: The schedulability of gDMA, gPREM, gCONT.

The reason gCONT+ outperforms gDMA is that the utilization of the generated task sets is based on m-core processor and then scheduled on $m + 1$ cores. That is, $4/5 = 80\%$ for $m = 4$. As a result, the extra core is used to execute more tasks. An important point to note is that by increasing Ut, the number of tasks increases and consequently memory utilization. Thus, gDMA outperforms gCONT+ in Figure 4.9(b) with $m = 8$. The overlapping mechanism hides the memory time and achieves better schedulability. In Figure 4.9(a), the memory utilization is low compared to computation. Thus, the extra processor core is used to execute computation phases. In contrast, the DMA of gDMA can only execute memory phases.

In Figure 4.10, we include gPREM in the comparison. Since gDMA handles memory more efficiently than gPREM, gDMA withholds schedulability longer before drops at around 6 in Figure 4.10(a). However, we recall that gDMA has larger carry-in workload than gPREM and gCONT. That is, $2m$ jobs can be loaded before the task under analysis can execute. To evaluate this case, we decrease the task periods to be within the range $[200, 1000] \times 1000$ clock cycles, i.e., the slack time is reduced or alternatively, the utilization is increased. As shown in Figure 4.10(b), the large amount of carry-in workload that can execute within the problem window of high priority tasks affects the schedulability of gDMA. We note that high priority tasks tend to have small problem window since we assign priorities based on rate monotonic. Based on this result, we cannot claim that gDMA dominates gCONT or gPREM. However, gDMA can benefit systems with high memory utilization and high priority tasks that can tolerate the blocking time of low priority tasks as in Figure 4.10(a).

4.5 Summary

gDMA is a new approach to globally schedule 3-phase tasks on a system with multicore processor and one DMA component. In this approach, the local memory of each core is divided into two equally sized partitions to ensure that while the core is busy executing one task out of one partition, the DMA component can load another task in the other partition. As a result, the latency for loading and unloading tasks from main memory can be fully or partially hidden within the computation time of other tasks. Hence, this approach provides two advantages for real-time systems. First, a predictable execution time because tasks execute out of local memories without memory stalls, and the *contention* for access to main memory is eliminated by scheduling the DMA operations. Second, an increased system performance by hiding the main memory latency. In addition, we introduced in this chapter a novel schedulability analysis to globally co-schedule processor cores as well as the DMA. In particular, we abstract the execution times of tasks on both cores and the DMA as *scheduling intervals*.

Chapter 5

Trading Cores for Memory Bandwidth

So far, we discussed multiprocessor scheduling in which each real-time application is modeled as sequential periodic or sporadic task and increasing the number of cores allows the system to execute larger number of tasks while each task cannot run at any speed faster than the speed of single-core processor. However, as technology evolves, there are always applications that demand more processing power than a single processor can provide.

In this chapter, we consider the scheduling of parallel real-time tasks. The goal is to meet the timing constraints of computation-heavy real-time applications with utilization greater than one such as synthetic vision and object tracking [79] which cannot be feasibly executed on single core. At the same time, many of such applications are easily parallelizable. In this case, the application has to be divided into parts to allow for the distribution of its load on multiple cores. In general, these parts depend on each other and this imposes a partial order in their execution. DAG and fork-join task models have been proposed to capture such parallelism for real-time tasks [20, 21, 80]. We can think of a parallel task as a collection of sequential *subtasks* with order relation in their execution.

Similar to sequential tasks, two notable scheduling schemes have been proposed for parallel tasks: *global* and *federated*. In global scheduling, all parallel tasks share one global queue in which the ready subtasks are inserted. The scheduler chooses the highest priority subtasks and schedules them on the available cores. In contrast, federated scheduling assigns a dedicated number of cores for each parallel task with utilization greater than or equal one. In this case, each parallel task has its own ready queue, and any greedy work-conserving scheduler can be used to schedule the parallel task on the assigned cores. In general, global scheduling incurs more run-time overhead than federated scheduling. This extra overhead comes from the preemption and migration of subtasks due to their priority. The preemption and migration are expected to increase with global scheduling of parallel tasks because they execute on more than one core. On the other hand, parallel tasks can be executed without preemption and migration with federated scheduling. Even with ignoring the run-time overhead, the capacity augmentation bound for federated scheduling has been shown to dominate that in global scheduling with EDF and RM [84].

Since parallel tasks are assigned dedicated cores in federated scheduling, they receive no interference from other tasks at processor cores level. However, parallel tasks can interfere with each other through the shared main memory. The previous research on parallel task model has not factored in the memory demand of real-time tasks. We argue that the bandwidth of main memory should be considered when scheduling real-time parallel tasks. A modern uniprocessor can easily saturate the bandwidth of a state-of-the-art memory. With chip multiprocessor (CMP) where main memory is shared between multiple cores, the situation is even worse. An important observation is that the number of cores in one chip is expected to continue increasing based on Moore's law (number of transistors doubles every 18-24 months). In contrast, the memory bandwidth is expected to increase but in a slower rate (10-15% per 12 months) [88]. As a result, the memory bandwidth will increasingly become scarcer. This important fact motivates the development of a new method to trade in system's cores for memory bandwidth.

To mediate memory access contention, bandwidth partitioning has been proposed to achieve fairness for real-time systems. An arbiter like RR, for instance, is fair because it automatically divides the bandwidth into *equal-sized* partitions. For a system with m active cores, the latency of each memory request can be delayed, at the worst-case, by $m-1$ other requests. This effectively means that each core receives $1/m$ of the bandwidth. Obviously, uniform distribution of bandwidth is not always the best solution because different tasks have different demands with respect to computation and memory. Different methods have been proposed to partition the memory bandwidth into unequal partitions. However, none of them (as far as we know) have been applied on federated scheduling of parallel tasks.

In this chapter, we propose a novel method to assign cores to tasks by taking into account the memory demand of each parallel task. We start by a naive approach to account for memory demand with RR arbiter. Then, we apply our method on two arbitration schemes: one is software-based and the other one is hardware-based. The results show that our method significantly improves the schedulability of real-time parallel tasks when compared to memory oblivious methods. To further enhance the execution time of parallel tasks, we propose in Section 5.8 static scheduling algorithm and employ the co-scheduling approach on each parallel task.

5.1 System Model

We consider a set of n parallel tasks $\Gamma = \{T_1, \ldots, T_n\}$. These tasks are assumed to be high utilization, i.e., they need at least one core to execute. We assume that low utilization tasks are separately scheduled on a set of cores in away similar to multiprocessor scheduling of sequential tasks. We focus on high utilization tasks as they can be utilized to trade in cores for memory bandwidth. Thus, we use m_i to denote the number of cores (cluster size) assigned to T_i. Similarly, we use $1/q_i$ to denote the bandwidth fraction assigned to T_i. We let $m^{used} = \sum_{i=1}^{n} m_i$ and $BU = \sum_{i=1}^{n} 1/q_i$. We consider a system that contains m cores and one shared memory. The total bandwidth utilization can be at most one and therefore $BU \leq 1$. Obviously, the shared memory in some systems can have multiple ports and connected via complex interconnects such as crossbars. However, we focus on single arbiter that partition the memory bandwidth between cores.

The bound on the *makespan* of T_i is denoted as e_i for a given m_i and q_i, i.e., e_i is a function of these two parameters. Each parallel task T_i is characterized by a 4-tuple of parameters (e_i^m, e_i^x, L_i, D_i) detailed as follows. e_i^m is the total time to access main memory by all subtasks of T_i assuming full memory bandwidth. We assume that e_i^m is constant and upper bounds the memory time on a given range of number of cores. This assumption works when the added data/code due to sharing between cores does not significantly increase the number of memory requests. In fact, a characterization of PARSEC benchmark suite [27] has shown that the number of memory requests for PARSEC workloads remains almost the same with increasing number of cores.

In addition, e_i^m represents the time to serve last-level cache misses of T_i in cache-enabled systems. We note that federated scheduling allows us to apply optimizations to minimize the makespan bound since each task is assigned a dedicated cluster of cores, i.e., it has no interference from other tasks at the core level. For example, a static algorithm such as list-scheduling [19] can be used. With static scheduling algorithm, the order of execution and the mapping of subtasks to cores are known and can be controlled at compile time. In addition, the execution of subtasks can be non-preemptive, i.e., there is no cache related preemption delays (CRPD). These two points have the advantage to greatly simplify the cache analysis. In particular, the problem reduces to single-core non-preemptive cache analysis methods for private caches, and for shared caches, there are generally two ways to bound inter-core interference: spatial isolation and joint analysis. We show in our recent survey paper [62] a large number of cache management techniques to provide *spatial isolation* for shared caches. These techniques can be used to assign each task a private partition in shared cache to avoid the interference effect of other tasks. Within the cache partition of one task, joint analysis methods can be used [147] to estimate the number of cache misses. Similar to systems without caches, e_i^m can be bounded by the maximum number of last-level cache misses that the cache analysis provides on a given range of number of cores.

e_i^x is the total computation time of all subtasks on one core, i.e., the sequential computation time. L_i is the execution time of the task assuming an infinite number of cores, i.e., the computation time of the critical path. We emphasize that both e_i^x and L_i are pure computation with zero memory time. That is, we assume that $e_i^x = e_i - e_i^m$ where e_i represents the worst-case makespan on one core and with full memory bandwidth. This model can be applied on any platform in which memory delay is additive to task's execution time [69] or a fully timing-compositional core [139] in which the task's execution time is the sum of computation time and memory access time. We consider a sporadic task model where the period represents the minimum inter-arrival time between any two instances and a constrained-deadline D_i which can be less than or equal to the period.

Our analysis only uses these values (e_i^m, e_i^x, L_i, D_i) without relying on any dependency structure. That is, the parallel task can unfold in any shape at run-time as long as these upper-bound values are not violated. Similarly, the order and the position of memory requests are unknown in advance. Instead, we assume any arrival pattern as long as their total access time is bounded by e_i^m. That is, these memory requests can be evenly distributed over the execution time of the task or clustred in a short amount of time. This is a big advantage for our method to be applied in practical settings. Moreover, given a DAG structure, e_i^m, e_i^x and L_i can be obtained in polynomial time [40].

The problem we want to address in this chapter can be stated as follows. *Given a number of cores m, a shared memory bandwidth and n high utilization parallel tasks, we seek to assign the m cores and the memory bandwidth to these n parallel tasks such that all of them meet their deadlines.* In other words, we need to output for each parallel task the following.

1. The number of cores m_i.
2. The bandwidth fraction $1/q_i$.

5.2 Federated Scheduling

In this section, we review federated scheduling as found in literature [84, 19, 20]. Since previous work does not consider memory time, we will assume tasks are made of computation e_i^x. We will show how to integrate memory time later in this chapter. Federated scheduling is a generalization of partitioned scheduling to parallel tasks. In this scheduling scheme, tasks in Γ are divided into two sets:

1. high utilization tasks Γ^{high} with $e_i^x \geq D_i$ and
2. low utilization tasks Γ^{low} with $e_i^x < D_i$.

Each task $T_i \in \Gamma^{high}$ is assigned m_i dedicated cores and scheduled using any greedy work-conserving scheduler. On the other hand, Γ^{low} tasks are scheduled on the remaining cores in a way similar to multiprocessor scheduling of sequential tasks. Here, Γ^{low} tasks are treated as sequential tasks even though they have internal parallelism. For DAG task model, an execution ordering following the topological order of dag nodes is a valid sequential execution.

In [83], the authors propose *capacity augmentation bound*, defined below, which extends the notion of utilization bound to parallel tasks.

Definition 11. A scheduling algorithm A provides a capacity augmentation bound of b if algorithm A schedules any task set with the following conditions:

1. $Ux \leq m/b$ and
2. $L_i \leq D_i/b$ for all tasks.

It is shown in [84] that the capacity augmentation bound of federated scheduling is 2, meaning that federated scheduling can accept any task set with total utilization $Ux \leq m/2$ and each $L_i \leq D_i/2$. This bound dominates that for global schemes such as global EDF or global RM. In addition, the computed bound of 2 is proven in [84] to be tight with large value of m by showing that no scheduler can provide a better bound than $2 - 1/m$.

A bound on the makespan of $T_i \in \Gamma^{high}$, when scheduled by any greedy work-conserving scheduler on m_i dedicated cores, can be obtained as follows:

$$e_i = \frac{e_i^x - L_i}{m_i} + L_i. \tag{5.1}$$

The intuition behind this formula is that in the worst-case $m_i - 1$ cores are idle while the critical-path is executed. This leads to the worst possible waste of processor time.

A sufficient schedulability test for each $T_i \in \Gamma^{high}$ can be easily established as $e_i \leq D_i$. Therefore, T_i is guaranteed to meet its deadline if it is assigned m_i cores derived from (5.1) as follows:

$$m_i = \left\lceil \frac{e_i - L_i}{D_i - L_i} \right\rceil . \tag{5.2}$$

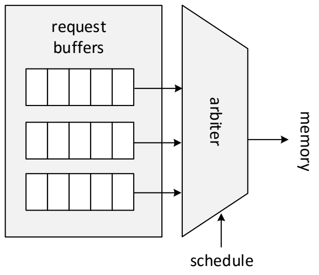


Figure 5.1: Bus arbiter design.

The remaining cores $m - \sum_{T_i \in \Gamma^{high}} m_i$ are assigned to schedule Γ^{low} tasks by any multiprocessor scheduling strategy with utilization bound $\geq 1/2$ such as partitioned EDF or partitioned RM [89, 9].

In contrast to capacity augmentation bound, our objective in this chapter is to provide an assignment algorithm for cores and memory bandwidth. We observe that this assignment algorithm can be used as a schedulability test by checking that the total number of used cores, $m^{used} \leq m$ and bandwidth utilization, $BU \leq 1$.

