공학박사 학위논문

System-wide Time vs. Density Tradeoff in Real-Time Multicore Fluid Scheduling

실시간 멀티코어 플루이드 스케줄링에서 전체 시스템의 시간 · 밀도 트레이드오프

2017년 8월

서울대학교 대학원

전기 · 컴퓨터공학부

김 강 욱

Abstract

System-wide Time vs. Density Tradeoff in Real-Time Multicore Fluid Scheduling

Kang-Wook Kim
Department of Electrical Engineering and Computer Science
The Graduate School
Seoul National University

Recent parallel programming frameworks such as OpenCL and OpenMP allow us to enjoy the parallelization freedom for real-time tasks. The parallelization freedom creates the time vs. density tradeoff problem in fluid scheduling, i.e., more parallelization reduces thread execution times but increases the density. By system-widely exercising this tradeoff, this dissertation proposes a parameter tuning of real-time tasks aiming at maximizing the schedulability of multicore fluid scheduling. The experimental study by both simulation and actual implementation shows that the proposed approach well balances the time and the density, and results in up to 80% improvement of the schedulability.

keywords : real-time, multi-core scheduling, fluid scheduling, time vs. density tradeoff

Student Number : 2009-20755

Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Motivation and Objective

The advent of parallel programming, such as OpenCL [43] and OpenMP [20], allows us to parallelize a real-time task in many different ways, which we call parallelization freedom. As an example, Fig. 1.1 shows the execution times of threads according to the number of threads that the program is parallelized into. The key observation in this example is that the execution time of each thread is reduced when the number of parallel threads is increased while the sum of the execution times of threads is increased due to the parallelization overhead. In other words, more parallelization reduces the finish time of the program but, as the tradeoff, requires more total computation amount, and vice versa.

Due to this parallelization freedom, we can take a tradeoff between "time" and "density". That is, if we parallelize a task into a larger number of threads, the "time" for executing each thread becomes shorter. However, due to the parallelization overhead, the total computation amount of the task, i.e., the sum of thread execution times, becomes larger, which in turn makes the task's "density" larger, where the task density is defined as the task's total computation amount divided by its deadline.

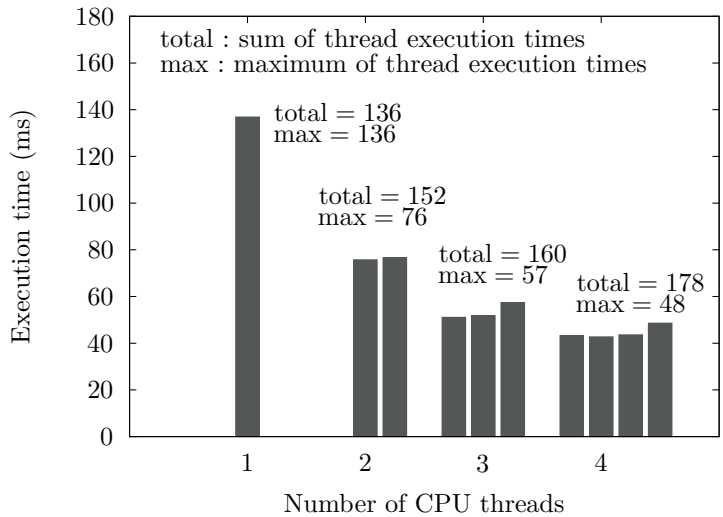


Figure 1.1 Measured thread execution times for an edge-detection program

Regarding this time vs. density tradeoff, there exist the "time bound" and the "density bound".

- **Time bound:** We have to parallelize a task into a large enough number of threads such that each thread execution time becomes shorter than the given deadline. Otherwise, the deadline cannot be met in any way since each parallelized thread should be sequentially executed. Thus, the given deadline can be considered as an upper bound for the thread execution time.

- **Density bound:** A well known theory says that the ideal fluid scheduling [7] or its practical variations [16, 25, 38, 47, 60] can schedule all the threads of all the tasks before their deadlines if the system density, i.e.,

the sum of task densities, is smaller than or equal to the number of cores at any time. Thus, the number of cores can be considered as an upper bound below which the system density must be kept at all the times.

This dissertation tries to system-widely enjoy this tradeoff to maximize the schedulability of the multicore fluid scheduling [7, 16, 25, 38, 47, 60] with multi-segment tasks. For this, the proposed approach is to minimize the peak system density under the time bound constraints by tuning four parameters of every task: (1) artificial period, (2) artificial deadline, (3) offset, and (4) parallelization.

1.2 Approach

Since optimally determining these four parameters of all the tasks is a pretty complex combinatorial problem, this dissertation takes a four-step approach. First, an optimal tradeoff of time and density for each individual single segment task is proposed. In this step, the artificial deadline and parallelization of a task are optimally determined [1].

Secondly, for a group of single segment tasks with the same period, an optimal parameter tuning for minimizing the peak density of the group is proposed. This step proposes how to decide task offset, artificial deadline, and parallelization when given group of tasks have same period.

Thirdly, by repeatedly using these optimal per-task and per-group tradeoff

[1] Since this step only considers each task independently, the artificial period and offset are not determined.

methods, this step proposes a near optimal algorithm for the entire task set to determine the parameters of all the tasks such that the peak system density can be reduced as much as possible.

Finally, extension from the system-wide tradeoff between time and density for single segment tasks to a system-wide tradeoff between time and density for multi-segment tasks is introduced. Based on the system-wide time vs. density tradeoff for single segment tasks, this step proposes how to extend to the multi-segment tasks and shows its appropriateness.

1.3 Organization

The rest of this dissertation is organized as follows:

- Chapter 2 surveys related work which includes the real-time workload model, real-time multicore scheduling, and parallel computing framework.

- Chapter 3 proposes system-wide time vs. density tradeoff of single segment task. This chapter introduces system model, per-task optimization (first step in Section 1.2), group optimization (second step in Section 1.2), and heuristic algorithm for system-wide time vs. density tradeoff (third step in Section 1.2).

- Chapter 4 extends single segment task model in Chapter 3 to multi-segment task model (fourth step in Section 1.2).

- Chapter 5 finally concludes this dissertation and discusses the limitation and the future work.

Chapter 2
Related Work

2.1 Real-Time Scheduling

2.1.1 Workload Model

There have been numerous researches on real-time multicore scheduling since Dhall and Liu first addressed it in 1978 [21]. [7] first presents an optimal algorithm for scheduling periodic sequential tasks on multicore resources. The algorithm is theoretically optimal, in terms of the competitiveness in schedulability, but it is impractical due to its heavy scheduling, preemption, and migration overheads. Thereafter, many researchers propose optimal multicore scheduling algorithms for sequential tasks with less overheads [16, 60].

Along with the advancement in parallel programming, the workload model has been extended from sequential to parallel models. [4] and [33] present scheduling algorithms for "multi-thread task model" where each task is parallelized into a fixed number of threads from the beginning to the end of its execution. Then, [37], [24] and [35] present scheduling algorithms for "fork-join task model" where each task consists of alternating single-thread segments and multi-thread segments, that is, a single-thread forks into multi-threads in the next segment and then they join into a single-thread and so on. [57], [46]

and [18] generalize this model into "multi-segment task model" where each task consists of a sequence of segments with arbitrary (but fixed) number of threads for each segment. [46] and [56] address the most general task model called "DAG (Directed Acyclic Graph) task model" where each task is modeled as a DAG and the vertexes can be executed simultaneously on multiple CPU cores as long as the precedence constraints among vertexes are satisfied. Most work on the multi-segment task model and the DAG task model, for example, [46, 56, 57], address the problem by assigning intermediate deadlines to segments or vertexes. In contrast, [18] addresses the problem by applying global EDF scheduling without assigning intermediate deadlines.

In all of the aforementioned work for the parallel tasks, it is assumed that the parallelization is pre-fixed, that is, a pre-fixed number of threads for each task in the multi-thread task model, for each segment in the multi-segment task model, and for each vertex in the DAG task model. Unlike this assumption, [52] addresses the case where each vertex in a DAG task has the freedom of parallelization. However, the model in [52] is not practical since the time needed for completing a vertex is simply modeled as the total computation amount divided by the number of threads assuming zero parallelization overhead.

2.1.2 Scheduling on Multicore Systems

Real-time multicore scheduling can be largely classified into partitioned scheduling approaches [5, 6, 15, 24, 41, 58] and global scheduling approaches [7, 8, 16,

18, 25, 33, 36–38, 46, 47, 56, 57, 60–62]. The global scheduling approaches can be further classified as the ones based on global EDF [8, 18, 33, 37, 56, 57, 61] and those based on the fluid scheduling model [4, 7, 16, 25, 36, 38, 46, 47, 60, 62]. P-Fair [7] is the first fluid scheduling algorithm and is known to be optimal for scheduling periodic sequential tasks on multicore resources. The algorithm is theoretically optimal in terms of the competitiveness in schedulability, but it is impractical due to its heavy scheduling, preemption, and migration overheads. Thereafter, many researchers propose more practical fluid scheduling algorithms such as PD^2 [60], LLREF [16], LRE-TL [25], DP-Fair [38], and U-EDF [47] with less overheads but still keeping the optimality. The method we propose in this paper can work with any of those fluid scheduling algorithms.

Along with such enhancements of fluid scheduling, the task model also has been extended from sequential to parallel models, "multi-thread task model" [4], "multi-segment task model" [46] and "DAG (Directed Acyclic Graph) task model" [46]. However, all of these researches assume that the parallelization is pre-fixed without enjoying the parallelization freedom, which is possible in modern parallel programming frameworks such as OpenCL and OpenMP.

Recently, [36] addresses the problem of optimally enjoying the parallelization freedom for the multi-segment task model. It optimally determines the parallelization option for the given deadline of each task such that the task density is minimized. The density of each task is simply added to obtain the peak system density assuming that all the tasks become active at the same

time in the worst case. Thus, it can be understood as conducting the time vs. density tradeoff for each task separately without the system-wide tradeoff by jointly considering all the tasks together.

2.1.3 Period Control

There have been several studies [23, 27, 42, 44, 45, 59] about interesting properties of harmonic tasks. In [27], it is shown that assigning artificial harmonic period smaller than original period improves schedulability for Rate-Monotonic (RM) scheduling policy. The algorithms proposed in [27] are applied for radar systems in [59] and for the multicore system in [23].

The studies in [44, 45] propose how to construct harmonic period when the periods of tasks are given as an acceptable period ranges. In these works, a period range is represented as an interval. By multiplying this interval by integers, the system constructs set of intervals for each task. For the entire tasks, if set of intervals are overlapped, then this overlapped intervals are used as period of each task.

On the other hands, [9–11] propose elastic scheduling which has varying task period. Whenever the total system utilization exceeds one, the period of each task is changed within given range to decrease total system utilization by acting as a linear spring system.

2.1.4 Real-Time Operating System

As one of the first widely used real-time operating system, QNX [28] is a commercial Unix-like real-time operating system. With advent of IEEE 1003.1b [1], QNX became popular. Currently it is used in vehicle systems and mobile phones. Another popular commercial real-time operating system is VxWorks [67]. It is used in avionic systems and Mars Rover project.

Real-Time Application Interface (RTAI) [55] is a community project developing a real-time extension for the Linux kernel. By patching the Linux kernel, RTAI provides interfaces that real-time application can use. It supports three types of scheduler which are uni-processor scheduler, symmetric multi-processor scheduler, and multi-uni-processor scheduler.

LITMUS^{RT} [14] is another real-time extension for the Linux kernel, which focuses on real-time scheduling and synchronization on multicore system. It supports useful features for real-time system research by providing various kinds of scheduler as a plugin module. Currently it provides partitioned EDF, global EDF, clustered EDF, partitioned fixed-priority, partitioned reservation-based scheduling, and PD^2.

2.2 Parallel Computing

2.2.1 Parallel Computing Framework

Open Multi Processing (OpenMP) is one of a parallel computing standard whose first version [49] is published in 1997. In OpenMP programming model,

a programmer should explicitly specify parallelism of a code block using additional directives. With this directives, OpenMP automatically parallelizes the code block into multiple threads. More specifically, a master thread forks multiple threads at the beginning of a parallel code block and waits finish of all threads at the end of the code block. Until version 3.0 [50] released in 2008, OpenMP only supports parallel programming on multiple processors which share the memory. That means it does not support accelerator such as GPGPU which uses separate memory. From version 4.0 [51] released in 2013, directives supporting GPU are added to OpenMP but it still has limitation that it does not support general GPUs.

Before release of OpenMP 4.0, in order to support parallel computing on accelerators, Open Accelerators (OpenACC) [66] was released in 2011. As like OpenMP, OpenACC also directive-based framework, but OpenACC targets for collaborative computing with host processor and accelerators. By limiting the sort of directive operators, OpenACC has more scalability compared to OpenMP, but it has limitation on the kind of supported program.

Another effort to support parallel programming is Open Computing Language (OpenCL) [65]. The OpenCL is a parallel computing framework for heterogeneous platforms such as CPUs, GPUs, FPGAs, and other accelerators. OpenCL program called "kernel" is written with C-like language (OpenCL C). Regarding to the platform a kernel code to be run, OpenCL compiles the kernel code to the binary which can be executed on the platform at either run-time or compile time. This gives great benefit that one kernel code can

be used for multiple heterogeneous platforms. The OpenCL application uses OpenCL APIs to configure target platforms, prepare kernel codes, and send command to run the kernel code on specific platform.