5.3 Bus Arbiter

In Figure 5.1, we show a diagram of bus arbiter in CMP architecture. The memory requests of different cores are inserted into request buffers. These buffers can be part of system's cores such as load/store queues [73] or part of the network-on-chip interface known as transaction queues [117]. In this latter case, the mapping between cores and request buffers can be changed by using reconfiguration abilities [60]. This allows a cluster of cores to share one request buffer. In addition, existing memory controllers have different buffers for each memory bank [118]. Thus, the requests originated from the same cluster can be inserted into one buffer if we assign each cluster a private bank. As we will see in Section 5.4, the configuration of request buffers whether per core or per cluster makes a significant difference in bounding the memory access time. Memory requests are then issued to main memory following a specific arbitration policy.

RR arbitration policy [107] has been proposed to arbitrate shared resources in real-time systems because it is fair and the access latency can be easily bounded. For instance, each core will receive $1/m$ of total bandwidth assuming each request is preceded by $m-1$ other

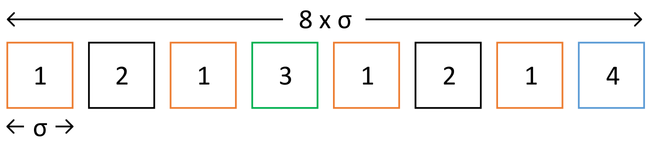


Figure 5.2: 4-core bus schedule and $\{\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8}\}$ bandwidth requirements.

requests. With such isolation, the memory access latency of a task can be obtained without the knowledge of other tasks [106]. While RR policy automatically divides the memory bandwidth into even partitions, the objective of an ideal resource arbiter is to provide a fraction (not necessarily equal) of the bandwidth for each core based on its demand.

Harmonic RR [145] has been proposed to partition the memory bandwidth into unequal sizes. The essential part with this arbitration scheme is to divide the time into slots of size σ, the access time of one memory request. Then, each core is assigned a number of slots based on its demand. Figure 5.2 shows a schedule example of 4 cores with unequal share of bandwidth. Here, each bandwidth requirement can be seen as an integer multiple of σ. For instance, a value of 1/2 means that the core needs one slot in every 2 slots.

5.4 Integrating Memory Demand

In this section, we show how to integrate the memory demand e_i^m into federated scheduling. Since e_i^m is the memory time for $q_i = 1$ (full memory bandwidth), the memory time with $q_i = 1/2$ (half memory bandwidth) is $e_i^m \times 2$. Thus, an upper bound on the makespan of T_i, when scheduled by any greedy work-conserving scheduler on m_i dedicated cores and q_i fraction of memory bandwidth, can be obtained as follows.

$$e_i = \frac{e_i^m}{q_i} + \frac{e_i^x - L_i}{m_i} + L_i \tag{5.3}$$

In this bound, the memory time (e_i^m/q_i) is safely assumed to be serialized without overlapping with computation. It is equivalent to say that the critical path L_i is executed on one core without overlapping with computation on other cores. This bound is valid because the parallel task is assumed to unfold at run-time without knowing the order or even the mapping of tasks to cores and the order or the position of memory requests.

In (5.3), we use $(e_i^x - L_i)/m_i + L_i$ as a coarse-grained bound on the computation makespan of T_i since we assume a greedy work-conserving scheduler similar to (5.1). As mentioned before, a static scheduling algorithm can be used instead. In this case, the bound on the computation makespan can be replaced by $f_i(m_i)$. In Section 5.8, we propose a static scheduling algorithm to execute a parallel task on its own cluster of cores.

Similar to before, a sufficient schedulability test for each $T_i \in \Gamma$ can be established as $e_i \leq D_i$. Thus, the bandwidth fraction q_i can be derived from (5.3) as a function of m_i as follows.

$$q_i(m_i) = \frac{e_i^m \times m_i}{(D_i - L_i) \times m_i - (e_i^x - L_i)} \tag{5.4}$$

For ease of reference, we use the following notation.

$$\Delta_i(m_i) = q_i(m_i) - q_i(m_i + 1). \tag{5.5}$$

Lemma 20. *The function $q_i(m_i)$ has the following two properties: $\Delta_i(m_i) > 0$ and $\Delta_i(m_i) > \Delta_i(m_i + 1)$.*

Proof. This can be proven by direct substitution of (5.4) in (5.5) and applying some algebraic manipulation. Alternatively, we note that the first derivative of q_i is negative and the second derivative is positive, meaning that the function is strictly decreasing and convex (decreasing slope). Since $\Delta_i = -dq_i/dm_i$, the lemma follows. □

The proof is shown for the computation makespan bound we consider in (5.3). As we stated early, this bound can be replaced by $f_i(m_i)$. For this case, the work in [123] has shown that these two properties can be fulfilled in many common cases of parallelization functions.

As per Lemma 20, $q_i(m_i)$ is a decreasing function since $\Delta_i(m_i) > 0$. In other words, giving $T_i \in \Gamma$ more cores reduces its memory bandwidth demand. We can think of $\Delta_i(m_i)$ as the amount of reduction in bandwidth fraction when $T_i \in \Gamma$ is given one more core. We illustrate the concept of trading cores for memory bandwidth by the following example.

Example 3. Consider a task T_1 with the following parameters: $e_1^m = 50$, $e_1^x = 100$, $D_1 = 150$ and for simplicity assume $L_1 = 0$. If we assign T_1 one core and full memory bandwidth, the makespan bound is $e_1 = 50 \times 1 + 100/1 = 150$ with zero slack time. Now, if we give T_1 one extra core, the makespan bound becomes $e_1 = 50 \times 1 + 100/2 = 100$ with 50 slack time. This slack time can be used to relax the memory demand with half memory bandwidth, i.e., $e_1 = 50 \times 2 + 100/2 = 150$.

In this example, we give T_1 one more core and gain half memory bandwidth which can be used by another task to meet its deadline. Our method presented in Section 5.5 finds the optimal choice to which tasks are taking the extra cores and which tasks are receiving the gained bandwidth.

Furthermore, the minimum number of cores to execute $T_i \in \Gamma$ while assuming full memory bandwidth ($q_i = 1$) can be derived from (5.3) as follows.

$$m_i^* = \left\lceil \frac{e_i^x - L_i}{D_i - L_i - e_i^m} \right\rceil \tag{5.6}$$

5.4.1 RR with (m) Request Buffers

In this section, we discuss how to bound the memory time on a system with per core request buffer and RR arbitration policy (mRR). In this configuration and m processor cores, each memory request can be delayed by $m-1$ requests of other cores. In other words, the memory time can be bounded as $e_i^m \times m$. However, since we assume in (5.3) that all memory requests within each cluster are serialized and served from one buffer, we observe the following better bound on the makespan.

$$e_i = e_i^m \times (1 + \sum\nolimits_{j \neq i} m_j) + \frac{e_i^x - L_i}{m_i} + L_i \tag{5.7}$$

In this bound, we only account for $\sum_{j \neq i} m_j$, the delay from buffers outside the cluster of T_i, and 1 for cluster under analysis T_i. That is, there is no interference from buffers within the same cluster as we assume in (5.3) that memory requests are served from one buffer. In mRR-Assign algorithm, we show how to assign cores to tasks. The algorithm starts by giving each task the minimum number of cores to execute assuming full memory bandwidth. Then, the algorithm computes the makespan for each task. Intuitively, if $e_i > D_i$ for any task, the only flexible parameter with mRR is to give more cores. We note that find-one in Line 5 returns the index of any task with $e_i > D_i$. Since e_i depends on m_i of other tasks as in (5.7), we need to update e_i of all tasks after each increment of m_i.

mRR-Assign is an efficient algorithm with computational complexity of $O(mn)$. The **while** loop iterates at most m times as we give one more core in each iteration as in Line 6. In each iteration, we update n tasks as in Line 7.

5.4.2 RR with (n) Request Buffers

We observe that the bound in (5.7) assumes all memory requests are served from one buffer for the cluster under analysis. On the other hand, the bound assumes the interference from

```
mRR-Assign
1  for each task $T_i$
2          $m_i = m_i^*$
3  compute $e_i$ as in (5.7)
4  while $(m^{used} \leq m) \wedge (\exists T_i : e_i > D_i)$
5          $j =$ find-one$(T_i \in \Gamma : e_i > D_i)$
6          $m_j = m_j + 1$
7          update $e_i$ for all tasks
8  return $(m^{used} \leq m)$
```

Table 5.1: Task set example.

	e^m	e^x	L	D
T_1	3552	14412	65	9254
T_2	629	12505	478	4830

all buffers of other clusters. Thus, we propose to assign one buffer for each cluster of cores to improve the upper-bound on memory access time. In this case, each cluster introduces a delay of one regardless of its size. Since the number of clusters $n \leq m^{used}$, the per cluster bound is always better than the per core bound. We refer to RR with per cluster buffers as nRR, and we use it as a baseline to compare with other methods.

Unlike mRR, the bandwidth fraction in nRR is constant, $q_i = 1/n$, and therefore m_i can be computed using the following closed form formula.

$$m_i^n = \left\lceil \frac{e_i^x - L_i}{D_i - L_i - n \times e_i^m} \right\rceil \tag{5.8}$$

5.5 Trading Cores for Memory Bandwidth

The problem with nRR is that it assigns equal fractions of bandwidth to all tasks. However, different tasks have different demands for memory and computation. We illustrate this with the following example.

Example 4. Assume a system with two high utilization tasks $n = 2$, eight cores $m = 8$ and task parameters as in Table 5.1. This task set is not schedulable by nRR in which each task

```
Optimal-Assign
1 for each task $T_i$
2     $m_i = m_i^*$
3 compute $q_i(m_i)$ as in (5.4)
4 while $(m^{used} \leq m) \wedge (BU > 1)$
5     compute $\Delta_i(m_i)$ as in (5.5)
6     $j$ = find-one($T_i \in \Gamma : \Delta_i(m_i)$ is maximum)
7     $m_j = m_j + 1$
8     update $q_j(m_j)$
9 return $(m^{used} \leq m)$
```

takes $q_i = 1/2$ even with $m = \infty$. This can be verified using Equation 5.3. However, the task set is schedulable if we assign $q_1 = 0.625$, $q_2 = 0.375$, and $m_1 = m_2 = 4$. We note that the same task set is not schedulable on a system with $m = 7$ by any bandwidth assignment. This can be proven by running this task set on our optimal algorithm described next.

The example shows that each task has a particular demand for memory and computation. In order to make the task schedulable, both resources (processor and memory) have to be adjusted to meet its demand. Thus, we propose Optimal-Assign algorithm to assign cores (m_i) and bandwidth fraction (q_i) for each $T_i \in \Gamma$. First, the algorithm assigns the minimum number of cores that guarantees that the computation demand will meet the deadline. Then, it computes the required bandwidth fraction $q_i \in \mathbb{R}$ as a function of m_i as in (5.4).

The objective of Optimal-Assign algorithm is to reduce the total bandwidth utilization BU to a value equal or below one. The intuition behind this algorithm is to give more cores to tasks that give the maximum bandwidth reduction. This metric is captured by finding the maximum $\Delta_i(m_i)$ as in Line 6. We note that find-one returns the index of task with largest value of $\Delta_i(m_i)$.

Optimal-Assign is efficient with computational complexity of $O(nm)$. The **while** loop iterates m times as we give one core in every iteration as in Line 7. In each iteration, we find the maximum value in a list of n tasks as in Line 5. Furthermore, we argue that Optimal-Assign algorithm is optimal in the following sense.

Lemma 21. *For a given number of cores,* Optimal-Assign *gives the smallest value of BU or, alternatively, largest amount of bandwidth reduction.*

Proof. This claim can be proven by induction. We use $P(x)$ to express the claim as a function of the number of cores. The base case $P(1)$ is easy to prove because OPTIMAL-ASSIGN chooses the maximum reduction in each step. Assume $P(x)$ is true in which $\sum_{i=1}^{n} m_i = x$. The total bandwidth reduction after the assignment of x cores is

$$\sum_{i=1}^{n} \sum_{y=m_i^*}^{m_i-1} \Delta_i(y).$$

Since OPTIMAL-ASSIGN chooses the maximum $\Delta_i(y)$ in each step, and for any task $\Delta_i(y) > \Delta_i(y+1)$ as per Lemma 20, there is no $\Delta_i(y)$ for $y > m_i - 1$ that can replace $\Delta_i(y)$ in above formula and leads to a larger bandwidth reduction. Now, assume that $\Delta_i(m_i) \geq \Delta_j(m_j) \forall j \neq i$. OPTIMAL-ASSIGN will choose $\Delta_i(m_i)$ and this ensure that $P(x+1)$ is true. □

5.6 OPTIMAL-ASSIGN Implementation

OPTIMAL-ASSIGN algorithm was proposed to mitigate the rigidity of nRR. However, $q_i \in \mathbb{R}$ was assumed in this algorithm, i.e., the memory bandwidth can be partitioned into any real number which is not always practical. In this section, we apply OPTIMAL-ASSIGN on two practical arbitration schemes, and show how to bound the memory time of each task.

5.6.1 Software Regulated Memory

In this section, we describe a software based arbitration called Memguard [146]. We then propose an analysis to bound the memory time of each task. The analysis works by converting the problem into two modes of operation, and then applying OPTIMAL-ASSIGN algorithm on each mode.

Memguard is a technique designed to partition the memory bandwidth from software level. It uses per-core regulators to monitor and enforce the memory bandwidth allocation. In particular, each regulator monitors the number of memory requests performed by each core in a given period P. This can be achieved through HPC. These counters are configured to trigger an event when any core exceeds its allocated bandwidth (budget). Upon receiving such event, the corresponding core is suspended. The core recharges its budget and resumes its activity at the next period. The main idea is that within a period of time P, each core can issue a number of memory requests equivalent to the assigned bandwidth Q_i. P is a system parameter decided at design time and its value is set based on the overhead imposed to synchronize all cores.

Memguard was proposed to regulate the memory of individual cores. In this section, we extend the idea of Memguard to federated scheduling with cluster of cores. In a platform such as P4080 [58], a complex chain of conditions can be established to trigger one event. In our case with a cluster of cores executing one parallel task, we can configure the HPC to trigger an event when the aggregated number of memory requests from a cluster of cores exceeds the assigned bandwidth. Memguard leaves the fine grained arbitration at the level of one memory request to the hardware. We consider RR as the arbitration policy between clusters and each cluster uses one request buffer similar to nRR (Section 5.4.2).

Like Memguard with individual cores, we assume that the clusters are synchronized with one period P and $\sum_{i=1}^{n} Q_i \leq P$. In addition, both P and Q_i represent time and their ratio Q_i/P is unit-less. We can think of Q_i/P as the fraction of time that a task is allowed to access memory. To ensure full memory budget at each task release, we assume the task's period to be integer multiple of P.

Memguard for partitioned multicore processor has been analyzed in [142]. This work considers sequential tasks and assumes the budget for each core is given. The main focus was to compute the response time of real-time tasks on regulated cores. In contrast, our analysis considers parallel tasks with federated scheduling. In particular, we seek to determine Q_i and compute the memory access time for each task under Memguard regulation.

We recall that the memory part of each task can be bounded as $e_i^m \times n$ when we rely on nRR hardware arbiter only. However, the memory time can be modified due to the effects of Memguard regulation. Since $\sum_{i=1}^{n} Q_i \leq P$, other clusters can consume at most $P - Q_i$ memory time within each regulation period. Thus, a task T_i can be delayed for at most $P - Q_i$ time in each regulation period. We distinguish two modes to compute the memory delay of any task. Unlike the analysis in [142] which computes a delay term and adding it to the memory time of the task without interference, our analysis proceeds by computing an inflation factor for memory. This is done for simplicity, but the two concepts are effectively equivalent: given a memory time e_i^m and a delay Δ, the inflated memory time is simply $e_i^m + \Delta$. As an example, in the case of pure RR contention delay, [142] would compute a delay term $\Delta = e_i^m \times (n-1)$, while in our case we consider an inflated memory time $e_i^m \times n$.

Regulation mode

The task in this mode consumes its budget Q_i and waits until the next period, i.e., it suffers a delay of $P - Q_i$ in each period. For this mode, the inflated memory time can be computed as follows.

$$\left\lfloor \frac{e_i^m}{Q_i} \right\rfloor \times P + P \tag{5.9}$$

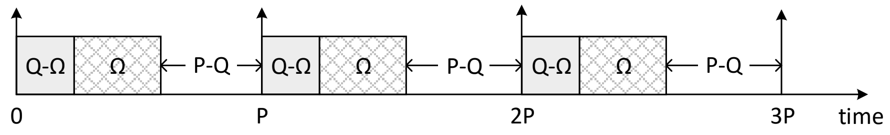
The formula states that the number of regulation periods is $\lfloor e_i^m / Q_i \rfloor$. For each regulation period, the inflated memory time is P (Q_i memory time plus $P - Q_i$ delay). Any remaining ($e_i^m \bmod Q_i$) memory must complete within one last period; hence, its inflated time can be bounded by P. Since we assume any arrival pattern of memory requests, we consider the worst-case pattern that leads to maximum delay. That is, all memory requests are clustered at the beginning of regulation periods without computation between them. If, however, there is computation between them, say for x time, the task will be stalled for $P - Q_i - x$ in each regulation period.

Contention mode

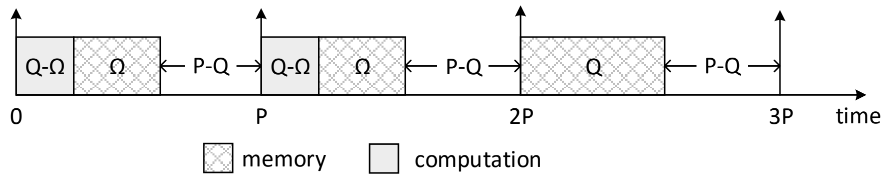
Unlike the previous mode, the task in this mode consumes less than its budget, and the contention of its memory causes the maximum delay of $P - Q_i$. Due to RR arbitration, every memory access can be delayed by $(n-1)$ requests of other clusters. Hence, the task must access memory for Ω_i time to suffer the maximum delay of $P - Q_i$ within each period. This can be stated as follows.

$$\begin{aligned} \Omega_i \times (n-1) &= P - Q_i \\ \Omega_i &= \frac{P - Q_i}{n-1} \end{aligned}$$

We observe two possible cases under contention mode as illustrated in Figure 5.3 for three regulation periods. The intuition behind the following analysis is to think that each task has memory and computation, and within each regulation period, the task can spend Q_i time by either computation or memory. In addition, the task can be delayed by $P - Q_i$ when it generates Ω_i amount of memory within each regulation period as stated above. Since, as per our task model, the positions or even the order of computation and memory are unknown at compile time, we try to bound the memory time by constructing the worst case pattern between memory and computation that leads to maximum memory delay.



(a) Enough computation to fill $Q - \Omega$ in each regulation period.



(b) No enough computation o fill $Q - \Omega$ in each period.

Figure 5.3: Two sub-cases for contention mode.

In case (a), there is enough computation to fill $Q - \Omega$ gap in every regulation period as shown in Figure 5.3(a). This causes all memory to suffer contention. Hence, the inflated memory time is simply $e_i^m \times n$. However, in case (b), there is no enough computation to fill $Q - \Omega$ gap in each regulation period, for example, period 3 in Figure 5.3(b). Thus, the task has to execute memory because the task budget is not consumed yet. In this case, the inflated memory time can be upper bounded by

$$\left\lfloor \frac{e_i^*}{Q_i} \right\rfloor \times P + P, \tag{5.10}$$

where $e_i^* = e_i^m + (e_i^x - L_i)/m_i + L_i$. We consider e_i^*, the combined memory and computation time, since either of them can execute within Q_i time. Similar to before, $\lfloor e_i^*/Q_i \rfloor$ represents the number of regulation periods. For each period, the task executes for Q_i (memory and computation) and waits for $P - Q_i$. The remaining execution time is again upper bounded by P.

We note that the memory time under contention mode cannot exceed $C_i^m \times n$ because we assume per cluster buffer and RR arbitration policy at the hardware level. Thus, the memory time will be the minimum between case (a) and case (b). In contrast, the memory time under regulation mode may exceed $C_i^m \times n$ due to the regulation effect.

We just described how to compute the memory time under each mode. We now specify a condition upon which we determine the mode that should be used to bound the memory time.

Lemma 22. *When $Q_i < \Omega_i$, the memory time under regulation mode is longer than the memory time under contention mode. On the other hand, when $Q_i \geq \Omega_i$, the memory time under case (a) or case (b) of contention mode is longer than or equal the memory time under regulation mode.*

Proof. Clearly, when $Q_i < \Omega_i$, the task cannot cause the maximum delay of $P - Q_i$ by contention. Since the task is delayed by $P-Q_i$ (the maximum possible delay) in each period under regulation mode, the memory time under this mode is longer than the memory time under contention mode. On the other hand, when $Q_i \geq \Omega_i$, the contention mode leads to longer or equal memory time even though, under both modes, the task is assumed to receive $P - Q_i$ delay in each period. We recall that under regulation mode, the task is assumed to consume its budget Q_i in each period. In contrast, case (a) of contention mode assumes that the task consumes $\Omega_i \leq Q_i$ budget in each period while case (b) assumes the task consumes Ω_i in some periods and Q_i in some other periods. In both cases, the number of regulation periods cannot be less than the number of periods under regulation mode in which the task is assumed to consume Q_i memory in each period. □

We can safely remove the floor function from (5.9) and (5.10), and the formulas still serve as an upper bound to memory time. In addition, we propose to let $Q_i/P = q_i$. Thus, (5.9) can be re-written as follows.

$$\frac{e_i^m}{q_i} + P \tag{5.11}$$

Similarly, the memory time in contention mode is bounded by (5.10) and cannot exceed $e_i^m \times n$. Thus, the memory time can be re-written as

$$\min(e_i^m \times n, \frac{e_i^*}{q_i} + P). \tag{5.12}$$

Since the formulas are approximated and Q_i is replaced by $P \times q_i$, the conditions in Lemma 22, which is based on Q_i, can be re-stated in terms of q_i as follows.

Lemma 23.

$$\textit{memory time} = \begin{cases} (5.11) & \textit{if } q_i < (n - P/e_i^m)^{-1}, \\ (5.12) & \textit{otherwise.} \end{cases}$$

Proof. We prove this lemma by showing that the memory time under each mode is no less than the memory time under the other mode for the corresponding value of q_i. For regulation mode, $q_i < (n - P/e_i^m)^{-1}$. Thus,

$$\begin{aligned}\frac{e_i^m}{q_i} + P &> e_i^m \times (n - P/e_i^m) + P \\ &> e_i^m \times n - P + P \\ &> e_i^m \times n.\end{aligned}$$

It is enough to only check with $e_i^m \times n$ because the memory time in contention mode cannot exceed $e_i^m \times n$. Similarly, the memory time under contention mode can be either

$$\frac{e_i^*}{q_i} + P \geq \frac{e_i^m}{q_i} + P,$$

since $e_i^* \geq e_i^m$, or $e_i^m \times n$ in which

$$\frac{e_i^m}{q_i} + P \leq e_i^m \times (n - P/e_i^m) \leq e_i^m \times n.$$

□

By adding the computation time to the inflated memory time, we obtain the following makespan bounds under each mode. For regulation mode, we have:

$$e_i^{reg} = \frac{e_i^m}{q_i} + P + \frac{e_i^x - L_i}{m_i} + L_i, \tag{5.13}$$

and for contention mode, we have:

$$e_i^{cnt} = \min(e_i^m \times n, \frac{e_i^*}{q_i} + P) + \frac{e_i^x - L_i}{m_i} + L_i. \tag{5.14}$$

Similar to (5.4), we can derive $q_i^{reg}(m_i)$ from (5.13) and $q_i^{cnt}(m_i)$ from (5.14) by enforcing the schedulability test $e_i^{reg} \leq D_i$ and $e_i^{cnt} \leq D_i$, respectively.

In Figure 5.4, we show a graphical representation of these two functions for a given task T_i. The bandwidth fraction q_i ranges from 0 to 1 and the middle bold dotted line represents the condition as in Lemma 23 in which the lower area is for regulation mode and the upper area is for contention mode. Each point in these two curves represents a pair of values (q_i, m_i) in which the task is schedulable. The m_i^n vertical line represents $e_i^m \times n$

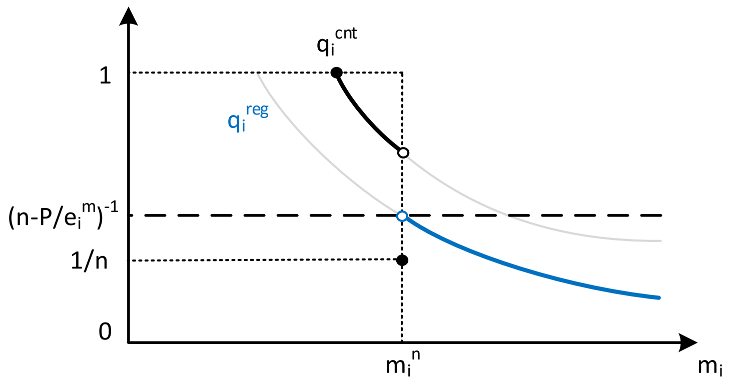


Figure 5.4: Two modes of operation for Memguard.

and computed as in (5.8). In contention mode, the memory time cannot exceed this line as indicated by min function in (5.14).

We propose to apply Optimal-Assign to obtain q_i bandwidth fractions by assuming each task operates in one particular mode. This approach results in 2^n possible combinations as each task can be in two different modes. As a final step, we let $Q_i = P \times q_i$, and then program Memguard's regulators to enforce the bandwidth assignment.

An important point to note is that by having tasks operating in different modes, Memguard is able to assign unequal fractions of memory bandwidth in which some tasks have $q_i > (n - P/e_i^m)^{-1}$ and some others have $q_i < (n - P/e_i^m)^{-1}$. In contrast, nRR always assigns $q_i = 1/n$ for all tasks.

5.6.2 Harmonic RR

Harmonic RR is known to achieve 100% bandwidth utilization. Therefore, our goal in this section is to find a set of harmonic fractions by trading a minimum number of cores. With harmonics of base 2, each $q_i = 1/2^j$ and j is an integer positive number.

A naive approach is to search the whole space of all possible harmonics and take the set of harmonics that gives the minimum number of cores. However, this may take $(n-1)^n$ number of iterations. It is because the range of possible harmonics for each task is $1/2^1$ to $1/2^{n-1}$. We elaborate on this as follows. The goal of harmonics assignment is to

Harmonic-Assign

```
1   $q_i = 1, \forall T_i \in \Gamma$
2   compute $\Theta_i(q_i)$ as in (5.15)
3   $target = 1$
4   **repeat**
5       **while** $BU > target$
6           $j =$ find-one($T_i \in \Gamma : \Theta_i(q_i)$ is maximum)
7           $q_j = q_j/2$
8       **if** $BU < target$
9           $rem = target - BU$
10          $indx =$ find-all($T_i \in \Gamma : q_i > rem$)
11          $target = target - \sum_{i \in indx} q_i$
12          remove $T_i : i \in indx$ from $\Gamma$
13          $q_i = 1, \forall T_i \in \Gamma$
14  **until** $BU = target$
15  $m_i = \lceil m_i(q_i) \rceil, \forall T_i$
16  $test = (m^{used} \leq m)$
```

achieve 100% utilization. With $n-1$ tasks, the largest sum of harmonics less than 1 is $1/2 + 1/4 + \cdots + 1/2^{n-1}$. It is easy to see that increasing any of them gets the sum greater than or equal to 1. Now, the n^{th} task can be assigned a value of $1/2^{n-1}$ and the total utilization sums up to 1. We observe that any smaller harmonics will never sum up to 1.