Since OpenCL is a specification of APIs, its implementations for specific platforms are needed. CPU and GPU vendors such as Intel [29], AMD [3], NVIDIA [48] provide OpenCL implementations for their CPU and GPU products. For academic use, SnuCL [34] and pocl [31] are developed for heterogeneous CPU/GPU clusters. In this dissertation, pocl is modified to support real-time OpenCL applications.

2.2.2 Shared Resource Management

In multicore systems, the memory subsystems such as shared cache, memory controller, and prefetching hardware are regarded as shared resources since multiple cores can access those shared resources at the same time. Thus during access to the shared resources, a core can be interfered by other cores which also access same resource. In Commercial Off-The-Shelf (COTS) system, the interference cannot be bounded, which means it causes degradation of entire system throughput and aggravating unfairness of each core. Especially in real-time systems, the system can be fail due to this unexpected and unpredictable interference. In order to overcome this problem, there has been many researches for the shared resource management in multicore system.

One effort is reserving the portion of resource for each core, which called resource reservation technique. Resource reservation techniques [2, 12, 13, 40,

54,70,71] have been widely studied especially in real-time community. The proposed methods in [12,13,40] can be applied to CPU, but they have limitations that it is not possible to apply to memory subsystem. In order to overcome this problem, [70, 71] propose MemGuard which reserves DRAM bandwidth for each core in COTS multicore system. MemGuard counts the number of cache misses of each core to identify the number of DRAM accesses of each core. If the number of DRAM accesses of one core exceeds threshold during specific time window, scheduler immediately evacuating the tasks in the ready queue to guarantee DRAM bandwidth of other cores.

Another effort is isolating shared resource proposed in [17, 19, 22, 26, 30, 32, 39, 53, 63, 64, 68, 69, 72]. For the shared cache, there have been simulation studies [26, 30, 53, 63] for partitioning the cache to prevent the interference from other accesses. The studies in [17, 19, 39, 64] adopt page coloring which partitions cache by allocating corresponding physical page. For the DRAM memory, [22,32,68,69] proposed DRAM bank partitioning. Since DRAM bank itself has local cache called row buffer, interference from other access causes unexpected row miss. Thus, allocating partitioned bank isolates the performance of DRAM memory.

Chapter 3
System-wide Time vs. Density Tradeoff with Parallelizable Periodic Single Segment Tasks

3.1 Introduction

As described in Chapter 1, this chapter addresses the problem of system-wide time vs. density tradeoff for single segment tasks by controlling four parameters, i.e., artificial deadline, artificial period, offset, and parallelization. For this, Section 3.2 introduces the problem description. Section 3.3 gives intuitive examples how system-wide time vs. density tradeoff works by controlling the four parameters. Section 1.2 describes the proposed approaches in this chapter, i.e., per-task optimization, group optimization, and heuristic algorithm. Finally Section 3.5 shows experimental results of both simulation and actual implementation.

3.2 Problem Description

We consider a system with M homogeneous CPU cores and N parallelizable periodic tasks. A parallelizable periodic task is defined as $\tau_i = (P_i, D_i, C_i(O_i))$ as shown in Fig. 3.1, where P_i is the period and D_i is the relative deadline. Also, O_i is the parallelization option indicating that the task τ_i is parallelized

into O_i threads. O_i ranges from 1 to $O^{\max}$ meaning that we have options of parallelizing the task τ_i into single thread, i.e., $O_i = 1$, two threads, i.e., $O_i = 2$, and up to $O^{\max}$ threads, i.e., $O_i = O^{\max}$ as shown in the table of Fig. 3.1 (Fig. 3.1 shows an example case where the parallelization option $O_i = 2$ is chosen). Also, we use $e_i^k(O_i)$ $(1 \leq k \leq O_i)$ to denote the execution time of the k-th thread with the parallelization option of O_i and $C_i(O_i)$ to denote the total computation amount or simply "computation amount" of all O_i threads, i.e., $C_i(O_i) = \sum_{k=1}^{O_i} e_i^k(O_i)$. Out of the O_i thread execution times, the largest one is denoted by $e_i^{\max}(O_i) = \max_{1 \leq k \leq O_i} e_i^k(O_i)$. In general, a larger parallelization option $O'(> O)$ makes the thread execution time smaller, i.e., $e_i^{\max}(O') < e_i^{\max}(O)$ but makes the computation amount larger, i.e., $C_i(O') > C_i(O)$, due to the parallelization overhead [1].

Note that we use this simple task model with a single segment just for the simplicity of explanation. Our approach also works for the multi-segment task model where each task consists of multiple segments with parallelization freedom, which will be explained in Chapter 4.

For scheduling these N parallelizable periodic tasks on M homogeneous CPU cores, we assume the ideal fluid scheduling algorithm, e.g., P-Fair [7] or its practical variations, e.g., PD^2 [60], LLREF [16], LRE-TL [25], DP-Fair [38] and U-EDF [47] that reduce the number of preemptions and migrations while still keeping the optimality. Just like other researches on fluid scheduling [36,

[1] If there is a larger parallelization option that increases both thread execution times and the computation amount, that option does not help the schedulability in any way. Thus, we can simply exclude that option in our consideration.

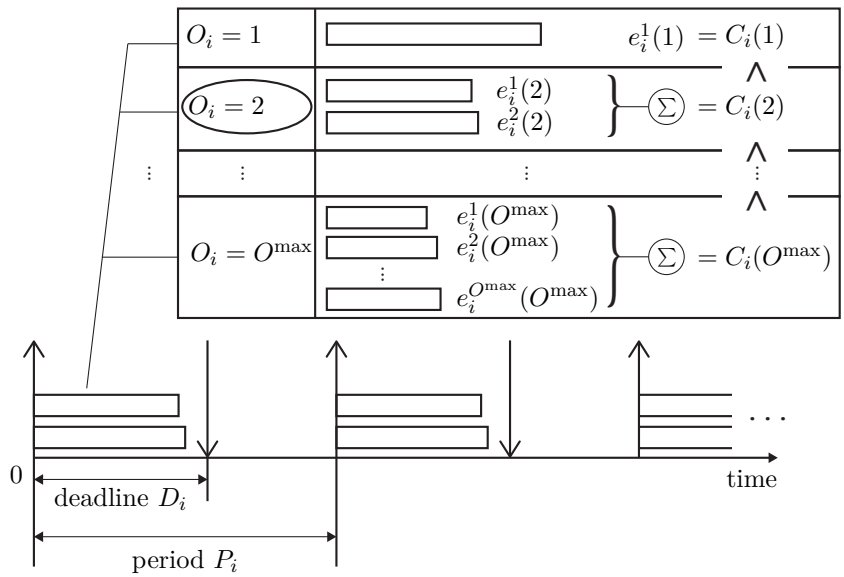


Figure 3.1 Parallelizable periodic task model

46], we assume that the thread executions on CPU cores can be preempted at any time and the cost for preemption and migration is negligible. Also, we assume memory and bandwidth partitioning such as PALLOC [69] and MemGuard [70] to avoid inter-core interferences. Under these assumptions, the given task set is schedulable if the following two conditions are met:

- **Time bound condition**: Each thread execution time of every task should be smaller than or equal to the given deadline, i.e, time bound. Otherwise, the given deadline cannot be met in any way since a thread should be executed sequentially. Fig. 3.2(a) shows an example task with period 15, deadline 10, and the single thread execution time 12. Since the thread should be sequentially executed even on two cores, there is

no way to finish the thread before the given deadline 10.

- **Density bound condition**: In the fluid scheduling [7], a thread is defined to be *active* from its release time to its absolute deadline. Also, the density of an active thread is defined as its execution time divided by its relative deadline, i.e., the absolute deadline minus the release time. Thus, the *system density* at a time is defined as the sum of densities of threads which are active at that time. With these definitions, the system density at all the times should be lower than or equal to the number of cores, i.e., density bound [7]. Otherwise, some threads cannot meet their deadlines by the fluid scheduling. Fig. 3.2(b) shows the same example task as in Fig. 3.2(a) except that the task is parallelized into four threads whose execution times are 5, 5, 6, and 6, respectively. In this case, the system density from the task release time to the task deadline is $22/10 = 2.2$, which is larger than the density bound 2. This means that the computation demand per unit time is 2.2 which is greater than the computation supply 2 by the two cores per unit time. Thus, even the optimal scheduling like the fluid scheduling cannot finish all the four threads before the deadline 10.

For satisfying these time bound and density bound conditions, the parallelization freedom of all the tasks should be carefully exercised. A larger parallelization option O_i for a task τ_i makes each thread execution time shorter. Thus, it is favorable to the time bound condition. However, it makes the total

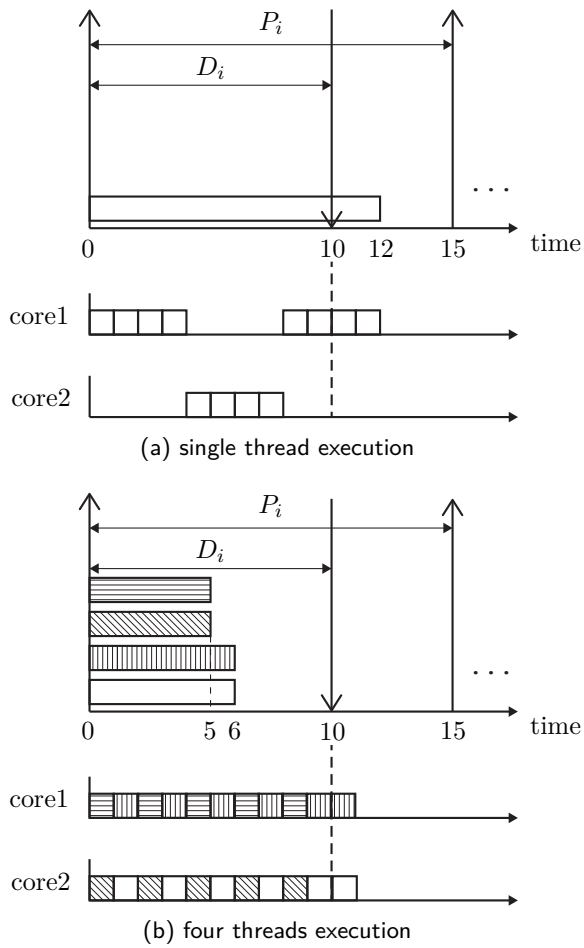


Figure 3.2 Example task with period 15 and deadline 10

computation amount $C_i(O_i)$ larger due to the parallelization overhead and hence the system density larger. Thus, it is unfavorable to the density bound. We call this a time vs. density tradeoff caused by the parallelization freedom.

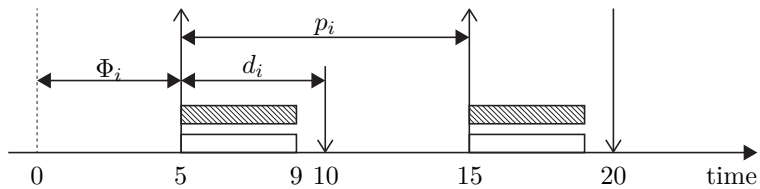
The problem we tackle in this paper is how to system-widely exercise this tradeoff to maximize the schedulability. For this, we have four controllable parameters, i.e., the task offset Φ_i—the first job's release time, the parallelization option O_i, the artificial deadline $d_i(\leq D_i)$, and the artificial period $p_i(\leq P_i)$ for every tasks $\tau_i(1 \leq i \leq N)$ [2]. For a task τ_i, if we use the offset Φ_i, the parallelization option O_i, the artificial deadline d_i, and the artificial period p_i, its density function along time t is formulated as follows:

$$\delta_i((\Phi_i, O_i, d_i, p_i), t) = \begin{cases} C_i(O_i)/d_i & (t - \Phi_i) \bmod p_i \ \leq \ d_i \\ 0 & (t - \Phi_i) \bmod p_i \ > \ d_i \end{cases}$$

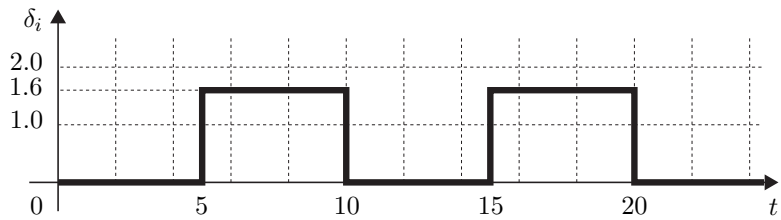
meaning that the task density is $C_i(O_i)/d_i = \sum_{k=1}^{O_i} e_i^k(O_i)/d_i$ while the task is active and zero otherwise. The density function $\delta_i((\Phi_i, O_i, d_i, p_i), t)$ is depicted as the time-density curve in Fig. 3.3 for an example task τ_i whose parameters are tuned as $\Phi_i = 5$, $O_i = 2$, $d_i = 5$, and $p_i = 10$. The task τ_i's active density, i.e., $C_i(O_i)/d_i$, is simply called τ_i's density and denoted by $\delta_i(O_i, d_i)$.

With this density function of each task τ_i, our system-wide time vs. density tradeoff problem can be formulated as the following problem of minimizing

[2] We are targeting control tasks whose performances improve as increasing their sampling and actuation frequencies. Thus, an artificial period p_i ($\leq P_i$) does not violate the control performance requirement provided by the original period P_i.



(a) τ_i with $\Phi_i = 5$, $O_i = 2$, $d_i = 5$, and $p_i = 10$



(b) τ_i's time-density curve

Figure 3.3 Time-density curve example of $\delta_i((5, 2, 5, 10), t)$

the peak of the system density, i.e., the sum of all N task density functions, within the hyper period HP while satisfying the time bound condition for each thread:

$$\text{minimize} \quad \max_{0 \le t < HP} \sum_{i=1}^{N} \delta_i((\Phi_i, O_i, d_i, p_i), t)$$

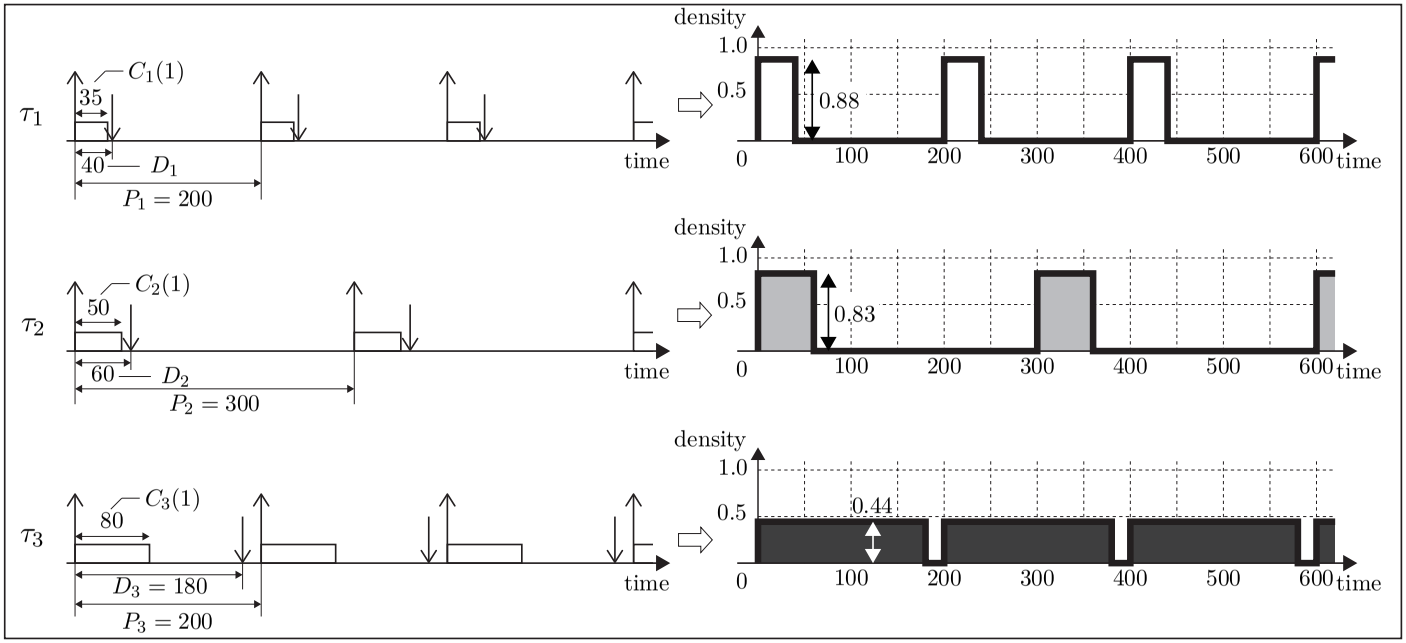
subject to

$$
\begin{aligned}
0 \leq \quad & p_i \quad \leq P_i && (1 \leq i \leq N) \\
0 \leq \quad & \Phi_i \quad < p_i && (1 \leq i \leq N) \\
1 \leq \quad & O_i \quad \leq O^{\max} && (1 \leq i \leq N) \\
& d_i \quad \leq D_i && (1 \leq i \leq N) \\
e_i^{\max}(O_i) \leq \quad & d_i && (1 \leq i \leq N)
\end{aligned}
$$

The first constraint says that the artificial period p_i should be shorter than the original period P_i to ensure the original or a higher task repetition rate. The second constraint says that it is good enough to exercise the offset Φ_i within one period p_i since any offset larger than one period p_i can be represented by its modulo p_i operation. The third constraint naturally comes from the available parallelization options. The fourth constraint says that the artificial deadline should be controlled under the original deadline. Finally, the fifth constraint is the time bound condition saying that each of O_i parallelized threads should have execution times $e_i^k(O_i)$s $(1 \leq k \leq O_i)$ shorter than or equal to the artificial deadline, i.e., time bound, d_i.

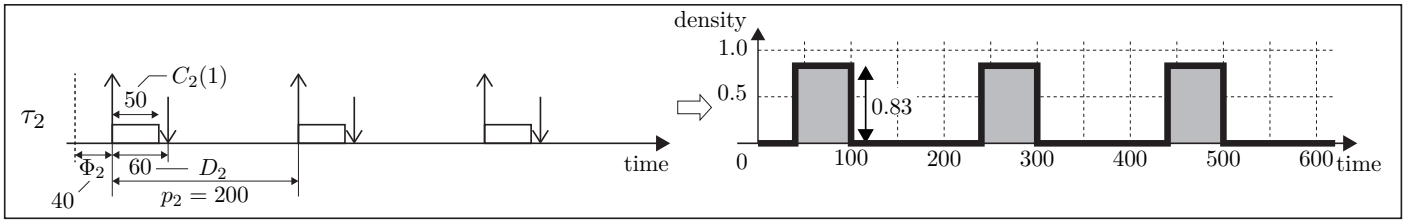
3.3 Motivating Example

As a motivating example, Fig. 3.4(a) shows an example task set with three periodic tasks $\{\tau_1, \tau_2, \tau_3\}$ for the case of using the original period P_i, the original deadline D_i, zero offset, and the single thread option, i.e., $O_i = 1$. For the three tasks, the right side of Fig. 3.4(a) depicts their time-density curves.

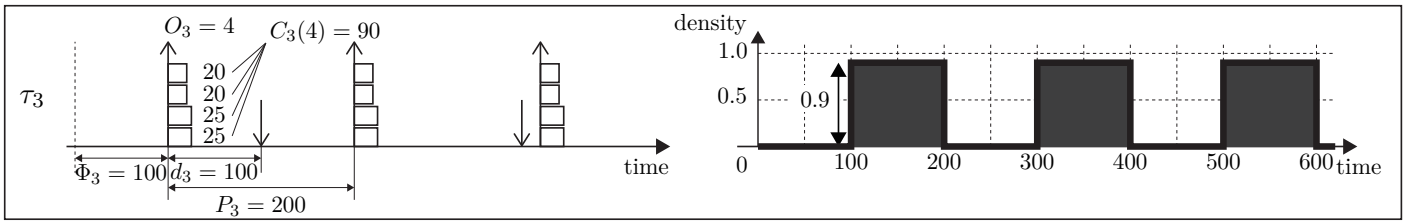


(a) No parameter tuning

Figure 3.4 Motivating example tasks



(b) τ_2's parameters are tuned as $\Phi_2 = 40$ and $p_2 = 200$

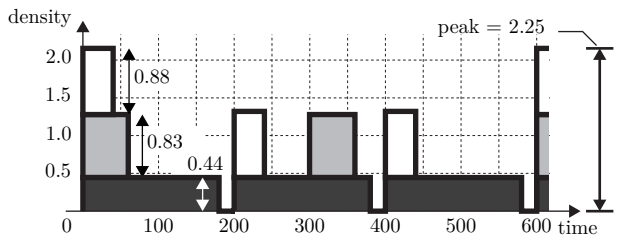


(c) τ_3's parameters are tuned as $O_3 = 4$, $d_3 = 100$, and $\Phi_3 = 100$

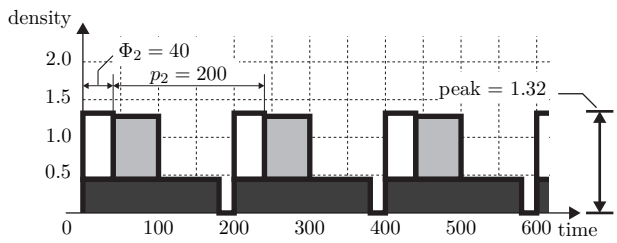
Figure 3.4 Motivating example tasks (cont'd)

The time-density curve of the system density is simply the sum of each task's time-density curve as depicted in Fig. 3.5(a). In this case, the peak system density is 2.25. Thus, for the schedulability of the task set satisfying the density bound condition, we need at least three CPU cores. If we tune task τ_2's parameters such that the offset $\Phi_2 = 40$ and the artificial period $p_2 = 200$ as in Fig. 3.4(b), then the time-density curve of τ_2 is reshaped as in the right-side of Fig. 3.4(b). Thanks to this tuning of τ_2's parameters, the system density curve becomes Fig. 3.5(b) where the peak system density is reduced to 1.32. Now, the task set becomes schedulable with two CPU cores. If we further tune τ_3's parameters such that the parallelization option $O_3 = 4$ (i.e., $C_3(O_3) = 90$), the artificial deadline $d_3 = 100$, and the offset $\Phi_3 = 100$ as in Fig. 3.4(c), the time-density curve of τ_3 is reshaped accordingly. As a result, the system density curve becomes Fig. 3.5(c) where the peak system density is further reduced to 0.9. Thus, the same task set now becomes schedulable only with one core.

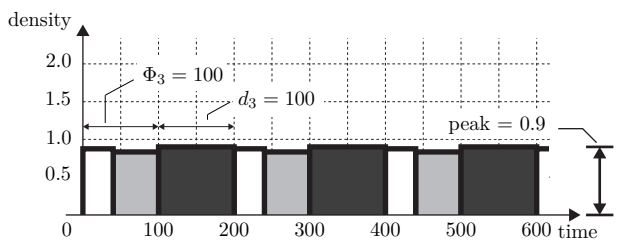
As shown in this motivating example, this paper aims at developing a systematic way to determine all the task parameters with the objective of minimizing the peak system density while satisfying the time bound conditions. By minimizing the peak system density, we can expect the schedulability improvement in the following two senses: (1) when the number of CPU cores is fixed, we can schedule more tasks and (2) when the set of tasks is fixed, we can schedule it with a smaller number of cores.



(a) No parameter tuning



(b) Φ_2 and p_2 are tuned



(c) O_3, d_3, and Φ_3 are tuned

Figure 3.5 System-wide time-density curve

3.4 Proposed Approach

In the aforementioned problem, we have four control knobs for each task τ_i, i.e., the parallelization option O_i, the artificial deadline d_i, the offset Φ_i, and the artificial period p_i. The proposed solution approach to optimally exercising these four control knobs is explained in the following order. Section 3.4.1 describes the per-task optimal tradeoff of time and density, which optimally determines the parallelization option O_i once an artificial deadline d_i is fixed for a task τ_i. Then, Section 3.4.2 describes a method to optimally determine the artificial deadlines d_is (also O_i due to Section 3.4.1) and the offsets Φ_is for a task group with the same period such that the peak density of the group can be minimized. Using Section 3.4.1 and Section 3.4.2 as building block methods, Section 3.4.3 explains a heuristic algorithm that finds a near-optimal solution, i.e., (Φ_i, O_i, d_i, p_i)s for every task $\tau_i \in \{\tau_1, \tau_2, \cdots, \tau_N\}$, that minimizes the peak system density.

3.4.1 Per-task Optimal Tradeoff of Time and Density

In the per-task tradeoff for individually minimizing τ_i's density, i.e., $\delta(O_i, d_i) = C_i(O_i)/d_i$, O_i can be optimally determined once d_i is fixed. More specifically, for the given d_i, we need to choose a large enough parallelization option O_i to satisfy the time bound condition, i.e., $e_i^{\max}(O_i) \leq d_i$. Lemma 3.1 says that out of the parallelization options that satisfy the time bound condition,the smallest one is the optimal O_i for the given d_i.