Instead, we propose an efficient algorithm with polynomial-time complexity. Although it is not optimal, it achieves very close to optimal results as shown in Section 5.7.

The basic intuition to Harmonic-Assign algorithm is to find the task that gives the maximum decrease of memory utilization with minimum increase in m_i. This metric is captured by finding the maximum $\Theta_i(q_i)$ as in Line 6, where

$$\Theta_i(q_i) = \frac{q_i - q_i/2}{m_i(q_i/2) - m_i(q_i)}. \tag{5.15}$$

The algorithm then increases q_i in harmonic steps, i.e., $q_i = q_i/2$ as in Line 7. Here, m_i is initially assumed to be real number and derived from (5.3) for a given q_i as follows.

$$m_i(q_i) = \frac{(e_i^x - L_i) \times q_i}{(D_i - L_i) \times q_i - e_i^m} \tag{5.16}$$

The algorithm at the final step (Line 15), rounds each m_i to the closest integer value.

The algorithm initially set *target = 1* and set all bandwidth fractions to 1. Then, the **while** loop decreases the fractions until $BU \leq target$. If the total bandwidth utilization is equal to *target*, the algorithm stops: we achieved 100% memory bandwidth utilization. If, however, the total bandwidth utilization is less than the target, we can stop, but the remaining bandwidth will be unused. We recall that the bandwidth utilization is a decreasing function of the number of cores, i.e., in this case we may waste some cores for unused bandwidth.

Therefore, we design the algorithm to continue until we reach 100% bandwidth utilization. We set the remaining fraction to *rem*. We then remove all T_i such that $q_i > rem$ from Γ. The intuition is that we need at least a similar fraction to increase any fraction to the next harmonic, for example, we need at least 1/4 to increase 1/4 to 1/2. The algorithm then repeats until there is no remaining bandwidth. We note that find-all returns the indices of all tasks that satisfy the condition.

Lemma 24. Harmonic-Assign *is guaranteed to terminate.*

Proof. It is guaranteed because the algorithm will remove at least one task in Line 12. Assume that $BU = \sum_{i \neq j} q_i + q_j > target$, and the fraction of T_j is reduced to the next harmonic such that $\sum_{i \neq j} q_i + q_j/2 \leq target$. We observe that

$$\sum_{i \neq j} q_i + q_j/2 + q_j/2 > target,$$

as per our assumption. By rephrasing the terms, we have

$$q_j/2 > target - (\sum_{i \neq j} q_i + q_j/2) = rem.$$

In other words, the algorithm is guaranteed to remove T_j. □

The complexity of Harmonic-Assign algorithm is $O(n^4)$. We note that the **while** loop iterates n^2 times in case we increase the harmonics of all tasks from 2^1 to 2^{n-1}. In each iteration, we update $\Theta_i(q_i)$ for n tasks. In addition, the **repeat** loop can repeat n times as in the worst case, we remove just one task in Line 12.

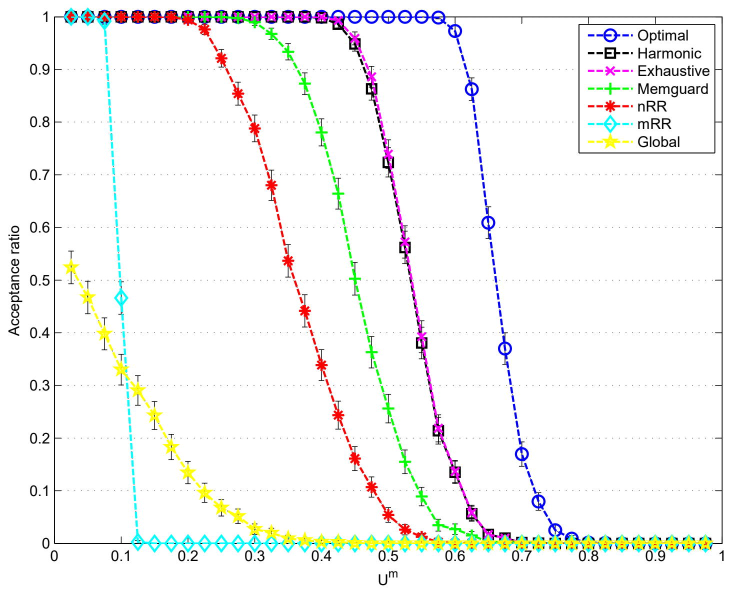


Figure 5.5: The schedulability for all arbitration policies with acceptance ratio metric and varying Um.

5.7 Evaluation

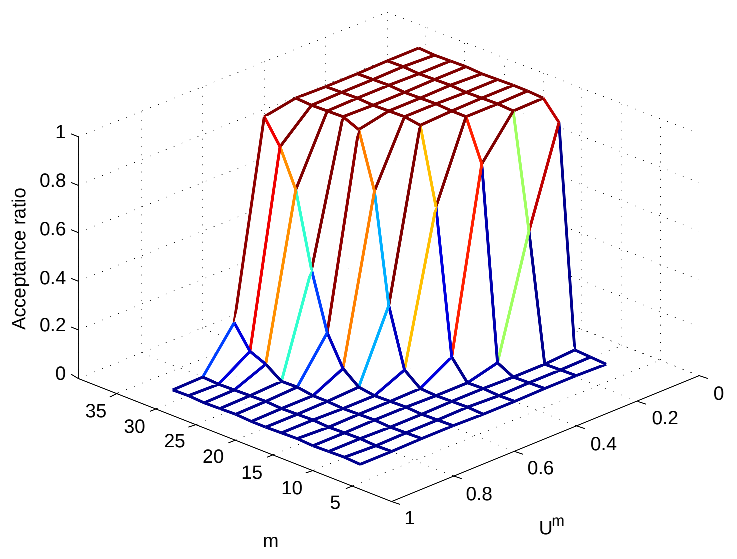
In this section, we evaluate the performance of our proposed method. The task sets are randomly generated in our experiments and their parameters are set as follows. Since we consider two different resources (memory and processor), the task sets have to be characterized by two values, namely, Um and Ux. We use UUNIFAST [28] algorithm to generate task sets with uniform distribution in the space of utilization values. For memory utilization, we set UUNIFAST(n, Um) to generate n memory utilization values u_i^m such that their summation equals to Um and each value falls within $[0, Um]$ range. Similarly, we generate u_i^x values as $1 +$ UUNIFAST$(n, Ux - n)$ to ensure that each task is high utilization. The deadline values are randomly generated within the range $[10, 100]$ ms. Then, the memory time e_i^m and computation time e_i^x are set accordingly as $u_i^m \times D_i$ and $u_i^x \times D_i$, respectively. The critical paths L_i are set to be $0.2 \times (D_i - e_i^m)$. We vary such percentage within a range of $[0, 1]$ in one of our experiments. For each point in the experiments, we take the average value over 1000 task sets. The experiments are repeated 100 times and the error bars represent the first decile and the ninth decile.We use default values of $n = 5$, $Ux = 10$, $m = 32$ in our experiments unless otherwise specified.

In the first set of experiments, we use the *acceptance ratio* metric to evaluate schedulability. It is the number of accepted task sets over the total number of generated ones. We compare all arbitration policies discussed above. The Um values are varied from 0.025 to 0.975 with 0.025 step size. As shown in Figure 5.5, the OPTIMAL-ASSIGN algorithm dominates all methods. In addition, nRR significantly outperforms mRR. Between OPTIMAL-ASSIGN and nRR, both Memguard and harmonic RR show good improvement. The graph indicates that both methods are lower bounded by nRR and upper bounded by OPTIMAL-ASSIGN. An important point to note is that nRR assigns equal bandwidth fractions for all tasks. For task sets with different memory requirements this equal assignment is not optimal. Both harmonic RR and Memguard are good to balance the assignment but harmonic RR is more flexible. We note that there is only a marginal difference between our proposed heuristic in HARMONIC-ASSIGN algorithm and the exhaustive search of all harmonics.

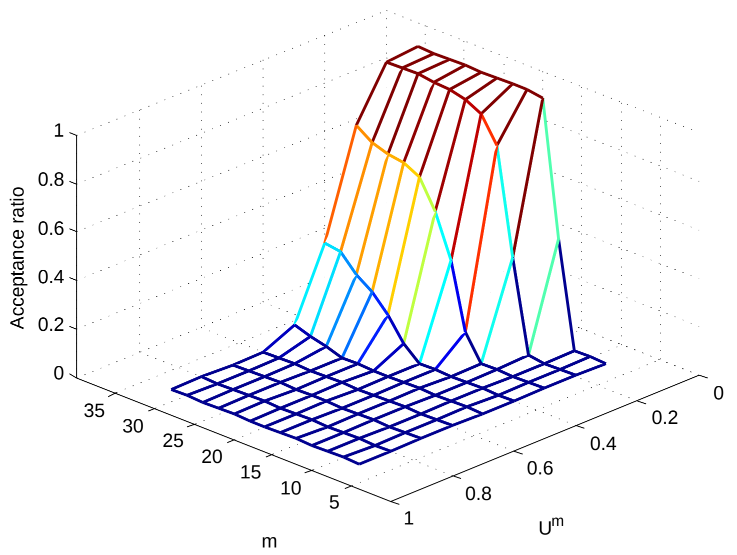
We also test against global EDF using the response time analysis (RTA) as in [100] which has been shown to have better results than the test in [18] and the capacity augmentation bound of 2.618 as in [84]. In this test, we inflate the memory portion of each task by m, the number of cores. Since there is dependence on the number of cores and the inflation factor, we start by one core, we then iteratively increase the number of cores until we reach convergence or the task set is declared unschedulable. The schedulability result of global EDF is poor compared to nRR, our base line method. The reason is that global EDF does not improve the memory portion upon nRR since it uses RR and one request buffer for each core.

In the second set of experiments, we vary both Um and m. We plot in Figure 5.6 the acceptance ratio as a third dimension to see the effect of both parameters. The area under the curve (the schedulability) is clearly larger for OPTIMAL-ASSIGN. The extra cores are utilized to balance the memory part. On the other hand, the extra cores cannot be utilized with rigid nRR in which each task is assigned $1/n$ bandwidth fraction.

In the third set of experiments, the number of cores used, m^{used}, to reduce the bandwidth utilization below or equal one is used as a metric to see how each algorithm can utilize system's cores. Unlike previous experiments where m is imposed, here we set $m = \infty$ and plot the number of cores used by each algorithm. Again, we vary the total memory utilization values from 0.1 to 0.8. For the sake of fair comparison, we generate task sets that are feasible with respect to nRR, i.e., $u_i^m < 1/n$, otherwise, the task set will not be schedulable because the memory time will be inflated by n. The algorithm UUNIFAST is modified to discard all $u_i^m \geq 1/n$. We note that the maximum value of Um is set to 0.8 because it is impossible to achieve a utilization of 1 with n tasks and each $u_i^m < 1/n$. The results in Figure 5.7 clearly show a great advantage of OPTIMAL-ASSIGN against



(a) Optimal-Assign



(b) nRR

Figure 5.6: The acceptance ratio by changing m and Um.

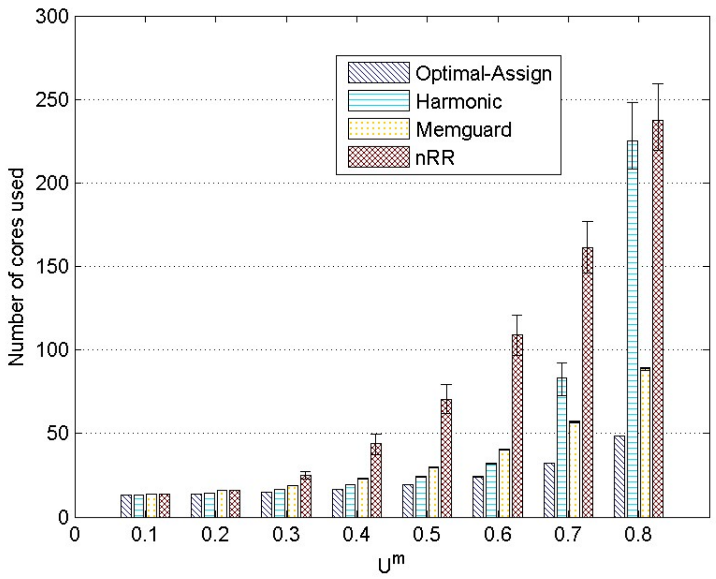


Figure 5.7: The number of cores used, m^{used}.

nRR with respect to reducing the number of cores. An interesting point to note is that Memguard is achieving better results than harmonic RR when memory utilization is high. It is because harmonic RR assigns fractions with harmonic granularity $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \ldots,$ while Memguard assigns fractions in a finer granularity.

In the last set of experiments, we vary L_i of each task to be within a given percentage of $(D_i - e_i^m)$. We use UUNIFAST(n, L) to generate n percentages such that their summation is equal to L and each percentage falls within $[0, L]$ range. We use weighted schedulability [23] to understand the sensitivity of each algorithm to L_i. We can think of weighted schedulability metric as the area under the schedulability curve, i.e., each point in Figure 5.8 corresponds to the area under the curve in Figure 5.5 for each L value. We recall that Memguard is able to partition the bandwidth unequally as long as there are some tasks operating in different modes. The results indicate that increasing L_i affects Memguard non-linearly while affects others linearly. It is because the advantage of Memguard relies on the amount of computation time $(e_i^x - L_i)/m_i + L_i$ of parallel tasks as detailed in Section 5.6.1. In contrast, L_i affects other methods by reducing the slack time $(D_i - L_i)$ only. We also note that all methods converge to one point when $L = 1$. In other words, tasks are mostly sequential and there is no advantage in trading more cores.