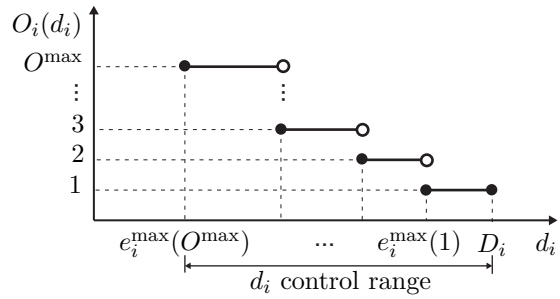
Lemma 3.1. *For a given d_i, the smallest O_i that satisfies $e_i^{max}(O_i) \leq d_i$ is the optimal O_i for minimizing τ_i's density $\delta(O_i, d_i) = C_i(O_i)/d_i$.*

Proof. Once d_i is fixed, the difference of task density $C_i(O_i)/d_i$ comes only from the computation amount $C_i(O_i)$. Since $C_i(O_i)$ becomes larger as increasing the parallelization option O_i, the smallest parallelization option O_i that satisfies $e_i^{\max}(O_i) \leq d_i$ gives the smallest $C_i(O_i)$ and hence minimizes $\delta(O_i, d_i) = C_i(O_i)/d_i$ satisfying $e_i^{\max}(O_i) \leq d_i$. □

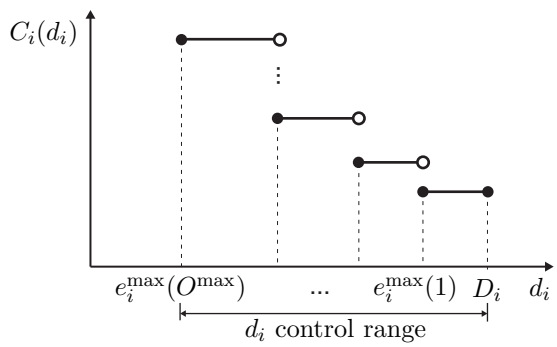
Thanks to Lemma 3.1, the parallelization option O_i can be represented as a function of the artificial deadline d_i without loss of optimality. Such a function is denoted by $O_i(d_i)$ and depicted in Fig. 3.6(a). Also, the optimal computation amount $C_i(O_i(d_i))$ and the optimal task density $\delta_i(O_i(d_i), d_i) = C_i(O_i(d_i))/d_i$ can also be represented as functions of d_i. Those functions are denoted by $C_i(d_i)$ and $\delta_i(d_i)$, and they are depicted in Figs. 3.6(b) and (c), respectively.

3.4.2 Peak Density Minimization for a Task Group with the Same Period

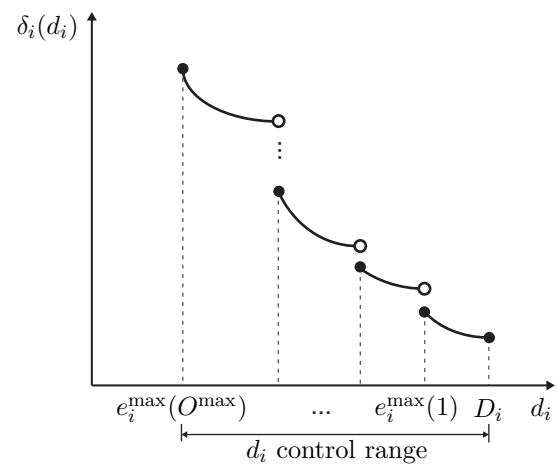
Since the parallelization option O_i is optimally represented as a function of d_i thanks to the per-task tradeoff, in the system-wide tradeoff of time and density, the remaining control knobs are the artificial deadline d_i, the task offset Φ_i, and the artificial period p_i. In this subsection, we explain how to optimally determine d_i and Φ_i for a group of tasks with the same period P to minimize the peak density of the group called the "peak group density".



(a) Optimal $O_i(d_i)$



(b) Optimal $C_i(d_i)$



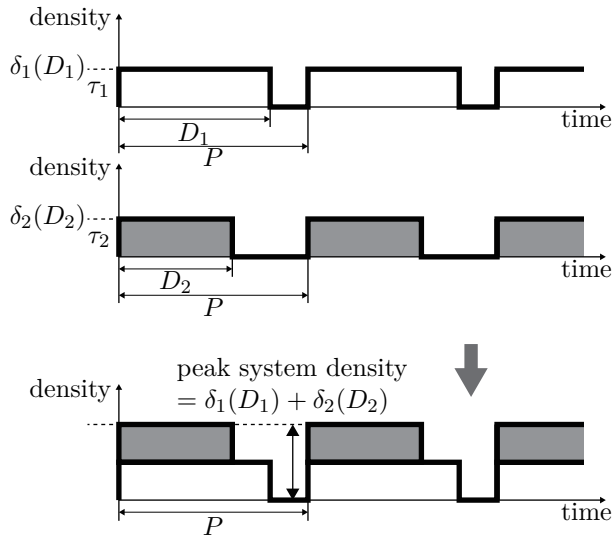
(c) Optimal $\delta_i(d_i)$

Figure 3.6 Per-task optimal $O_i(d_i)$, $C_i(d_i)$ and $\delta_i(d_i)$

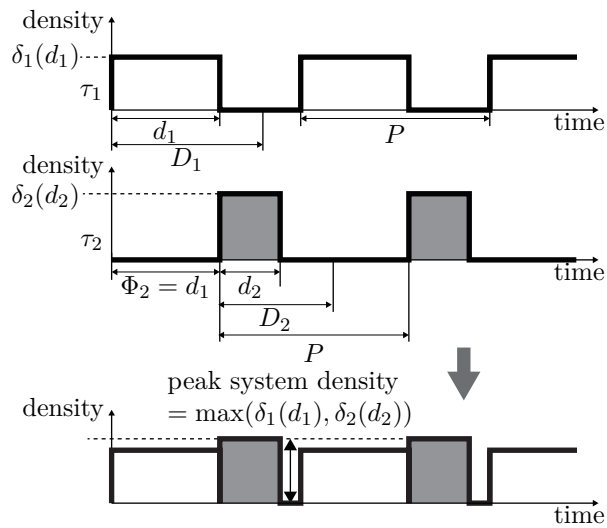
For this, we consider so called "horizontal placement" that disjointly places the tasks' active intervals in the time window of P. In order to give the intuition on how the horizontal placement works by controlling d_i and Φ_i, Fig. 3.7 shows an example group of two tasks τ_1 and τ_2 with the same period P. Fig. 3.7(a) depicts the time-density curves of τ_1 and τ_2 with original deadlines D_1 and D_2 and zero offsets. In this case, the active intervals of τ_1 and τ_2 overlap and hence the peak group density is the sum of two task densities as shown in Fig. 3.7(a). On the other hand, if we shrink the artificial deadlines of τ_1 and τ_2, that is, from D_1 to d_1 and from D_2 to d_2 as in Fig. 3.7(b), it is possible to disjointly place the active intervals of τ_1 and τ_2 within one period P by controlling the task offset, i.e., $\Phi_2 = d_1$. The reduced artificial deadlines d_1 and d_2 give increased task densities, i.e., $\delta_1(d_1) > \delta_1(D_1)$ and $\delta_2(d_2) > \delta_2(D_2)$. However, the peak group density may be smaller since it is not the sum of the two task densities, i.e., $\delta_1(D_1) + \delta_2(D_2)$, but the maximum of them, i.e., $\max(\delta_1(d_1), \delta_2(d_2))$.

The rest of this subsection explains how to find the optimal horizontal placement, that is, the one that minimizes the peak group density out of all possible horizontal placement solutions for a task group consisting of n tasks with the same period P. The tasks in the group are denoted by $\{\tau_1, \tau_2, \cdots, \tau_n\}$.

In order to disjointly place the active intervals in the time window P, Φ_is



(a) Schedule with original deadlines



(b) Schedule with artificial deadlines and adjusted offsets

Figure 3.7 Example of horizontal placement

can be represented with d_is without loss of optimality as follows:

$$\begin{aligned}
\Phi_1 &= 0, \\
\Phi_2 &= d_1, \\
\Phi_3 &= d_1 + d_2, \\
&\vdots \\
\Phi_i &= d_1 + d_2 + \cdots + d_{i-1}, \\
&\vdots \\
\Phi_n &= d_1 + d_2 + \cdots + d_{n-1}.
\end{aligned} \tag{3.1}$$

Also, the sum of d_is should be less than or equal to P, i.e., $\sum_{i=1}^{n} d_i \leq P$, so that the n tasks' active intervals can be placed within one period P.

Now, the optimal horizontal placement problem becomes a problem of determining only one control knob d_i for each task $\tau_i \in \{\tau_1, \tau_2, \cdots, \tau_n\}$ such that the peak group density, i.e., $\max(\delta_1(d_i), \delta_2(d_2), \cdots, \delta_n(d_n))$, can be minimized under the constraint of $\sum_{i=1}^{n} d_i \leq P$.

Horizontal placement problem

$$\begin{aligned}
\text{minimize} \quad & \max_{1 \leq i \leq n} \delta_i(d_i) \\
\text{subject to} \quad & \sum_{i=1}^{n} d_i \leq P.
\end{aligned} \tag{3.2}$$

When solving this problem, we have difficulty that $\delta_i(d_i)$ is a discontinuous

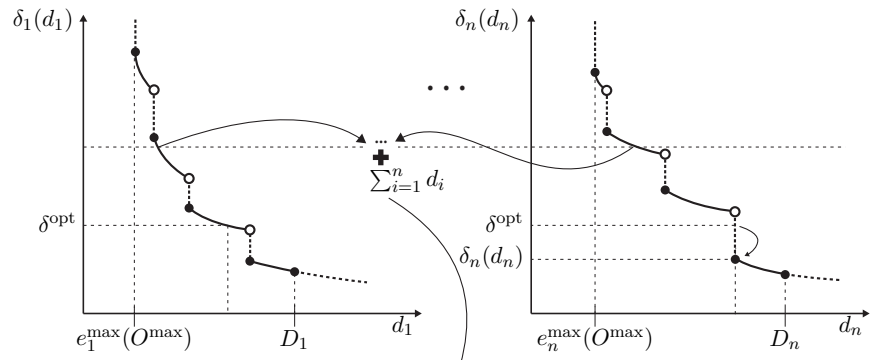
function defined on a limited range of d_i, i.e., $[e_i^{\max}(O^{\max}), D_i]$, as shown in Fig. 3.6(c). In order to overcome this difficulty of discontinuity, we connect discontinuous parts with vertical line segments as depicted as dotted vertical line segments in Figs. 3.8(a) and (b). Also, we extend the range of d_i beyond D_i as depicted as dotted curves in the range of $d_i > D_i$ in Figs. 3.8(a) and (b). By this extension, we now have a continuous relation called a $(d_i, \delta_i(d_i))$-relation for each task τ_i. In this continuous relation, any $(d_i, \delta_i(d_i))$ pair on a solid curve is a real one while any pair on a dotted part is an imaginary one.

With this continuous $(d_i, \delta_i(d_i))$-relation for each task τ_i, the following lemma says that we can find an optimal solution for the horizontal placement problem, by considering only cases where the densities of all the tasks are the same, i.e., uniform densities,

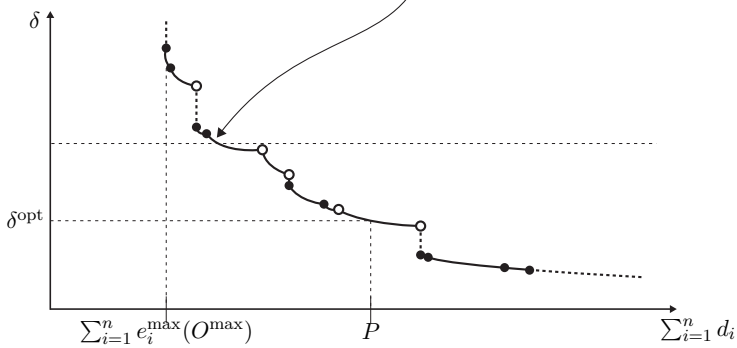
$$\delta_1(d_1) = \delta_2(d_2) = \cdots = \delta_n(d_n).$$

Lemma 3.2. *For the above horizontal placement problem with the continuous $(d_i, \delta_i(d_i))$-relations, consider a solution $\delta_i(d_i)$ $(1 \leq i \leq n)$ with non-uniform densities (see Fig. 3.9(a)). Its peak group density, i.e., $\max_{1 \leq i \leq n} \delta_i(d_i)$, is denoted by δ. For such a solution, there exist another solution with uniform densities that are all equal to δ (see Fig. 3.9(b)).*

Proof. From a non-uniform density solution as in Fig. 3.9(a), for each task τ_i whose density is smaller than the peak density δ (τ_1 and τ_3 in Fig. 3.9(a)), we can increase its density $\delta_i(d_i)$ to δ by decreasing d_i or keeping it the same

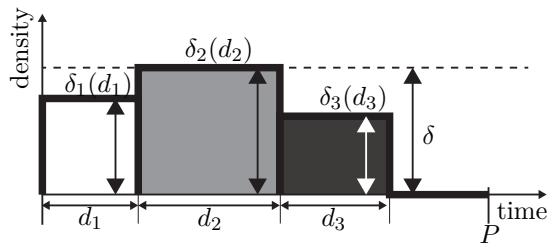


(a) Continuous $(d_1, \delta_1(d_1))$-relation for τ_1 (b) Continuous $(d_n, \delta_n(d_n))$-relation for τ_n

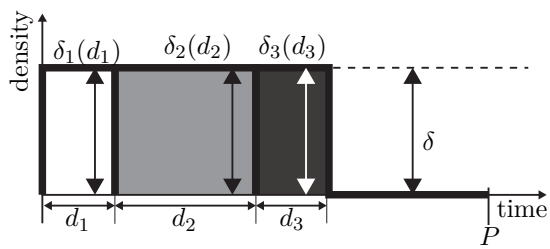


(c) Continuous $(\sum_{i=1}^{n} d_i, \delta)$-relation

Figure 3.8 Continuous $(d_i, \delta_i(d_i))$-relations for τ_is and their continuous $(\sum_{i=1}^{n} d_i, \delta)$-relation



(a) A solution with non-uniform densities



(b) A solution with uniform densities

Figure 3.9 Solutions for horizontal placement problem

along the continuous curves in Figs. 3.8(a) and (b). For such transformed solution with uniform densities of δ, the deadlines d_is after transformation are never greater than those before transformation. Thus, the transformed solution still satisfies the constraint $\sum_{i=1}^{n} d_i \leq P$ of the horizontal placement problem. Therefore, the transformed solution with uniform densities is also a valid solution. □

Thanks to Lemma 3.2, when we solve the horizontal placement problem with the continuous $(d_i, \delta_i(d_i))$-relations, it is enough to consider only cases of $\delta_1(d_1) = \delta_2(d_2) = \cdots = \delta_n(d_n) = \delta$ and find the smallest δ. Thus, without loss of optimality, for each density value δ, its corresponding d_i values in the $(d_i, \delta_i(d_i))$-relations $(1 \leq i \leq n)$ can be summed up as in Figs. 3.8(a) and (b)

resulting in the continuous $(\sum_{i=1}^{n} d_i, \delta)$-relation as in Fig. 3.8(c). This relation guides us how to find the smallest δ by controlling $\sum_{i=1}^{n} d_i$. The following lemma says that the smallest δ can be found when $\sum_{i=1}^{n} d_i = P$.

Lemma 3.3. *The smallest uniform density denoted by δ^{opt}, that is, an optimal solution for the horizontal placement problem with the continuous $(d_i, \delta_i(d_i))$-relations, can be found when $\sum_{i=1}^{n} d_i = P$.*

Proof. The uniform density δ in the $(\sum_{i=1}^{n} d_i, \delta)$-relation (see Fig. 3.8(c)) decreases as increasing $\sum_{i=1}^{n} d_i$. Thus, δ is the smallest when $\sum_{i=1}^{n} d_i$ is the largest, that is, when $\sum_{i=1}^{n} d_i = P$ under the constraint of the horizontal placement problem, i.e., Eq. (3.2). □

Thanks to Lemma 3.3, we can find the optimal uniform density δ^{opt} by finding the cross point between the $(\sum_{i=1}^{n} d_i, \delta)$-relation and the vertical line at P as in Fig. 3.8(c). If the vertical line at P meets with a dotted vertical line segment of the $(\sum_{i=1}^{n} d_i, \delta)$-relation, the bottom most cross point is the optimal solution. If P is too short, that is, $P < \sum_{i=1}^{n} e_i^{\max}(O^{\max})$, the vertical line at P does not meet with the $(\sum_{i=1}^{n} d_i, \delta)$-relation. This is the case where the horizontal placement is not possible since even the minimum possible artificial deadlines $e_i^{\max}(O^{\max})$s for all n tasks cannot be fit into such a short period P.

The δ^{opt} value found in this way is the optimal uniform density for the horizontal placement problem with the continuous domain of $(d_i, \delta_i(d_i))$ for every task τ_i $(1 \leq i \leq n)$. Now, we have to map δ^{opt} to the original discontinuous function $\delta_i(d_i)$ depicted as the solid curve segments in Figs. 3.8(a) and (b).

For this, we draw a horizontal line at δ^{opt} on each $(d_i, \delta_i(d_i))$-relation graph as in Figs. 3.8(a) and (b) and find the cross point. There are three cases for the cross point:

- Case 1: If the cross point is on a solid curve segment as the τ_1 case, i.e., Fig. 3.8(a), it is a real pair of $(d_i, \delta_i(d_i))$. This means that the discrete parallelization option $O_i(d_i)$ determined by the per-task optimal tradeoff in Section 3.4.1 indeed makes the task density $\delta_i(d_i) = C_i(d_i)/d_i$ equal to δ^{opt} found in the continuous domain.

- Case 2: If the cross point is on the dotted vertical line segment as the τ_n case, i.e., Fig. 3.8(b), it is an imaginary pair of $(d_i, \delta_i(d_i))$, that is $\delta_i(d_i)$ for the d_i of the cross point does not actually exist due to discrete parallelization option O_i. The real $\delta_i(d_i) = C_i(d_i)/d_i$ for the found d_i is the bottom most point of the dotted vertical line segment. In this case, $\delta_i(d_i)$ is smaller than δ^{opt}.

- Case 3: If the cross point is on the rightmost dotted curve segment, i.e., $d_i > D_i$, the found d_i is not valid. Thus, we use D_i and its corresponding $O_i(D_i)$ as the solution. In this case, $\delta_i(D_i)$ is greater than δ^{opt}.

Theorem 3.1 says that the solution found by the above procedure is an optimal solution for the original discrete problem of horizontal placement.

Theorem 3.1. *The solution Φ_i, O_i, d_i $(1 \leq i \leq n)$ found by the above procedure is an optimal solution for disjointly placing the active intervals of the n*

tasks with the same period P such that the peak group density, i.e, $\max_{1 \le i \le n} \delta_i(d_i)$, is minimized.

Proof. Due to Lemma 3.1, once we determine the artificial deadline d_i for τ_i, we can optimally determine its parallelization option $O_i(d_i)$ and its corresponding density $\delta_i(d_i)$. Also, Φ_is in Eq. (3.1) ensure no-overlapping of active intervals without loss of optimality if $\sum_{i=1}^{n} d_i \le P$. Thus, the remaining problem is to determine $d_1, d_2, \cdots, d_n$. Due to Lemma 3.2 and Lemma 3.3, δ^{opt}, i.e, the smallest possible uniform density when $\sum_{i=1}^{n} d_i = P$, is indeed the optimal peak group density found with the continuous $(d_i, \delta_i(d_i))$-relations. The mapping from the imaginary continuous domain to the real discontinuous domain makes each task's density $\delta_i(d_i)$ stay the same as δ^{opt}, i.e., Case 1, or become smaller, i.e., Case 2. If the mapping makes all tasks' densities smaller, the peak group density becomes smaller than δ^{opt}, which is not possible because the optimal solution found in the discontinuous domain cannot be better than the one found in the continuous solution space, i.e., a superset of the discontinuous domain. Thus, there exists at least one task whose density stays the same as δ^{opt} after mapping. Therefore, the final solution by the discrete mapping also gives the same peak group density as the optimal one found in the continuous domain. One exception happens when there exists any task τ_i whose cross point found in the range of $d_i > D_i$, i.e., Case 3. If this case happens, we use D_i and its corresponding $\delta_i(D_i)$ which is greater than the continuous optimal uniform density δ^{opt}. Even in this case, we still do not lose the optimality because $C_i(D_i)/D_i$ is the minimum possible density for

τ_i whatsoever and hence no other solution can have the peak group density smaller than $C_i(D_i)/D_i$. □

3.4.3 Heuristic Algorithm for System-wide Time vs. Density Tradeoff

Using the per-task optimal tradeoff of time and density in Section 3.4.1 and the optimal horizontal placement for a task group with the same period in Section 3.4.2, this section proposes a heuristic algorithm for a system-wide tradeoff of time and density which tries to minimizes the peak system density while satisfying the time bound conditions of all the tasks in the entire task set. Intuitively speaking, the algorithm repeats the period harmonization and the horizontal placement in order to eventually partition the given set of tasks, i.e, $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_N\}$, into several harmonic period groups, i.e., $\Gamma[1], \Gamma[2], \cdots$, where each group has horizontally placed tasks with harmonic periods. It is more specifically described as follows:

Heuristic Algorithm: For each task τ_i in the given task set $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_N\}$, find the artificial period $p_i(\leq P_i)$, the offset Φ_i, the artificial deadline $d_i(\leq D_i)$, and the parallelization option O_i such that the peak system density can be maintained as low as possible while satisfying the time bound conditions of all the tasks.

Input: Given set of tasks, i.e., $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_N\}$.

Output: Partitioned harmonic period groups of horizontally placed tasks, i.e., $\Gamma[1], \Gamma[2], \cdots$.

begin procedure

1. initialized $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_N\}$
2. initialized harmonic period group index $x = 0$
3. **while** $(\Gamma \neq \emptyset)$
4. **PeriodHarmonization**(Γ), $n = |\Gamma|$

   // $\tau_i \in \Gamma$ is sorted in ascending order of artificial period p_i
5. $x = x + 1$, $\Gamma[x] = \emptyset$, $\delta^{\text{peak}}[x] = 0$
6. **for** $i = 1$ **to** n
7. $\delta^{\text{peak}}_{\text{new}}$ = **HorizontalPlacement** $(\Gamma[x] \cup \tau_i)$
8. **if** $(\delta^{\text{peak}}_{\text{new}} \leq \delta^{\text{peak}}[x] + C_i(D_i)/D_i)$
9. $\Gamma = \Gamma - \{\tau_i\}$, $\Gamma[x] = \Gamma[x] \cup \{\tau_i\}$
10. $\delta^{\text{peak}}[x] = \delta^{\text{peak}}_{\text{new}}$
11. **end if**
12. **end for**
13. finalize $\Gamma[x]$
14. recover the original periods of tasks left in Γ
15. **end while**

end procedure

Let us explain this algorithm together with an example in Fig. 3.10 with five tasks. In the algorithm, Γ denotes the set of tasks that are not included in any harmonic period group yet. $\Gamma[x]$ denotes the set of tasks in the x-th harmonic period group. Lines 1 and 2 in the algorithm initialize Γ and the

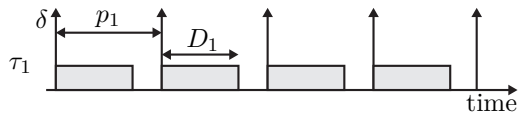
group index x. In Fig. 3.10, Γ is initialized as $\{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5\}$. The **while** loop from Line 3 to Line 15 iteratively adds tasks into groups until no task is left in Γ. Within the **while** loop, Line 4 harmonizes the periods of tasks in Γ. For the period harmonization, we use the method of [44] [3] that minimizes the harmonization penalty with the period control range of $[e_i^1(1), P_i]$ for each τ_i. Fig. 3.10 shows such harmonized periods p_1, p_2, p_3, p_4, p_5 for $\{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5\}$ where we use the task index as the ascending order of the harmonized period, i.e., if $i < j$ then $p_i \leq p_j$. Then, Line 5 makes the first harmonic period group $\Gamma[x{=}1]$ as the empty set. Since there is no task in $\Gamma[1]$ yet, its peak group density $\delta^{\text{peak}}[1]$ is initialized as zero.

After that, the **for** loop from Line 6 to 12 iteratively checks if it is beneficial to add each task τ_i into the current group $\Gamma[x]$ using the horizontal placement. Let us explain this **for** loop assuming that, in the first iteration of the loop, τ_1 is already added into the current group $\Gamma[1]$ as in Fig. 3.10(a). When we consider τ_2 in Fig. 3.10(b), Line 7 tries to perform the horizontal placement of Section 3.4.2 with the tasks already in the current group, i.e., $\Gamma[1] = \{\tau_1\}$ and the newly considered task, i.e., τ_2. Then, Line 8 checks if the horizontal placement of τ_i (e.g., τ_2 in Fig. 3.10(b)) together with the current group $\Gamma[x]$ (e.g., $\Gamma[1] = \{\tau_1\}$ in Fig. 3.10(b)) is not worse than simply stacking up τ_i's density. In the former case marked by ① in Fig. 3.10(b), the peak group density is given as the return value of **HorizontalPlacement** in Line 7 and it is denoted by $\delta_{\text{new}}^{\text{peak}}$. In the latter case marked by Ⓢ in Fig. 3.10(b), the

[3] Appendix A describes more detail about period harmonization.