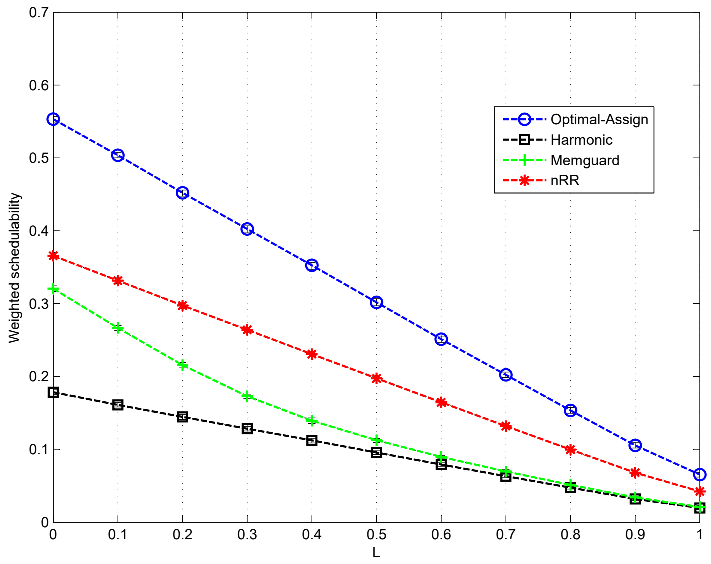


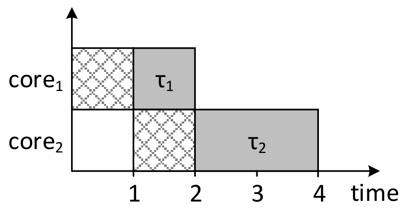
Figure 5.8: The effect of changing L_i.

5.8 Static Scheduling of 3-phase Parallel Task

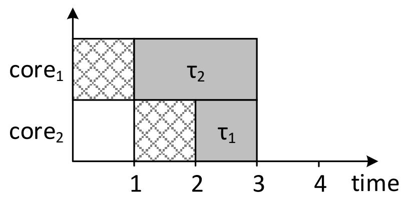
In previous discussion, we abstract the parallel task using three parameters (e_i^m, e_i^x, L_i) and use a coarse grained bound on the makespan as in (5.1), and we assume that the memory time is additive to the makespan as in (5.3) since we cannot provide a guarantee on the amount of overlap.

In this section, instead, we assume that the parallel task is modeled as a DAG, $G_i = (V_i, E_i)$. Here, $V_i = \{\tau_{i,1}, \tau_{i,2}, \ldots, \tau_{i,n_i}\}$ is a set of n_i subtasks and E_i is a set of edges. If $(\tau_{i,a}, \tau_{i,b}) \in E_i$ is a directed edge, then $\tau_{i,a}$ has to execute before $\tau_{i,b}$, i.e., there is a precedence constraint in their execution. Each subtask $\tau_{i,j}$ is divided into three phases: load, computation and unload with the following notation for their lengths: $e_{i,j}^v$, $e_{i,j}^x$ and $e_{i,j}^w$, respectively. Similar to before, each dag task is recurrent with period p_i and deadline D_i. In addition, each dag task is given a dedicated cluster with m_i cores and q_i bandwidth fraction.

In this section, we describe mthPREM, a static co-scheduler of application 3-phase subtasks. The objective of this scheduler is to minimize the application's *makespan*, i.e., the finish time of last subtask. In general, scheduling involves two steps: (1) allocation of cores to subtasks and (2) execution ordering of subtasks. The latter step matters even



(a) (τ_1, τ_2)



(b) (τ_2, τ_1)

Figure 5.9: The order of execution decides the makespan length.

though there is no precedence constraints between two subtasks. It is easy to see that different execution ordering leads to different lengths of makespan. For instance, assume a parallel task with two subtasks τ_1 and τ_2. The task index is omitted for simplicity. We show in Figure 5.9 two schedules with two different execution ordering. The execution order (τ_1, τ_2) has longer makespan than (τ_2, τ_1).

The computed schedule determines on which core each subtask should execute and the start time of its memory phase. There is no need to maintain the start time of computation phase because it starts immediately after the memory phase. There are two possible ways to implement the schedule of memory phase of each subtask. First, the order of these phases can be activated on-line using a timer. In this case, the multicore processor should have a common clock to synchronize the activation of memory phases across all cores. Another way of activating these phases, according to the schedule order, is by using semaphores. In this implementation, each two consecutive memory phases share a common semaphore where one waits and the other signals. This semaphore mechanism propagates through the memory phase of each subtask to enforce the execution order for a given schedule.

When the number of subtasks does not exceed the number of cores and there are no precedence constraints between them, scheduling involves only execution ordering. A polynomial-time algorithm that schedules these subtasks can be easily constructed. The intuition behind this algorithm is that subtasks with large computation phases should start first so that their contribution to the makespan is reduced. Another observation is that the contribution of memory phases to makespan is the same regardless of the execution ordering because they are serialized.

On the other hand, when the number of subtasks exceeds the number of cores, i.e., $n_i > m_i$, scheduling involves both core allocation and execution ordering as explained mthPREM-static

```
1  $t_{mem} = 0$
2  for each $core_j$
3      $wb(core_j) = 0$
4      $t_f(core_j) = 0$
5  Insert subtasks $\tau_{i,j} \in T_i$ into list L, according to the schedule order
6  for each $\tau_{i,j} \in L$
7      $t_m(\tau_{i,j}) = \max(t_{mem}, t_f(proc(\tau_{i,j})))$
8      $t_{mem} = t_m(\tau_{i,j}) + wb(proc(\tau_{i,j})) + e^v_{i,j}$
9      $t_f(proc(\tau_{i,j})) = t_{mem} + e^x_{i,j}$
10     $wb(proc(\tau_{i,j})) = e^w_{i,j}$
11 Sort cores $core_j$ into list L, in ascending order according to their finish time
12 for each $core_j \in L$
13     $t_w(core_j) = max(t_{mem}, t_f(core_j))$
14     $t_{mem} = t_w(core_j) + wb(core_j)$
15     $t_f(core_j) = t_{mem}$
16 return $\max_{j \in \{1,\ldots,m_i\}} t_f(core_j)$
```

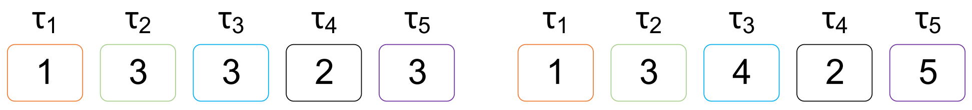
above. In order to measure the quality of the schedule, we define the application makespan as the schedule cost function. Thus, the cost of schedule S is

$$cost(S) = \max_{j \in \{1,\ldots,m_i\}} t_f(core_j), \tag{5.17}$$

where $t_f(core_j)$ is the core finish time, which is equal to the finish time of last subtask scheduled on that core.

mthPREM-static algorithm shows how the schedule is constructed for each parallel task T_i given the core allocation, $proc(\tau_{i,j})$ and the execution ordering. The output of mthPREM-static algorithm is: (1) $t_m(\tau_{i,j})$, the start time of each subtask memory phase and (2) $t_w(core_j)$, the start time of unload phases for last scheduled subtasks on each core.

The **for** loop in Line 6 schedules the load and unload phases of all subtasks except the unload phases of last scheduled subtasks on each core. The max in Line 7 is used to schedule memory phase after computation phase. The $wb()$ in Line 8 holds the unload phase of the previous subtask, scheduled on the same core, to be merged with the load



(a) Core allocation (3 cores). (b) Execution ordering.

Figure 5.10: GA chromosomes.

phase of the current subtask. The **for** loop in Line 12 schedules the unload phases of last scheduled subtasks on each core. The cores are first sorted in ascending order according to their finish time to reduce the contribution of unload phases to the makespan.

So far, we computed the schedule in mthPREM-static assuming that core allocation (the spatial assignment) and execution ordering (the temporal assignment) are given. Now, we discuss the problem of optimizing such decisions. In fact, finding a schedule of minimum cost is NP-hard [42], because as number of subtasks and cores increase, the total number of possible schedules becomes vast. Consequently, it is impractical to do an exhaustive search to find the optimal solution. Stochastic search algorithms are good candidates for tackling such problems. Random search is one possible technique. In [116], the authors show that evaluating several hundreds or several thousands random schedules is enough to get, with high confidence, close to optimal solution, given that the random samples are independent and identically distributed.

Moreover, meta-heuristic search algorithms, such as genetic algorithm (GA), is another technique often used to speed up the search process. Therefore, we develop a genetic algorithm to search for a good solution to mthPREM. For core allocation, we use the value encoding scheme to represent the solution as shown in Figure 5.10(a). In this solution encoding (the chromosome), each subtask is assigned an integer number from 1 to m_i, the number of cores, corresponding to its core allocation. For execution ordering, we use the permutation encoding to represent the solution as shown in Figure 5.10(b). In this solution encoding, each subtask is assigned an integer number from 1 to n_i, the number of subtasks, corresponding to its execution order. GA procedure outlines the major steps of the genetic algorithm where G is the number of generations.

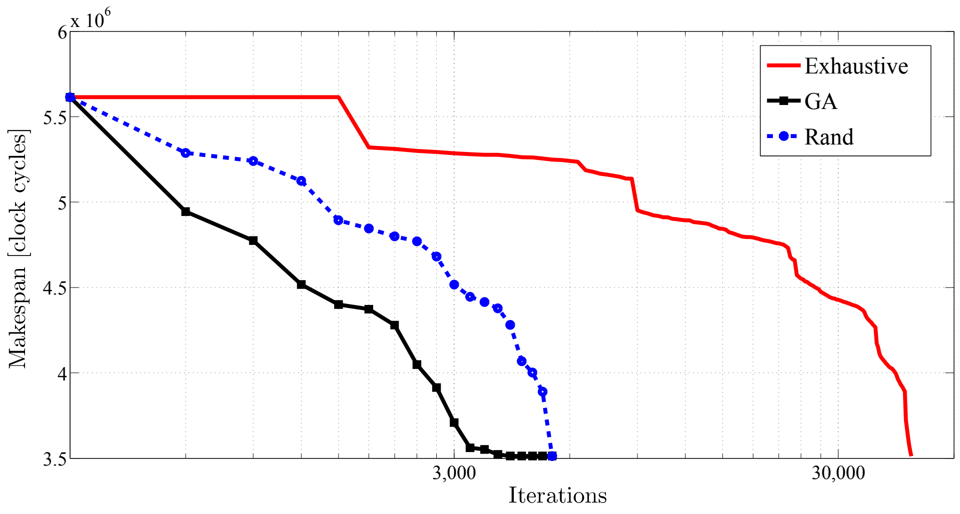


Figure 5.11: Comparison between exhaustive, random and GA.

```
GA
1   Create initial population of random 100 schedules
2   Evaluate the schedules as in mthPREM-static
3   for $i = 1 \to G$
4       Move the best 50 schedules to next generation
5       Crossover the best 50 allocation/ordering chromosomes
6       Mutate the best 50 ordering chromosomes
7       Evaluate the new 50 schedules
8   return the best schedule
```

We employ a single point *crossover* between two chromosomes for both core allocation and execution ordering. For mutation operator, we randomly pick two subtasks, then swap their order. However, execution ordering has to be corrected to make sure that no two subtasks have the same order and the precedence constraint between any two subtasks is not violated. Figure 5.11 shows a comparison between GA, random and exhaustive search for an application with 6 subtasks and 2 cores processor. Both GA and random search run for 5000 iterations, and the exhaustive runs for the whole search space which is 46080 iterations. Note that we set $G = 98$ for GA to get 5000 iterations. The GA algorithm reaches close to 5% of the optimal solution in less than 3000 iterations whereas random search reaches the same value at close to 5000 iterations.

Table 5.2: NPB characteristics

benchmark	parallel	barrier
is	16	22
ep	4	0
cg	51	2057
ft	42	0
bt	312	255
sp	916	415
lu	118	269

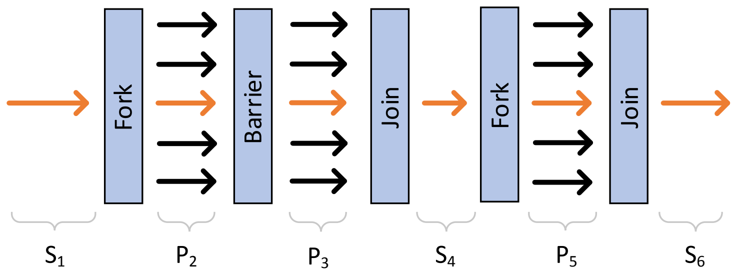


Figure 5.12: Fork-join parallel model.

5.9 mthPREM Evaluation

We evalaute mthPREM on OpenMP NAS Parallel Benchmarks (NPB) [72]. These benchmarks employ the *fork-join* parallel model in which the application alternates between sequential and parallel segments as depicted in Figure 5.12. The application starts as a single sequential subtask until a parallel construct is encountered such as `parallel start` and `parallel end` pair in which the application splits into multiple concurrent subtasks. In addition, the parallel segment may be followed by multiple other parallel segments that are separated by *barrier* synchronization. After the parallel segment, all subtasks synchronize again into one sequential subtask. This fork-join structure can be repeated multiple times. Fork-join is a popular programming model employed in systems such as OpenMP and Java [81]. The benchmarks consist of five parallel kernels and three simulated applications. Table 5.2 shows the number of OpenMP parallel constructs for these benchmarks.