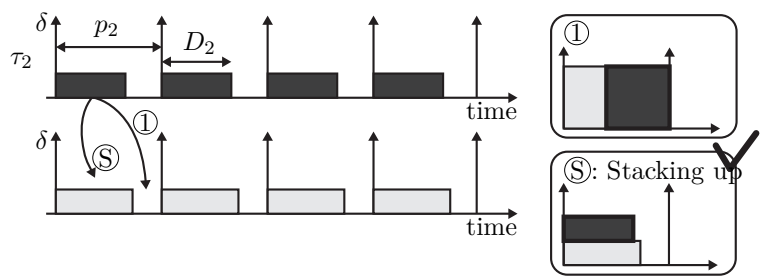
initial $\Gamma = \{\tau_1,\ \tau_2,\ \tau_3,\ \tau_4,\ \tau_5\}$
$\Gamma[1] = \emptyset$

$\Gamma = \{\tau_2,\ \tau_3,\ \tau_4,\ \tau_5\}$
$\Gamma[1] = \{\tau_1\}$



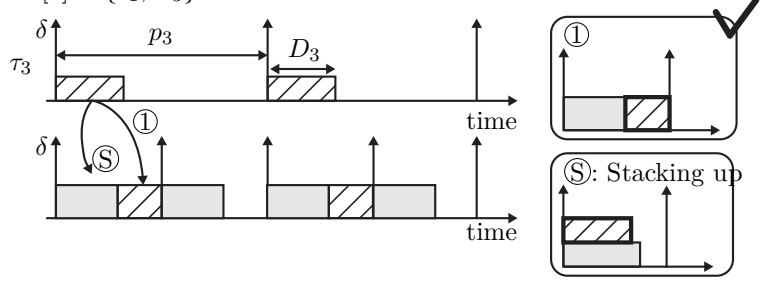
(a) Considering τ_1

$\Gamma = \{\tau_2,\ \tau_3,\ \tau_4,\ \tau_5\}$
$\Gamma[1] = \{\tau_1\}$



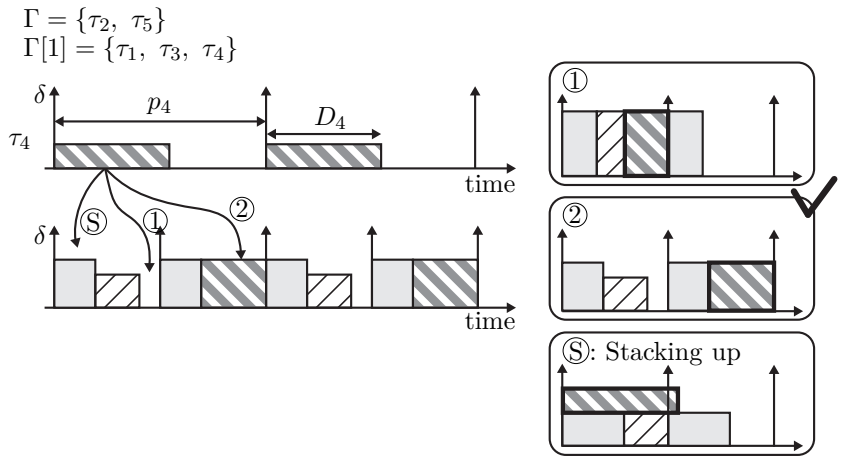
(b) Considering τ_2

$\Gamma = \{\tau_2,\ \tau_4,\ \tau_5\}$
$\Gamma[1] = \{\tau_1,\ \tau_3\}$

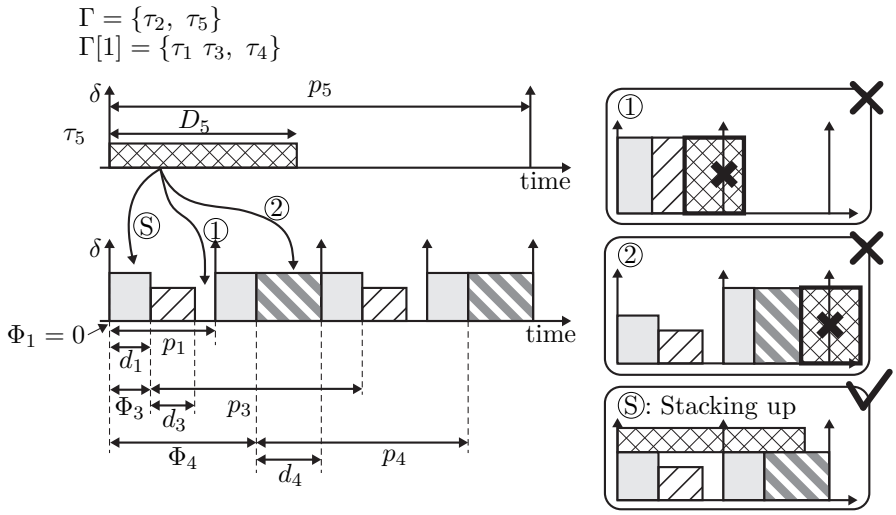


(c) Considering τ_3

Figure 3.10 Example of the heuristic algorithm for the system-wide time vs. density tradeoff



(d) Considering τ_4



(e) Considering τ_5

Figure 3.10 Example of the heuristic algorithm for the system-wide time vs. density tradeoff (cont'd)

resulting peak group density is given by the peak group density of the current $\Gamma[x]$, i.e., $\delta^{\text{peak}}[x]$, plus τ_i's original density, i.e., $C_i(D_i)/D_i$. If the horizontal placement is not worse than the stacking up, then τ_i is extracted from Γ and added into the current group $\Gamma[x]$. In the example of Fig. 3.10(b), let's assume that the horizontal placement is worse, i.e., the **if** condition in Line 8 is not true. Then, τ_2 is not added into $\Gamma[1]$ and remains in Γ as in Fig. 3.10(b).

When we consider τ_3 in Fig. 3.10(c), p_3 is two times of p_1. Thus, we have two p_1 intervals for the horizontal placement of τ_1 and τ_3. However, these two intervals have no difference. Thus, we perform the horizontal placement with the first p_1 interval as shown in Fig. 3.10(c) and add τ_3 to $\Gamma[1]$ because the horizontal placement is better. When we consider τ_4 in Fig. 3.10(d), we also have two p_1 intervals. Thus, **HorizontalPlacement** in Line 7 performs the horizontal placement for both intervals, i.e., (1) horizontal placement of $\{\tau_1, \tau_3, \tau_4\}$ in the first interval as marked by ① and (2) horizontal placement of $\{\tau_1, \tau_4\}$ in the second interval as marked by ②. In this example, horizontally placing τ_4 into the second interval has a smaller peak group density of $\Gamma[1]$ than placing τ_4 into the first interval. Thus, **HorizontalPlacement** in Line 7 places τ_4 in the second interval as in Fig. 3.10(d). In general, when we consider τ_i whose p_i is Z times of p_1, **HorizontalPlacement** in Line 7 tries all Z intervals together with already placed tasks in those intervals and chooses the best one in terms of minimizing the peak group density of group $\Gamma[x]$. Also, in Fig. 3.10(d), note that placing τ_4 in the second interval may result in d_1 that is different from the already calculated one. In that case, we use the smallest

value as shown in the figure. This way, horizontal placement is still valid in all the intervals and the peak group density is not affected. When we consider τ_5 in Fig. 3.10(e), **HorizontalPlacement** in Line 7 considers four intervals since $p_5/p_1 = 4$. Let us assume that the horizontal placements in the four intervals are all infeasible[4]. Then, **HorizontalPlacement** returns ∞ and hence τ_5 is not added into $\Gamma[1]$.

When the **for** loop terminates at Line 13, we can finalize $\Gamma[1]$ with three tasks, i.e., τ_1, τ_3, and τ_4, and their parameters, p_is, Φ_is, and d_is can be fixed as in Fig. 3.10(e). Also, Their corresponding O_is can be fixed by the $O_i(d_i)$ functions due to the per-task optimal tradeoff in Section 3.4.1.

At Line 14, for the example of Fig 3.10(e), τ_2 and τ_5 are left in Γ. Their periods are recovered to the original periods P_2 and P_5 so that they can be considered for the second group $\Gamma[2]$ in the next iteration of the **while** loop.

Once no task is left in Γ, **while** loop terminates resulting in x groups. In the worst case, the peak group densities of all the x groups can overlap. Thus, the peak system density is the sum of peak group densities of the x groups. If this peak system density is not greater than the number of CPU cores, i.e., M, then all the parallelized threads of all N tasks can be scheduled meeting deadlines with a fluid scheduler.

[4]Fig. 3.10(e) shows the horizontal placement only for the first two intervals because the other two are the same with the first two in this example.

3.5 Experimental Results

In this section, we justify the proposed method by both simulation and actual implementation.

3.5.1 Simulation Study

Firstly, we conduct simulation studies to investigate the schedulability improvement by the proposed algorithm. For this, the task set Γ is formed with N synthetic single-segment tasks, where the number of tasks N is uniform-randomly picked from [3, 15]. For each task τ_i in Γ, its period P_i is randomly generated following the uniform distribution from [200ms, 1000ms]. Also, its thread execution time $e_i^1(1)$ for the single thread option $O_i = 1$ is given using a ratio of P_i. The ratio is randomly generated using the normal distribution with the average of 0.5 and standard deviation of 0.1. Thus, $e_i^1(1)$ is randomly determined as the randomly generated ratio multiplied by P_i. For this single thread option, recall that the computation amount $C_i(1)$ is equal to the thread execution time $e_i^1(1)$. Then, the thread execution time $e_i^k(O_i)$ for the parallelization option O_i is determined using a so called "parallelization overhead factor" α. Specifically, all the thread execution times, i.e., $e_i^1(O_i), e_i^2(O_i), \cdots, e_i^{O_i}(O_i)$, are assumed to be the same as $e_i^k(O_i) = e_i^1(1)(\alpha + (1-\alpha)/O_i)$. With this model, when $\alpha = 0$, that is, when there is no parallelization overhead, $e_i^k(O_i) = e_i^1(1)/O_i$. On the contrary, when $\alpha = 1$, that is, when the parallelization overhead is the maximum, $e_i^k(O_i)$

is equal to $e_i^1(1)$, that is, there is no execution time reduction by the parallelization. With this model, $e_i^k(O_i)$ is determined using a uniform-randomly generated α in the range of $[0, 0.1]$. The deadline D_i of τ_i is determined using a so called "deadline factor" β as $D_i = \beta \cdot P_i$. Such generated N tasks in Γ are scheduled using M homogeneous CPU cores, where $M = 8$ if not otherwise mentioned.

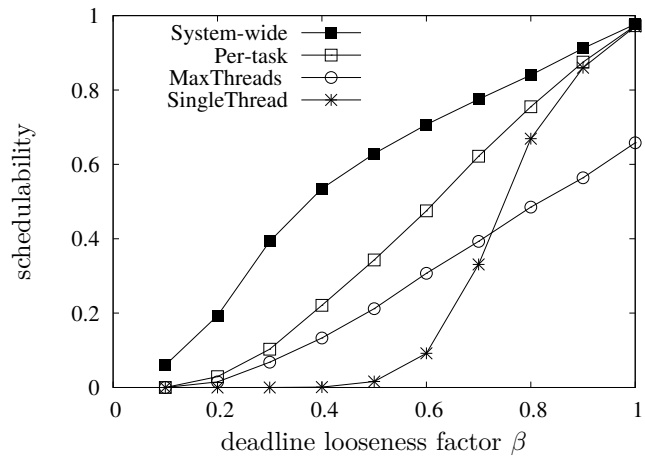
In order to show the schedulability improvement, we compare our approach marked as "System-wide" with the following three approaches.

- No parallelization: Every task is executed with a single thread without parallelization. In this case, the peak system density of Γ is $\sum_{i=1}^{N} e_i^1(1)/D_i = \sum_{i=1}^{N} C_i(1)/D_i$. This approach is marked as "SingleThread".

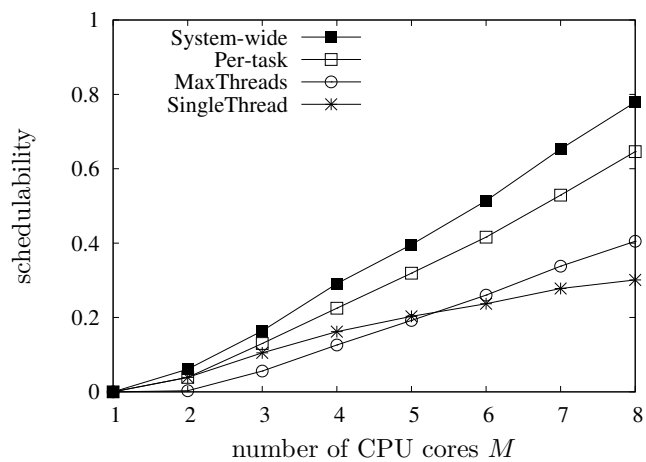
- Max parallelization: Every task is parallelized into the maximum number of threads, i.e., $O^{\max}$. In this case, the peak system density of Γ is $\sum_{i=1}^{N} C_i(O^{\max})/D_i$. It is marked as "MaxThreads".

- Per-task time vs. density tradeoff: It is the approach in [36] where every task uses the optimal $O_i(D_i)$ for the given D_i by the per-task time vs. density tradeoff. Since there is no system-wide consideration, each task's active interval overlaps all together in the worst case and hence the peak system density of Γ is $\sum_{i=1}^{N} C_i(O_i(D_i))/D_i$. It is marked as "Per-task".