We evaluate the performance of mthPREM using the framework shown in Figure 5.13. We develop a dynamic instrumentation tool based on Pin [12] to analyze the NPB benchmarks. A central element of the tool is a pervasive memory profiler to capture the memory

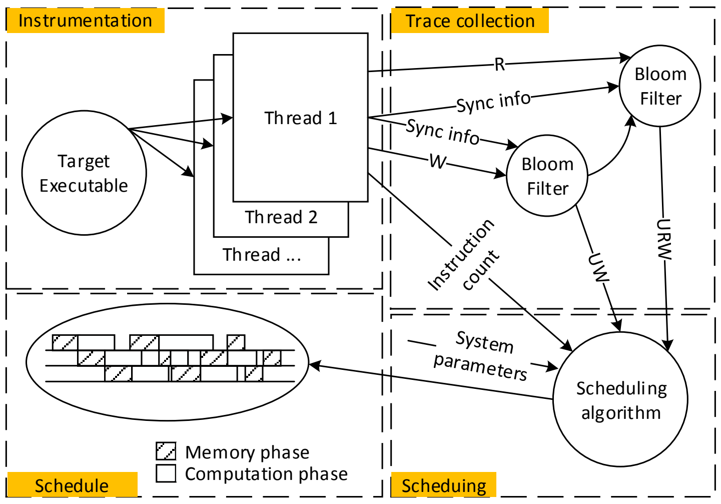


Figure 5.13: Evaluation framework.

traces for each subtask. In real applications, the memory profiler is expected to capture hundreds of millions of events [96]. Thus, we add some strategies to reduce the number of captured events that are enough to evaluate mthPREM. First, we trace memory accesses at the granularity of cache lines which cut the size of the generated trace by 16 in the case of 64 bytes cache lines. Second, the load and unload phases are mainly determined by the number of *unique* memory blocks accessed during the computation phase. In particular, both unique read and write cache lines (URW) are captured for load phase of each subtask, and the unique write cache lines (UW) are captured for unload phase of each subtask. Each subtask has to load write-data to prevent write misses during the computation phase. To capture only the unique addresses, we use a *bloom filter* for each subtask. Furthermore, the tool captures the synchronization information of the application for each subtask (e.g., `parallel` and `barrier`). This synchronization points are used to reset the counters for URW and UW, and record the trace information for each subtask during a parallel segment.

As system parameters, we model a simple 4-core processor. Each application is compiled and the instruction count is used to determine the computation phase for each subtask. We assume that non-memory instructions and memory instructions that hit in local memory take one clock cycle. The memory access time for one cache line of size 64 bytes is assumed to be 150 clock cycles. We apply MTHPREM-STATIC algorithm for each application based on the generated trace and system parameters.

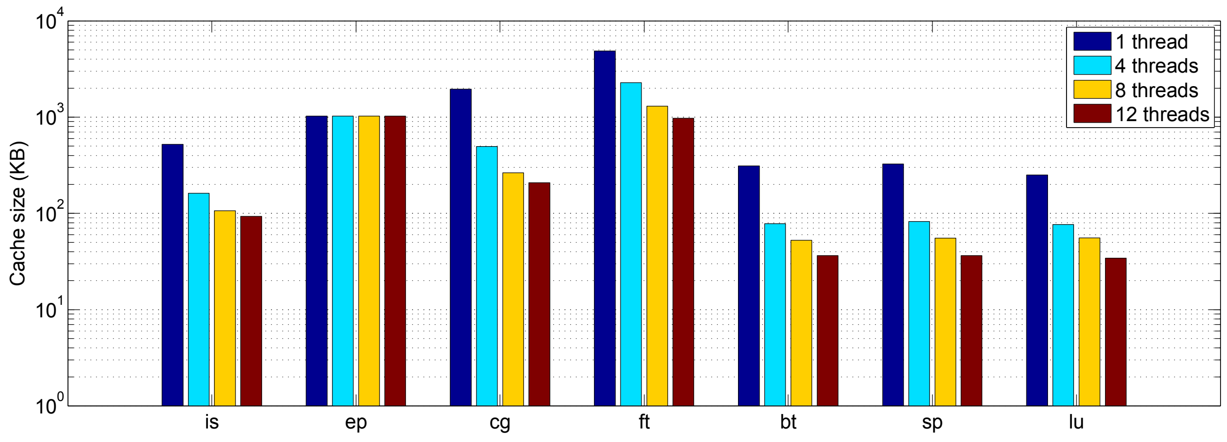


Figure 5.14: Maximum memory footprint.

Since we assume that the memory footprint of maximum subtask fits inside the local memory of one core, i.e., there is no capacity misses, we show in Figure 5.14 the maximum memory footprint for all benchmarks. It is clear that the memory footprint of one application decreases as the number of subtasks increases. In other words, with a fixed size of local memory, changing the number of subtasks allows some applications to execute according to mthPREM. We note that for some benchmarks like EP, the maximum memory footprint remains constant. It is due to the type of parallelism within the parallel segment. For work sharing constructs such as `parallel for`, the data is distributed and processed collaboratively by all subtasks. Therefore, the amount of data each subtask needs to process reduces as number of subtasks increases. In contrast, constructs like `sections`, implement different type of parallelism known as task parallelism in which subtasks concurrently execute independent codes. Hence, the amount of data processed remains constant, but the code size for each subtask changes.

Figure 5.15 shows the simulation results for NAS Parallel Benchmarks, all with class S data set, 8 subtasks and 4 cores. These results are obtained by running the GA procedure for 5000 iterations. The *no contention* bar is the best case execution where the memory time is not inflated as in contention execution. The no contention execution is really optimistic, and it is reported to see the lower bound of mthPREM. The results show that mthPREM achieves a good improvement over contention execution with overall speedup of 1.21%. The **weighted arithmetic mean (WAM)** is used to compute the overall speedup of the benchmark suite. The weights are computed as $w_i = e_i / \sum_{\forall T_i} e_i$ where e_i is the makespan under mthPREM. An important point to note is that the benchmarks show different improvements because they have different ratios of memory to computation. As

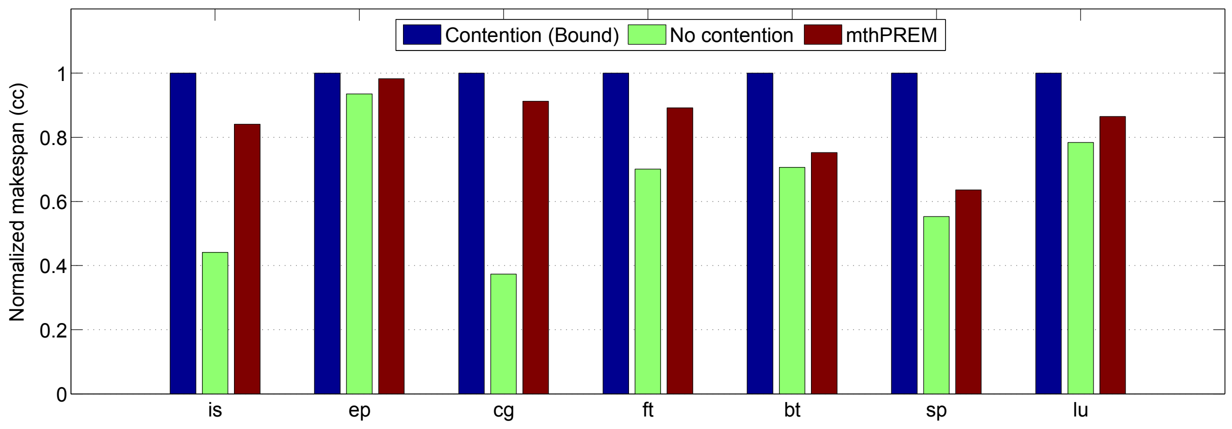


Figure 5.15: Simulation results for 8 subtasks and 4 cores processor.

this ratio increases, the improvement of mthPREM also increases.

In 3-phase execution model, we need to load each subtask into the local memory before execution. With unit-stride memory blocks, only one address pointer is sufficient to load all required data. In contrast, non-stride memory blocks need a data structure to keep track of all memory blocks. This clearly adds an overhead to load phase. We note that the same applies for unload phase. Indeed, the overhead depends on how the layout of memory blocks is scattered in main memory. We divide the footprint of each subtask into chunks of contiguous memory blocks, and we define the scatter ratio as the ratio between the number of chunks and the total number of memory blocks. In other words, 0% scatter ratio means continuous layout of all memory blocks. Figure 5.16 shows the load overhead for all benchmarks. In some benchmarks, almost all memory blocks are consecutive, and for others, there is some scattering but the ratio is limited by less than 10%.

5.10 Summary

Federated scheduling is an elegant approach to schedule parallel tasks on dedicated cores. Even though tasks do not share cores between each other, they can interfere through shared resources such as main memory. We propose a novel method to consider memory bandwidth when assigning cores to parallel tasks. We first integrate the memory demand of each task in the task model. Then, we formulate the execution time of parallel tasks as a function of (1) the number of assigned cores and (2) the amount of assigned bandwidth. The proposed method tunes system's resources (processor and memory) to fit the need of

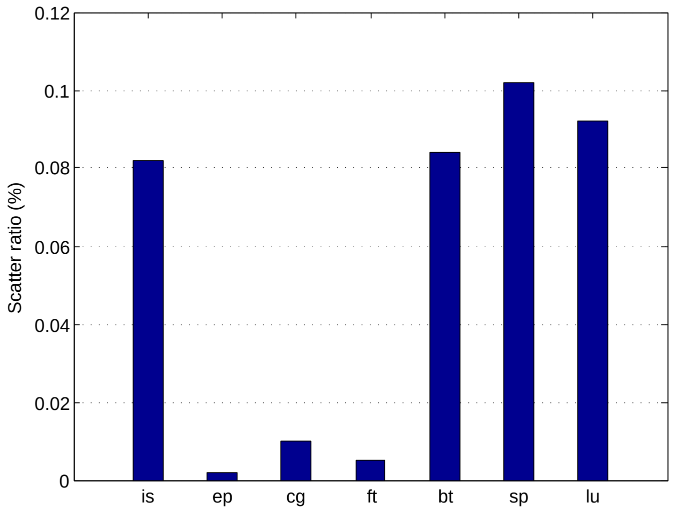


Figure 5.16: Memory scatter ratio.

system's tasks. We demonstrate the proposed strategy on two arbitration policies, one is software based and the other is hardware based. The results show a big advantage of our method compared to memory oblivious approaches.

In this chapter too, we describe an algorithm to schedule 3-phase parallel tasks in which each parallel task is assigned a dedicated number of cores and memory bandwidth. The scheduler objective is to minimize the application's makespan by finding the best execution ordering that hides the memory latency. This algorithm is evaluated on a set of NAS parallel benchmarks, and showed good improvement over contention execution in which subtasks access main memory without control.

Chapter 6

Conclusion

Even though multicore processors offer increased performance over single-core, their adoption to execute hard real-time applications is challenging. The use of physical shared resources in these platforms, such as memory bus and I/O, introduces variability in the execution time of applications. In this dissertation, we consider memory contention due to sharing the memory bus between multiple cores. Our solution is based on a co-scheduling approach in which each real-time task is decoupled into two phases: memory and computation. Thus, the system scheduler can mediate, from a software level, the use of processor cores as well as the main memory. Memory contention is avoided by allowing only one memory phase to be active at any given time. Furthermore, the execution of real-time tasks is improved by overlapping computation phases with memory phases.

We show how to harness this overlap between tasks on different cores using gPREM, a global scheduler of memory and computation phases. We propose for this scheduler a schedulability analysis to determine whether a given task, expressed as two phases, can be completed by its deadline. We propose another scheduler called gDMA to further increase the amount of overlap. Unlike gPREM where the processor is stalled while loading from main memory, gDMA utilize a DMA component to load from main memory while the processor is busy executing another task. Hence, the memory phases can be further overlapped with computation phases of tasks on the same core as well as tasks on other cores. We compare both gPREM and gDMA against contention-based execution where tasks access memory without control. The results indicate a good improvement in terms of schedulability, the number of schedulable task sets. Both gPREM and gDMA are proposed for sequential tasks in which parallel execution for one task is not permitted. However, there are real-time applications that need more than one core to execute. These are parallel tasks that can be seen as a collection of sequential subtasks.

Federated scheduling has been proposed to execute such parallel tasks for real-time systems. In this scheduling scheme, parallel tasks with processor utilization exceeding one are assigned a dedicated number of cores. Even though they do not interfere at the core level, they interfere at the memory level. We propose a method to assign each parallel task a number of cores and memory bandwidth in order to meet its demand. We initially assume each parallel task is executed on its private cores using a greedy work-conserving scheduler. Then, we propose mthPREM, a static scheduler that aims to reduce the application's makespan, and enhance the memory time by employing the co-scheduling approach at subtask level.

Our research can be extended in many directions. We envision the following relevant research to be carried out in the future.

- Instead of one resource and one memory phase, the real-time task can have multiple phases and more than one resource, e.g., multiple I/O devices.
- The overlap is an important measure to enhance the execution performance. Thus, a new scheduling algorithm can be designed to harness a larger amount of overlap and avoid some of the pessimism in the current analysis.
- The memory time of parallel tasks can be enhanced by allowing shared data to be serviced from a neighbor core instead of main memory. In addition, when sub-tasks that share large data are assigned to the same core, a large content of local memory can be re-used.

References

[1] Ishfaq Ahmad, Yu-Kwong Kwok, and Min-You Wu. Analysis, evaluation, and comparison of algorithms for scheduling task graphs on parallel processors. In *Proc. of International Symposium on Parallel Architectures, Algorithms, and Networks*, pages 207–213. IEEE, 1996.

[2] Benny Akesson, Kees Goossens, and Markus Ringhofer. Predator: a predictable SDRAM memory controller. In *Proc. of International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 251–256. ACM, 2007.

[3] Benny Akesson, Liesbeth Steffens, Eelke Strooisma, and Kees Goossens. Real-time scheduling using credit-controlled static-priority arbitration. In *Proc. of Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 3–14. IEEE, 2008.