Fig. 3.11(a) shows the schedulability of "System-wide" and the above three approaches as ranging the deadline factor $\beta = D_i/P_i$ from 0.1 to 1. For the schedulability measure, we consider 1000 task sets and count how many of



(a) Schedulability according to the deadline looseness factor β



(b) Schedulability according to the number of CPU cores M

Figure 3.11 Schedulability according to the deadline looseness factor and number of CPU cores

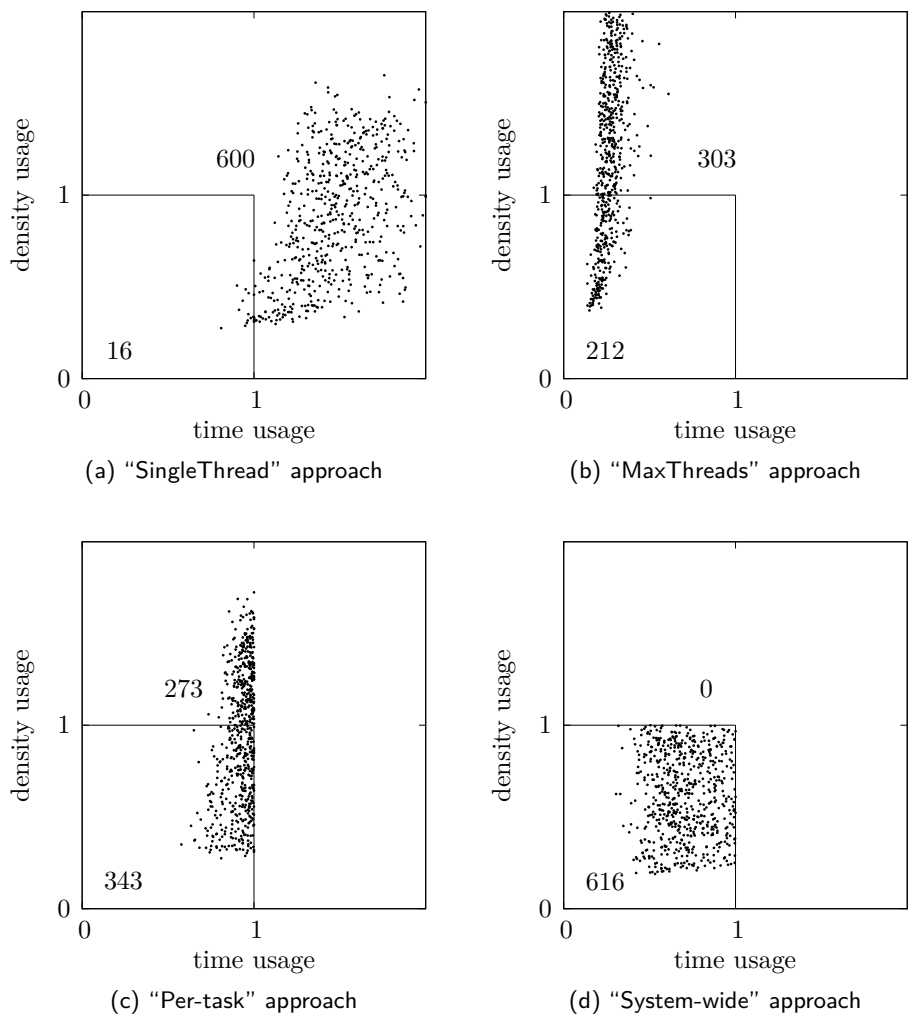
them are schedulable by each approach. Ranging the deadline factor β can be understood as ranging the looseness of time bound. When β is small, that is, when the time bound is tight, "SingleThread" cannot schedule most of task sets. This is because the thread execution time $e_i^1(1)$ overflows the time bound D_i most of times even when the peak system density is smaller than the density bound $M = 8$. On the other hand, "MaxThreads" can schedule more task sets since the thread execution times using the maximum parallelism can be smaller than the time bound D_i. However, as increasing β, the situation reverses. When the time bound is loose, the time bound is not bottleneck but the density bound $M = 8$ becomes bottleneck. Thus, "SingleThread" that gives favor to minimizing the density using a single thread has much higher schedulability than "MaxThreads" that uses large densities for using the maximum parallelism. "Per-task" trades off the time and the density for each task individually and hence gives higher schedulability than "SingleThread" and "MaxThreads" in the entire range of β. "System-wide" gives even higher schedulability since it takes the system-wide tradeoff of time and density considering all the tasks together. As a result, when $\beta = 0.5$, "System-wide" shows 80%, 191%, and 3750% higher schedulability than "Per-task", "MaxThreads", and "SingleThread", respectively.

In order to study the time vs. density tradeoff along the density bound, Fig. 3.11(b) compares the schedulability of the four approaches as changing the number of CPU cores M from 1 to 8. In this experiment, the time bound D_i is uniform-randomly generated in $[e_i^1(1), P_i]$. When M is small, it is a

density bottlenecked situation and hence "SingleThread" performs better than "MaxThreads". On the other hand, when M is large, the time is likely to be bottlenecked before the density and hence "SingleThread" performs worse than "MaxThreads". Also, in this experiment of density bound control, we can observe that "Per-task" and "System-wide" take good tradeoffs of time and density, i.e., per-task in the former and system-widely in the latter.

In order to more deeply investigate this time vs. density tradeoff by the above four approaches, Fig. 3.12 shows where each task set is positioned by the four approaches in the time and density domain. For this experiment, we randomly generate 1000 task sets with $\beta = 0.5$. Out of 1000 task sets, 616 task sets turn out to be schedulable by the best approach, i.e., "System-wide" [5]. For those 616 task sets, Figs. 3.12(a), (b), (c), and (d) plot (*time usage*, *density usage*)-points to show where each task set is positioned by the four approaches. For a task set, the *time usage* implies how much portion of the time bound is used. It is defined as the maximum ratio of thread execution times to the time bounds D_is. The *density usage* implies how much portion of the density bound is used. It is defined as the ratio of the peak system density to the density bound M. Fig. 3.12(a) of "SingleThread" shows that many task sets are biased toward large *time usages* and, as a result, 600 out of 616 task sets turn out to be unschedulable due to time bound overflows, i.e., *time usage* > 1. On the other hand, Fig. 3.12(b) of "MaxThreads" shows

[5] Other 384 task sets that are unschedulable by "System-wide" turn out to be also unschedulable by the other three approaches.



(a) "SingleThread" approach

(b) "MaxThreads" approach

(c) "Per-task" approach

(d) "System-wide" approach

Figure 3.12 Distribution of task sets in $(time\ usage, density\ usage)$-space

that many task sets are biased toward large *density usages* and, as a result, 404 out of 616 task sets are unschedulable due to density bound overflows, i.e., *density usage* > 1. "Per-task" keeps all 616 task sets under the time bound by the per-task tradeoff of time and density as shown in Fig. 3.12(c). However, many task sets, i.e., 273 out of 616, are positioned above the density bound, because "Per-task" cannot take advantage of the system-wide tradeoff. In Fig. 3.12(d), we can observe that "System-wide" places the task sets in the most well balanced positions in the time and density domain.

3.5.2 Actual Implementation Results

For the practical justification of our approach, we implement it based on Open Computing Language (OpenCL) [43] on AMD FX-9590 which has four physical cores with clock rate of 4.7 GHz. We use Portable Computing Language (pocl v0.11) [31] and modify its API to control each task's number of threads and to set each thread's parameters. For the fluid scheduling of threads, we use PD^2 algorithm provided by LITMUS^{RT}-2016.1 [14]. In order to focus on the time vs. density tradeoff issue in the experiment, we avoid unpredictable interferences as follows: First, we implement per-task bank partitioning—a modification of PALLOC [69], so that threads of each task only access the given DRAM bank without interfering other tasks' bank. Second, we make tasks detour the L1, L2, and L3 caches using the virtual memory mapped I/O device provided by LITMUS^{RT} to avoid unpredictable cache effects.

On this framework, we execute five benchmark OpenCL programs, i.e.,

Monte Carlo method (τ_1), matrix transpose (τ_2), and three Gaussian image filters for three different image sizes (τ_3, τ_4, τ_5). The WCET of each thread according to the parallelization option, i.e., $e_i^k(O_i)$, is profiled beforehand as shown in Table 3.1. The tasks executing the five benchmark programs, i.e., τ_1 ~ τ_5, have the deadlines and periods as ($D_1 = 600$, $P_1 = 1,000$), ($D_2 = 600$, $P_2 = 1,000$), ($D_3 = 600$, $P_3 = 800$),($D_4 = 6,000$, $P_4 = 20,000$), ($D_5 = 16,000$, $P_5 = 30,000$), where time unit is ms. More specifically, Table 3.2 shows the solutions found by each approach.

Fig. 3.13 shows the measured response times of each task by the four approaches when the number of CPU cores is four. The horizontal line in the graphs shows the deadline D_i, and thus the response time above it means a deadline violation (X mark on the graph). In "SingleThread", all the jobs of τ_2 miss their deadlines. This is because the WCET of τ_2 for the single thread option is longer than $D_2 = 600$ ms, i.e., time bound violation (see Table 3.2). This show a case of time bound violation. The peak system density of "SingleThread" is actually 3.06, which is lower than the density bound 4. Thus, even if we use a larger number of cores, we cannot make the task set schedulable by the "SingleThread" approach. One the other hand "MaxThreads" has the peak system density of 5.58 and hence makes the task set unschedulable due to the density bound violation. "Per-task" and "System-wide" have the peak densities of 3.29 and 2.53, respectively. Also, they satisfy the time bound conditions. Thus, they can successfully schedule all the tasks meeting deadlines with four CPU cores.

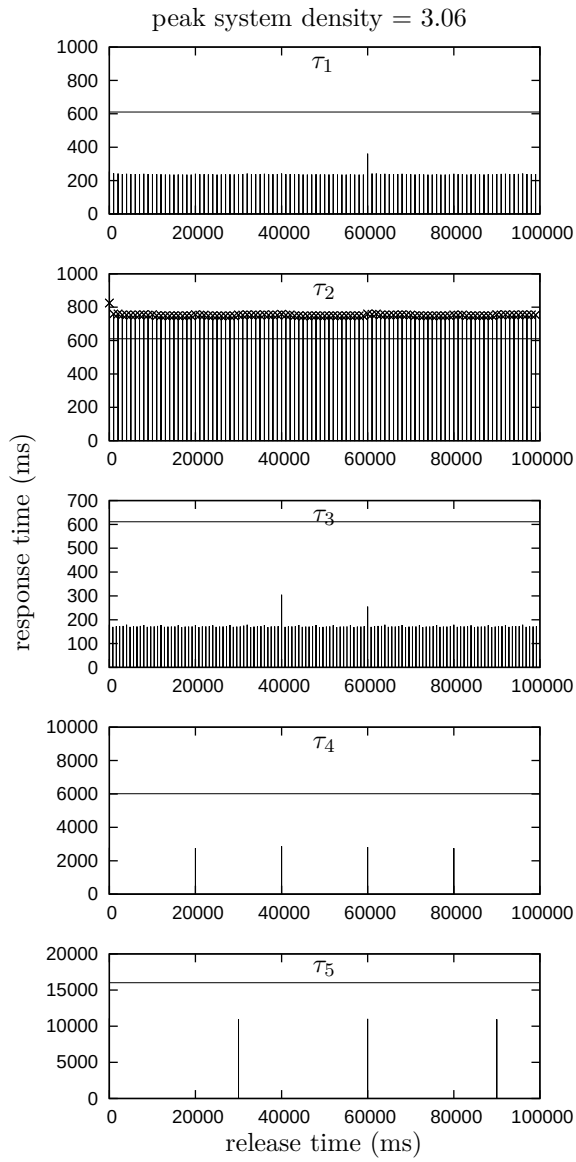
(ms)

O_i	τ_1		τ_2		τ_3		τ_4		τ_5	
	$e_i^{\max}(O_i)$	$C_i(O_i)$	$e_i^{\max}(O_i)$	$C_i(O_i)$	$e_i^{\max}(O_i)$	$C_i(O_i)$	$e_i^{\max}(O_i)$	$C_i(O_i)$	$e_i^{\max}(O_i)$	$C_i(O_i)$
$O_i = 1$	229	229	747	747	174	174	2,755	2,755	10,976	10,976
$O_i = 2$	198	395	443	886	91	182	1,563	3,125	6,241	12,481
$O_i = 3$	162	482	374	1,115	86	220	1,206	3,551	4,721	13,952
$O_i = 4$	152	573	361	1,423	58	225	1,095	4,371	4,602	18,369