[4] Ahmed Alhammad and Rodolfo Pellizzoni. Schedulability analysis of global memory-predictable scheduling. In *Proc. of International Conference on Embedded Software (EMSOFT)*, page 20. ACM, 2014.

[5] Ahmed Alhammad and Rodolfo Pellizzoni. Time-predictable execution of multi-threaded applications on multicore systems. In *Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE)*, page 29. European Design and Automation Association, 2014.

[6] Ahmed Alhammad and Rodolfo Pellizzoni. Trading cores for memory bandwidth in real-time systems. In *Proc. of Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2016.

[7] Ahmed Alhammad, Saud Wasly, and Rodolfo Pellizzoni. Memory efficient global scheduling of real-time tasks. In *Proc. of Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 285–296. IEEE, 2015.

[8] Sebastian Altmeyer, Roeland Douma, Will Lunniss, and Robert I Davis. Evaluation of cache partitioning for hard real-time systems. In *Proc. of Euromicro Conference on Real-Time Systems (ECRTS)*, pages 15–26. IEEE, 2014.

[9] Björn Andersson and Jan Jonsson. The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%. In *Proc. of Euromicro Conference on Real-Time Systems (ECRTS)*, page 33. IEEE, 2003.

[10] Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, et al. Building timing predictable embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4):82, 2014.

[11] 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 *Proc. of Euromicro Conference on Real-Time Systems (ECRTS)*, pages 215–224. IEEE, 2013.

[12] Moshe Bach, Mark Charney, Robert Cohn, Elena Demikhovsky, Tevi Devor, Kim Hazelwood, Aamer Jaleel, Chi-Keung Luk, Gail Lyons, Harish Patil, et al. Analyzing parallel programs with PIN. *Computer*, 43(3):34–41, 2010.

[13] Ke Bai, Jing Lu, Aviral Shrivastava, and Bryce Holton. CMSM: an efficient and effective code management for software managed multicores. In *Proc. of International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–9. IEEE, 2013.

[14] Stanley Bak, Emiliano Betti, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Real-time control of I/O COTS peripherals for embedded systems. In *Proc. of Real-Time Systems Symposium (RTSS)*, pages 193–203. IEEE, 2009.

[15] Stanley Bak, Gang Yao, Rodolfo Pellizzoni, and Marco Caccamo. Memory-aware scheduling of multicore task sets for real-time systems. In *Proc. of International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 300–309. IEEE, 2012.

[16] Theodore P Baker. Multiprocessor EDF and deadline monotonic schedulability analysis. In *Proc. of Real-Time Systems Symposium (RTSS)*, pages 120–129. IEEE, 2003.

[17] Sanjoy Baruah. Techniques for multiprocessor global schedulability analysis. In *Proc. of Real-Time Systems Symposium (RTSS)*, pages 119–128. IEEE, 2007.

[18] Sanjoy Baruah. Improved multiprocessor global schedulability analysis of sporadic DAG task systems. In *Proc. of Euromicro Conference on Real-Time Systems (ECRTS)*, pages 97–105. IEEE, 2014.

[19] Sanjoy Baruah. The federated scheduling of constrained-deadline sporadic DAG task systems. In *Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1323–1328. EDA Consortium, 2015.

[20] Sanjoy Baruah, Marko Bertogna, and Giorgio Buttazzo. *Multiprocessor Scheduling for Real-Time Systems*. Springer, 2015.

[21] Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leen Stougie, and Andreas Wiese. A generalized parallel task model for recurrent real-time processes. In *Proc. of Real-Time Systems Symposium (RTSS)*, pages 63–72. IEEE, 2012.

[22] Sanjoy K Baruah. The non-preemptive scheduling of periodic tasks upon multiprocessors. *Real-Time Systems*, 32(1-2):9–20, 2006.

[23] Andrea Bastoni, Björn Brandenburg, and James Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. *Proc. of Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT)*, pages 33–44, 2010.

[24] L Becchetti, M Dirnberger, A Karrenbauer, and K Mehlhorn. Feasibility analysis in the sporadic DAG task model. In *Proc. of Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2013.

[25] Marko Bertogna and Michele Cirinei. Response-time analysis for globally scheduled symmetric multiprocessor platforms. In *Proc. of Real-Time Systems Symposium (RTSS)*, pages 149–160. IEEE, 2007.

[26] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. Improved schedulability analysis of EDF on multiprocessor platforms. In *Proc. of Euromicro Conference on Real-Time Systems (ECRTS)*, pages 209–218. IEEE, 2005.

[27] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proc. of International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81. ACM, 2008.

[28] Enrico Bini and Giorgio C Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

[29] Silas Boyd-Wickizer, Austin T Clements, Yandong Mao, Aleksey Pesterev, M Frans Kaashoek, Robert Morris, Nickolai Zeldovich, et al. An analysis of linux scalability to many cores. In *Proc. of USENIX Symposium on Operating Systems Design & Implementation*, pages 1–16. USENIX Association, 2010.

[30] Paolo Burgio, Andrea Marongiu, Paolo Valente, and Marko Bertogna. A memory-centric approach to enable timing-predictability within embedded many-core accelerators. In *Proc. of the CSI Symposium on Real-Time and Embedded Systems and Technologies (RTEST)*, pages 1–8. IEEE, 2015.

[31] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011.

[32] Giorgio C Buttazzo, Marko Bertogna, and Gang Yao. Limited preemptive scheduling for real-time systems. a survey. *IEEE Transactions on Industrial Informatics*, 9(1):3–15, 2013.

[33] Paul Caspi and Oded Maler. From control loops to real-time programs. In *Handbook of Networked and Embedded Control Systems*, pages 395–418. Springer, 2005.

[34] Francisco J Cazorla, Eduardo Quiñones, Tullio Vardanega, Liliana Cucu, Benoit Triquet, Guillem Bernat, Emery Berger, Jaume Abella, Franck Wartel, Michael Houston, et al. Proartis: Probabilistically analyzable real-time systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(2s):94, 2013.

[35] Marco Cesati, Renato Mancuso, Emiliano Betti, and Marco Caccamo. A memory access detection methodology for accurate workload characterization. In *Proc. of International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 141–148. IEEE, 2015.

[36] Sudipta Chattopadhyay, Abhik Roychoudhury, and Tulika Mitra. Modeling shared cache and bus in multi-cores for timing analysis. In *Proc. of International Workshop on Software & Compilers for Embedded Systems*, page 6. ACM, 2010.

[37] Sudipta Chattopadhyay, Abhik Roychoudhury, Jakob Rosén, Petru Eles, and Zebo Peng. Time-predictable embedded software on multi-core platforms: Analysis and optimization. *Foundations and Trends in Electronic Design Automation*, 8(3-4):199–356, 2014.

[38] Derek Chiou, Prabhat Jain, Srinivas Devadas, and Larry Rudolph. Dynamic cache partitioning via columnization. In *Proc. of Design Automation Conference (DAC)*. Citeseer, 2000.

[39] 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 *Proc. of Euromicro Conference on Real-Time Systems (ECRTS)*, pages 25–34. IEEE, 2013.

[40] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

[41] Silviu S Craciunas, Christoph M Kirsch, Hannes Payer, Ana Sokolova, Horst Stadler, and Robert Staudinger. A compacting real-time memory management system. In *Proc. of USENIX Annual Technical Conference*, pages 349–362. USENIX Association, 2008.

[42] Alain Darte, Yves Robert, and Frédéric Vivien. *Scheduling and automatic parallelization*. Springer Science & Business Media, 2012.

[43] Dakshina Dasari, Björn Andersson, Vincent Nelis, Stefan M Petters, Arvind Easwaran, and Jinkyu Lee. Response time analysis of COTS-based multicores considering the contention on the shared memory bus. In *Proc. of International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1068–1075. IEEE, 2011.

[44] Dakshina Dasari, Vincent Nelis, and Benny Akesson. A framework for memory contention analysis in multi-core platforms. *Real-Time Systems*, pages 1–51, 2015.

[45] Robert I Davis and Alan Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Systems*, 47(1):1–40, 2011.

[46] Robert I Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys (CSUR)*, 43(4):35, 2011.

[47] Robert I Davis, Alan Burns, José Marinho, Vincent Nelis, Stefan M Petters, and Marko Bertogna. Global fixed priority scheduling with deferred pre-emption. In *Proc of International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–11, 2013.

[48] Robert I Davis, Luca Santinelli, Sebastian Altmeyer, Claire Maiza, and Liliana Cucu-Grosjean. Analysis of probabilistic cache related pre-emption delays. In *Proc. of Euromicro Conference on Real-Time Systems (ECRTS)*, pages 168–179. IEEE, 2013.

[49] Robert I Davis, Abhilash Thekkilakattil, Oliver Gettings, Radu Dobrin, and Sasikumar Punnekkat. Quantifying the exact sub-optimality of non-preemptive scheduling. In *Proc. of Real-Time Systems Symposium (RTSS)*. IEEE, 2015.

[50] J-F Deverge and Isabelle Puaut. WCET-directed dynamic scratchpad memory allocation of data. In *Proc. of Euromicro Conference on Real-Time Systems (ECRTS)*, pages 179–190. IEEE, 2007.

[51] Sudarshan Kumar Dhall. Scheduling periodic-time-critical jobs on single processor and multiprocessor computing systems. *PhD thesis*, 1977.

[52] Angel Dominguez, Sumesh Udayakumaran, and Rajeev Barua. Heap data allocation to scratch-pad memory in embedded systems. *Journal of Embedded Computing*, 1(4):521–540, 2005.

[53] Guy Durrieu, Madeleine Faugere, Sylvain Girbal, Daniel Gracia Pérez, Claire Pagetti, and Wolfgang Puffitsch. Predictable flight management system implementation on a multicore processor. In *Proc. of European Congress on Embedded Real-time Software and Systems (ERTS)*, 2014.

[54] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. *ACM Sigplan Notices*, 45(3):335–346, 2010.

[55] FAA. Position paper on multicore processors, CAST-32 (rev 0). https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-32.pdf. Accessed: 2016-02-26.

[56] Heiko Falk and Jan C Kleinsorge. Optimal static WCET-aware scratchpad allocation of program code. In *Proc. of Annual Design Automation Conference (DAC)*, pages 732–737. ACM, 2009.

[57] Nathan Fisher, Joël Goossens, and Sanjoy Baruah. Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible. *Real-Time Systems*, 45(1-2):26–71, 2010.

[58] Freescale. P4080 QorIQ integrated multicore communication processor family reference manual. Technical report, Freescale Semiconductors, 2011.

[59] Joël Goossens and Vandy Berten. Gang FTP scheduling of periodic and parallel rigid real-time tasks. *arXiv preprint arXiv:1006.2617*, 2010.

[60] Kees Goossens, John Dielissen, and Andreea Radulescu. Æthereal network on chip: concepts, architectures, and implementations. *Design & Test of Computers*, 22(5):414–421, 2005.

[61] Stijn Goossens, Jasper Kuijsten, Benny Akesson, and Kees Goossens. A reconfigurable real-time SDRAM controller for mixed time-criticality systems. In *Proc. of International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pages 1–10. IEEE, 2013.

[62] Giovani Gracioli, Ahmed Alhammad, Renato Mancuso, Antônio Augusto Fröhlich, and Rodolfo Pellizzoni. A survey on cache management mechanisms for real-time embedded systems. *ACM Computing Surveys (CSUR)*, 48(2):32, 2015.

[63] Giovani Gracioli and Antônio Augusto Fröhlich. Towards a shared-data-aware multicore real-time scheduler. In *Proc. of Time Scheduling Open Problems Seminar*, 2013.

[64] Nan Guan, Wang Yi, Qingxu Deng, Zonghua Gu, and Ge Yu. Schedulability analysis for non-preemptive fixed-priority multiprocessor scheduling. *Journal of Systems Architecture*, 57(5):536–546, 2011.

[65] Nan Guan, Wang Yi, Zonghua Gu, Qingxu Deng, and Ge Yu. New schedulability test conditions for non-preemptive scheduling on multiprocessor platforms. In *Proc. of Real-Time Systems Symposium (RTSS)*, pages 137–146. IEEE, 2008.

[66] Damien Hardy, Thomas Piquet, and Isabelle Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *Proc. of Real-Time Systems Symposium (RTSS)*, pages 68–77. IEEE, 2009.

[67] Damien Hardy and Isabelle Puaut. Estimation of cache related migration delays for multi-core processors with shared instruction caches. In *Proc. of International Conference on Real-Time Networks and Systems (RTNS)*, pages 45–54, 2009.

[68] Mohamed Hassan, Hiren Patel, and Rodolfo Pellizzoni. A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems. In *Proc. of Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 307–316. IEEE, 2015.

[69] Prathap Kumar Valsan Heechul Yun, Rodolfo Pellizzoni. Parallelism-aware memory interference delay analysis for COTS multicore systems. In *Proc. of Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2015.

[70] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[71] Intel. *IA-32 Architectures Software Developers Manual*, volume 3ª edition, 2011.

[72] Haoqiang Jin, Michael Frumkin, and Jerry Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical report, NAS-99-011, NASA Ames Research Center, 1999.

[73] Ron Kalla, Balaram Sinharoy, and Joel M Tendler. IBM Power5 chip: A dual-core multithreaded processor. *Micro*, 24(2):40–47, 2004.

[74] Shinpei Kato and Yutaka Ishikawa. Gang EDF scheduling of parallel task systems. In *Proc. of Real-Time Systems Symposium (RTSS)*, pages 459–468. IEEE, 2009.

[75] Hyoseung Kim, Dionisio de Niz, Bjorn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding memory interference delay in COTS-based multi-core systems. In *Proc. of Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–154. IEEE, 2014.

[76] Yooseong Kim, David Broman, Jian Cai, and Aviral Shrivastaval. WCET-aware dynamic code management on scratchpads for software-managed multicores. In *Proc. of Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014.

[77] Leonidas Kosmidis, Jaume Abella, Eduardo Quinones, and Francisco J Cazorla. A cache design for probabilistically analysable real-time systems. In *Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 513–518. IEEE, 2013.

[78] Leonidas Kosmidis, Jaume Abella, Eduardo Quinones, and Francisco J Cazorla. Multi-level unified caches for probabilistically time analysable real-time systems. In *Proc. of Real-Time Systems Symposium (RTSS)*, pages 360–371. IEEE, 2013.

[79] A Kurdila, M Nechyba, R Prazenica, W Dahmen, P Binev, R DeVore, and R Sharpley. Vision-based control of micro-air-vehicles: progress and problems in estimation. In *Proc. of Conference on Decision and Control (CDC)*, pages 1635–1642. IEEE, 2004.

[80] Karthik Lakshmanan, Shinpei Kato, and Ragunathan Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *Proc. of Real-Time Systems Symposium (RTSS)*, pages 259–268. IEEE, 2010.

[81] Doug Lea. A Java fork/join framework. In *Proc. of Conference on Java Grande*, pages 36–43. ACM, 2000.

[82] Hennadiy Leontyev and James H Anderson. A unified hard/soft real-time schedulability test for global EDF multiprocessor scheduling. In *Proc. of Real-Time Systems Symposium (RTSS)*, pages 375–384. IEEE, 2008.

[83] Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. Analysis of global EDF for parallel tasks. In *Proc. of Euromicro Conference on Real-Time Systems (ECRTS)*, pages 3–13. IEEE, 2013.

[84] Jing Li, Jian Jia Chen, Kunal Agrawal, Chenyang Lu, Chris Gill, and Abusayeed Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *Proc. of Euromicro Conference on Real-Time Systems (ECRTS)*, pages 85–96. IEEE, 2014.

[85] Yun Liang, Huping Ding, Tulika Mitra, Abhik Roychoudhury, Yan Li, and Vivy Suhendra. Timing analysis of concurrent programs running on shared cache multicores. *Real-Time Systems*, 48(6):638–680, 2012.

[86] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D Patel, Stephen A Edwards, and Edward A Lee. Predictable programming on a precision timed architecture. In *Proc. of International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 137–146. ACM, 2008.

[87] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[88] Fang Liu, Xiaowei Jiang, and Yan Solihin. Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance. In *Proc.*

of International Symposium on High Performance Computer Architecture (HPCA), pages 1–12. IEEE, 2010.

[89] 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.

[90] Jing Lu, Ke Bai, and Aviral Shrivastava. SSDM: smart stack data management for software managed multicores (SMMs). In *Proc. of Annual Design Automation Conference (DAC)*, page 149. ACM, 2013.

[91] Lars Lundberg. Multiprocessor scheduling of age constraint processes. In *Proc. of International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 42–47. IEEE, 1998.

[92] TR Maeurer and D Shippy. Introduction to the Cell multiprocessor. *IBM journal of Research and Development*, 49(4):589–604, 2005.

[93] Claudio Maia, Marko Bertogna, Luís Nogueira, and Luis Miguel Pinho. Response-time analysis of synchronous parallel tasks in multiprocessor systems. In *Proc. of International Conference on Real-Time Networks and Systems (RTNS)*, page 3. ACM, 2014.

[94] Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. Real-time cache management framework for multi-core architectures. In *Proc. of Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–54. IEEE, 2013.

[95] Renato Mancuso, Roman Dudko, and Marco Caccamo. Light-PREM: Automated software refactoring for predictable execution on COTS embedded systems. In *Proc. of International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA))*, pages 1–10. IEEE, 2014.

[96] Jaydeep Marathe and Frank Mueller. Source-code-correlated cache coherence characterization of OpenMP benchmarks. *IEEE Transactions on Parallel and Distributed Systems*, 18(6):818–834, 2007.

[97] José Marinho, Vincent Nélis, Stefan M Petters, Marko Bertogna, and Robert I Davis. Limited pre-emptive global fixed task priority. In *Proc. of Real-Time Systems Symposium (RTSS)*, pages 182–191. IEEE, 2013.

[98] Andrea Marongiu and Luca Benini. An OpenMP compiler for efficient use of distributed scratchpad memory in MPSoCs. *IEEE Transactions on Computers*, 61(2):222–236, 2012.

[99] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio Buttazzo. Memory-processor co-scheduling in fixed priority systems. In *Proc. of International Conference on Real Time and Networks Systems (RTNS)*, pages 87–96. ACM, 2015.

[100] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio C Buttazzo. Response-time analysis of conditional DAG tasks in multiprocessor systems. In *Proc. of Euromicro Conference on Real-Time Systems (ECRTS)*, pages 211–221. IEEE, 2015.

[101] Joseph Musmanno. Data intensive systems (DIS) benchmark performance summary. Technical report, DTIC Document, 2003.

[102] Geoffrey Nelissen, Vandy Berten, Joël Goossens, and Dragomir Milojevic. Techniques optimizing the number of processors to schedule multi-threaded tasks. In *Proc. of Euromicro Conference on Real-Time Systems (ECRTS)*, pages 321–330. IEEE, 2012.

[103] Kyle J Nesbit, Nidhi Aggarwal, James Laudon, and James E Smith. Fair queuing memory systems. In *Proc. of MICRO*, pages 208–222. IEEE, 2006.

[104] Jan Nowotsch, Michael Paulitsch, Arne Henrichsen, Pongratz Werner, and Schacht Andreas. Monitoring and WCET analysis in COTS multi-core-SoC-based mixed-criticality systems. In *Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2014.

[105] Takeshi Ogasawara. An algorithm with constant execution time for dynamic storage allocation. In *Proc. of International Workshop on Real-Time Computing Systems and Applications*, pages 21–25. IEEE, 1995.

[106] Marco Paolieri, Eduardo Quiñones, Francisco J Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for WCET analysis of hard real-time multicore systems. *ACM SIGARCH Computer Architecture News*, 37(3):57–68, 2009.

[107] Marco Paolieri, Eduardo Quiñones, Francisco J Cazorla, and Mateo Valero. An analyzable memory controller for hard real-time CMPs. *Embedded Systems Letters*, 1(4):86–90, 2009.

[108] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A predictable execution model for COTS-based embedded systems. In *Proc. of Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 269–279. IEEE, 2011.

[109] Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. Worst case delay analysis for memory interference in multicore systems. In *Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 741–746. IEEE, 2010.

[110] Bo Peng, Nathan Fisher, and Marko Bertogna. Explicit preemption placement for real-time conditional code. In *Proc. of Euromicro Conference on Real-Time Systems (ECRTS)*, pages 177–188. IEEE, 2014.

[111] Jason Poovey, Markus Levy, Shay Gal-On, Thomas M Conte, et al. A benchmark characterization of the EEMBC benchmark suite. *IEEE Micro*, 29(5):18–29, 2009.

[112] Isabelle Puaut and Christophe Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2007.

[113] Arthur Pyka, Mathias Rohde, and Sascha Uhrig. A real-time capable coherent data cache for multicores. *Concurrency and Computation: Practice and Experience*, 26(6):1342–1354, 2014.

[114] Arthur Pyka, Mathias Rohde, Pavel G Zaykov, and Sascha Uhrig. Case study: On-demand coherent cache for avionic applications. In *Proc. of Workshop on High-performance and Real-time Embedded Systems*, 2014.

[115] Eduardo Quinones, Emery D Berger, Guillem Bernat, and Francisco J Cazorla. Using randomized caches in probabilistic real-time systems. In *Proc. of Euromicro Conference on Real-Time Systems (ECRTS)*, pages 129–138. IEEE, 2009.

[116] P. Radojković, V. Čakarević, M. Moretó, J. Verdú, A. Pajuelo, F.J. Cazorla, M. Nemirovsky, and M. Valero. Optimal task assignment in multithreaded processors: a statistical approach. In *Proc. of Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 235–248, 2012.

[117] Andrei Radulescu, John Dielissen, Santiago González Pestana, Om Prakash Gangwal, Edwin Rijpkema, Paul Wielage, and Kees Goossens. An efficient on-chip NI offering guaranteed services, shared-memory abstraction, and flexible network configuration.

IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 24(1):4–17, 2005.

[118] Jan Reineke, Isaac Liu, Hiren D Patel, Sungjun Kim, and Edward A Lee. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *Proc. of International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 99–108. ACM, 2011.

[119] Tabish Rohan, Mancuso Renato, Wasly Saud, Alhammad Ahmed, S. Phatak Sujit, Pellizzoni Rodolfo, and Caccamo Marco. A real-time scratchpad-centric OS for multi-core embedded systems. In *Proc. of Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2016.

[120] Jakob Rosen, Alexandru Andrei, Petru Eles, and Zebo Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Proc. of Real-Time Systems Symposium (RTSS)*, pages 49–60. IEEE, 2007.

[121] A. Saifullah, K. Agrawal, Chenyang Lu, and C. Gill. Multi-core real-time scheduling for generalized parallel task models. In *Proc. of Real-Time Systems Symposium (RTSS)*, pages 217–226. IEEE, Nov 2011.

[122] 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.

[123] Peter Sanders and Jochen Speck. Energy efficient frequency scaling and scheduling for malleable tasks. In *Euro-Par 2012 Parallel Processing*, pages 167–178. Springer, 2012.

[124] Abhik Sarkar, Frank Mueller, Harini Ramaprasad, and Sibin Mohan. Push-assisted migration of real-time tasks in multi-core processors. *ACM Sigplan Notices*, 44(7):80–89, 2009.

[125] Simon Schliecker, Mircea Negrean, and Rolf Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 759–764. European Design and Automation Association, 2010.

[126] Martin Schoeberl. Time-predictable chip-multiprocessor design. In *Proc. of Conference on Signals, Systems and Computers*, pages 2116–2120. IEEE, 2010.

[127] Andreas Schranzhofer, Jian-Jia Chen, and Lothar Thiele. Timing analysis for TDMA arbitration in resource sharing systems. In *Proc. of Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 215–224. IEEE, 2010.

[128] Oliver Sinnen and Leonel Sousa. Experimental evaluation of task scheduling accuracy: Implications for the scheduling model. *IEICE Transactions on Information and Systems*, 86(9):1620–1627, 2003.

[129] Oliver Sinnen, Leonel Sousa, et al. Communication contention in task scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):503–515, 2005.

[130] Mladen Slijepcevic, Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones, and Francisco J Cazorla. Time-analysable non-partitioned shared caches for real-time multicore systems. In *Proc. of Annual Design Automation Conference (DAC)*, pages 1–6. IEEE, 2014.

[131] Vivy Suhendra and Tulika Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *Proc. of Design Automation Conference (DAC)*, pages 300–303. ACM, 2008.

[132] Vivy Suhendra, Abhik Roychoudhury, and Tulika Mitra. Scratchpad allocation for concurrent embedded software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(4):13, 2010.

[133] Abhilash Thekkilakattil, Robert I Davis, Radu Dobrin, Sasikumar Punnekkat, and Marko Bertogna. Multiprocessor fixed priority scheduling with limited preemptions. In *Proc. of International Conference on Real Time and Networks Systems (RTNS)*, pages 13–22. ACM, 2015.

[134] Sumesh Udayakumaran, Angel Dominguez, and Rajeev Barua. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Transactions on Embedded Computing Systems (TECS)*, 5(2):472–511, 2006.

[135] Bryan C Ward, Jonathan L Herman, Christopher J Kenna, and James H Anderson. Making shared caches more predictable on multicore platforms. In *Proc. of Euromicro Conference on Real-Time Systems (ECRTS)*, pages 157–167. IEEE, 2013.

[136] Saud Wasly and Rodolfo Pellizzoni. A dynamic scratchpad memory unit for predictable real-time embedded systems. In *Proc. of Euromicro Conference on Real-Time Systems (ECRTS)*, pages 183–192. IEEE, 2013.

[137] Jack Whitham and Neil C Audsley. Explicit reservation of local memory in a predictable, preemptive multitasking real-time system. In *Proc. of Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 3–12. IEEE, 2012.

[138] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008.

[139] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966–978, 2009.

[140] Zheng Pei Wu, Yogen Krish, and Rodolfo Pellizzoni. Worst case analysis of DRAM latency in multi-requestor systems. In *Proc. of Real-Time Systems Symposium (RTSS)*, pages 372–383. IEEE, 2013.

[141] Jun Yan and Wei Zhang. WCET analysis for multi-core processors with shared L2 instruction caches. In *Proc. of Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 80–89. IEEE, 2008.

[142] G. Yao, H. Yun, Z.P. Wu, R. Pellizzoni, M. Caccamo, and L. Sha. Schedulability analysis for memory bandwidth regulated multicore real-time systems. *IEEE Transactions on Computers*, 65(99):1–1, 2015.

[143] Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Emiliano Betti, and Marco Caccamo. Memory-centric scheduling for multicore hard real-time systems. *Real-Time Systems*, 48(6):681–715, 2012.

[144] Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Heechul Yun, and Marco Caccamo. Global real-time memory-centric scheduling for multicore systems. *IEEE Transactions on Computers*, 2015.

[145] Man-Ki Yoon, Jung-Eun Kim, and Lui Sha. Optimizing tunable WCET with shared resource allocation and arbitration in hard real-time multicore systems. In *Proc. of Real-Time Systems Symposium (RTSS)*, pages 227–238. IEEE, 2011.

[146] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in

multi-core platforms. In *Proc. of Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64. IEEE, 2013.

[147] Wei Zhang and Yan Jun. Static timing analysis of shared caches for multicore processors. *Journal of Computing Science and Engineering*, 6(4):267–278, 2012.

[148] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. *ACM SIGARCH Computer Architecture News*, 38(1):129–142, 2010.

[149] Michael Zimmer, David Broman, Chris Shaver, and Edward Lee. FlexPRET: A processor platform for mixed-criticality systems. In *Proc. of Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2014.