Table 3.1 Profiled WCET of each task

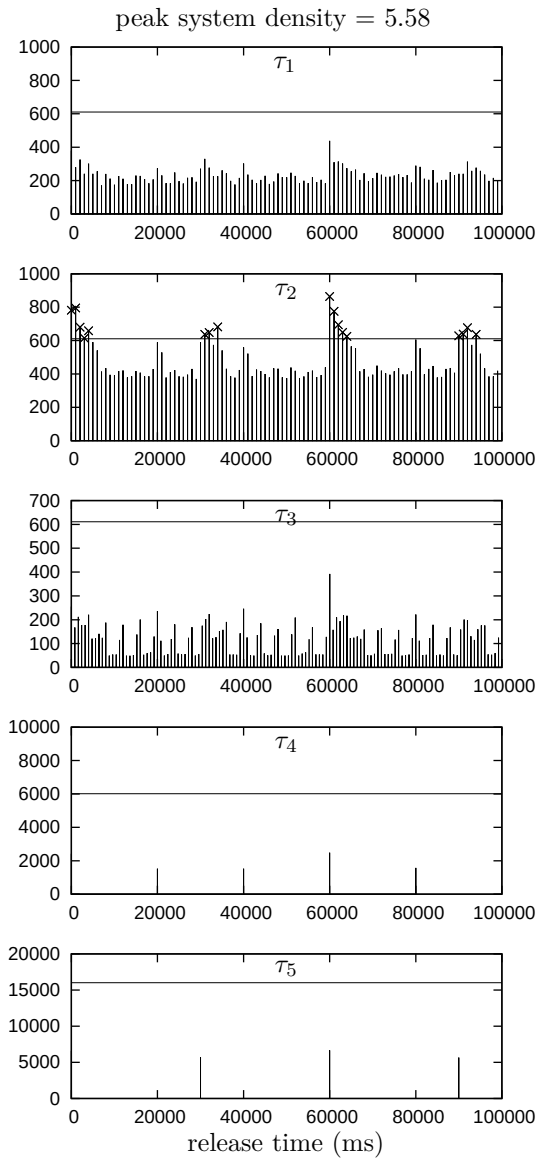
approach	task	Φ_i	d_i	p_i	O_i	$e_i^{\max}(O_i)$	$C_i(O_i)$	δ_i
"SingleThread"	τ_1	0	600	1,000	1	229	229	3.06
	τ_2	0	600	1,000	1	747	747	
	τ_3	0	600	800	1	174	174	
	τ_4	0	6,000	20,000	1	2,755	2,755	
	τ_5	0	16,000	30,000	1	10,976	10,976	
"MaxThreads"	τ_1	0	600	1,000	4	152	573	5.58
	τ_2	0	600	1,000	4	361	1,423	
	τ_3	0	600	800	4	58	225	
	τ_4	0	6,000	20,000	4	1,095	4,371	
	τ_5	0	16,000	30,000	4	46,02	18,369	
"Per-task"	τ_1	0	600	1,000	1	229	229	3.29
	τ_2	0	600	1,000	2	443	886	
	τ_3	0	600	800	1	174	174	
	τ_4	0	6,000	20,000	1	2,755	2,755	
	τ_5	0	16,000	30,000	1	10,976	10,976	
"System-wide"	τ_1	99	215	800	2	198	395	2.53
	τ_2	314	484	800	2	443	886	
	τ_3	0	99	800	2	91	182	
	τ_4	0	4,012	20,000	1	2,755	2,755	
	τ_5	4,012	15,987	20,000	1	10,976	10,976	

Table 3.2 The solutions for τ_1, τ_2, τ_3, τ_4, and τ_5 according to "SingleThread", "MaxThreads", "Per-task", and "System-wide" approaches



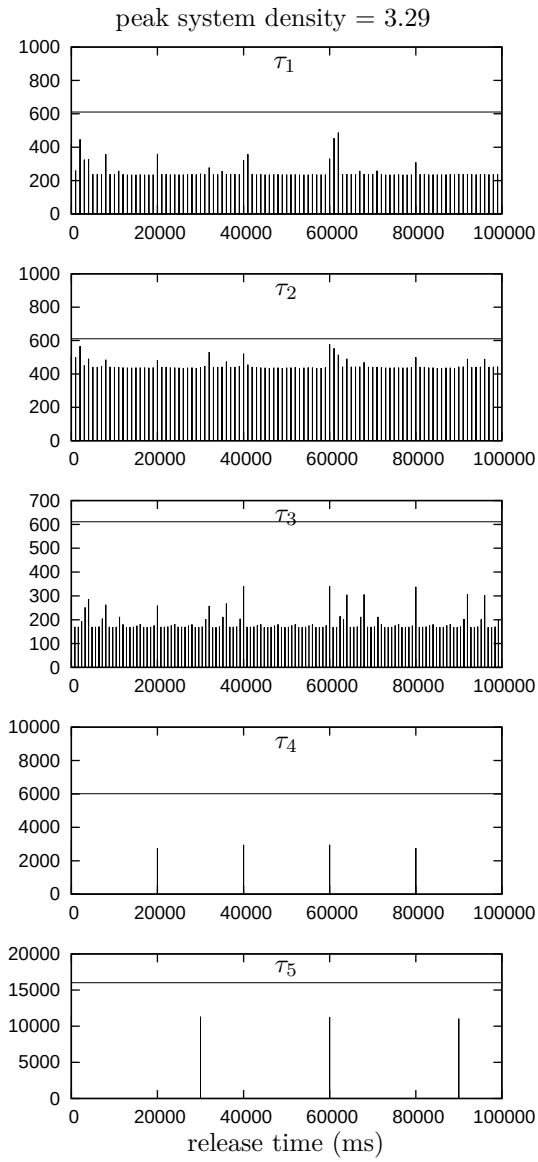
(a) "SingleThread"

Figure 3.13 Measured response times of each approach with 4 CPU cores



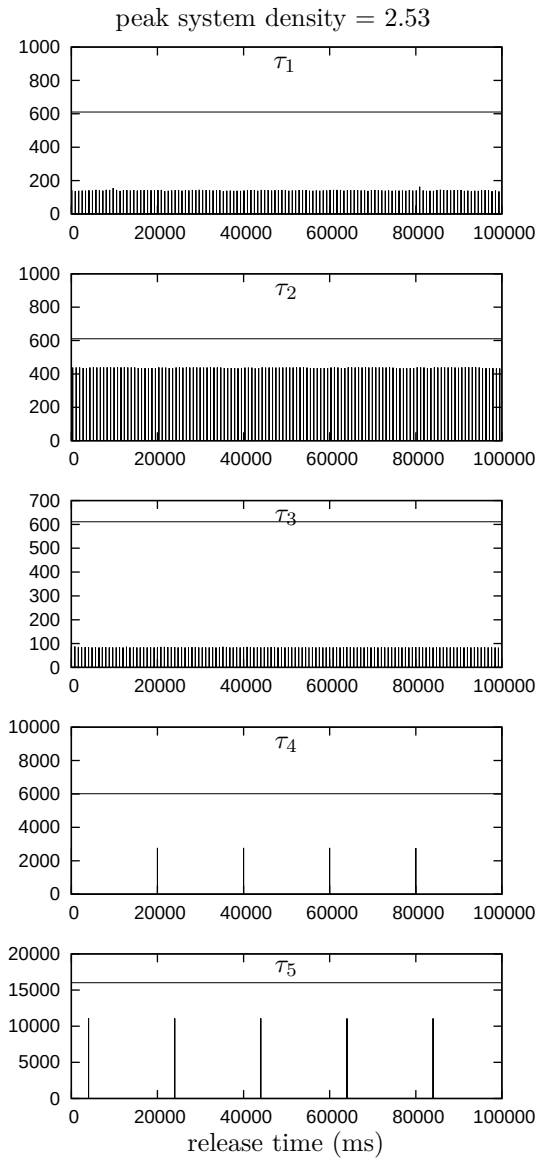
(b) "MaxThreads"

Figure 3.13 Measured response times of each approach with 4 CPU cores (cont'd)



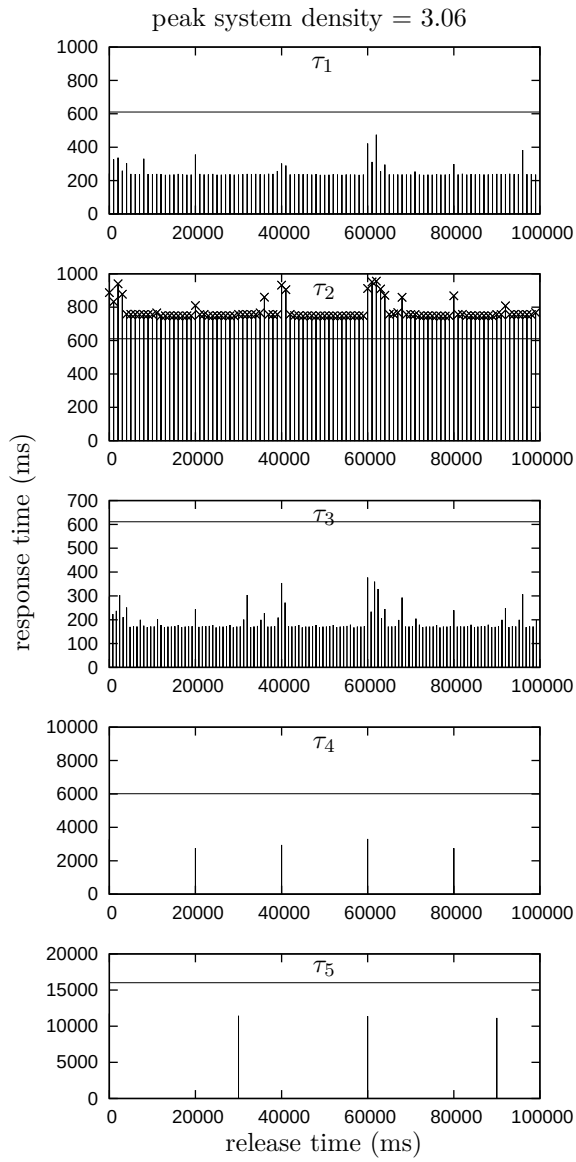
(c) "Per-task"

Figure 3.13 Measured response times of each approach with 4 CPU cores (cont'd)



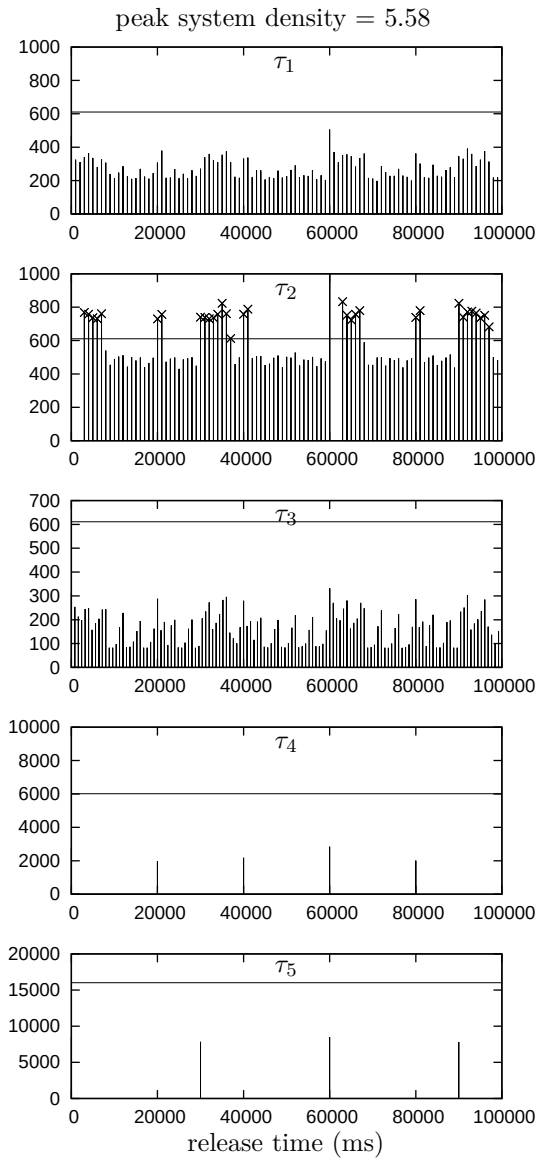
(d) "System-wide"

Figure 3.13 Measured response times of each approach with 4 CPU cores (cont'd)



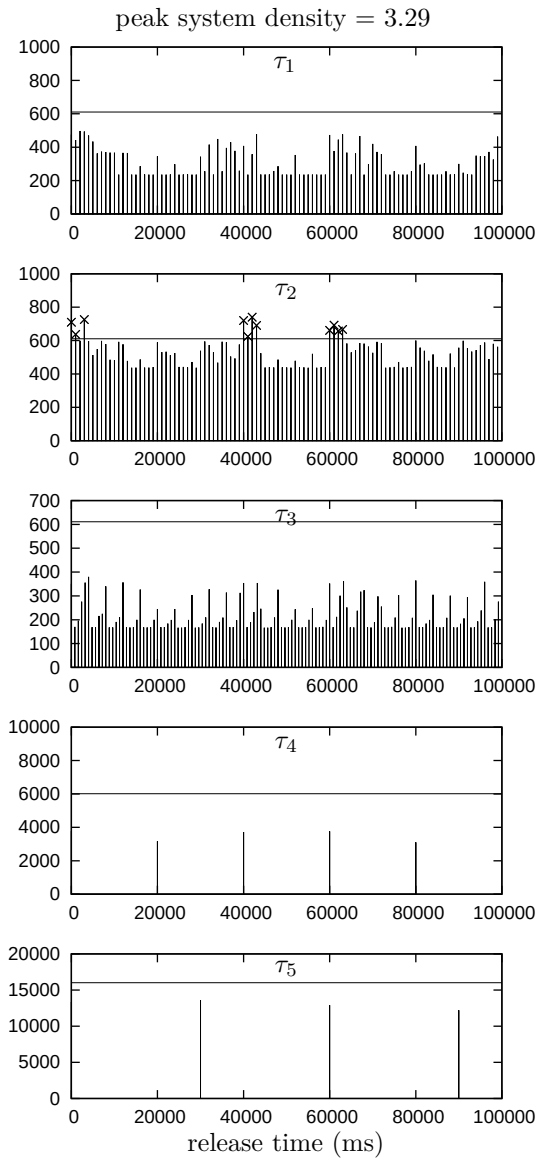
(a) "SingleThread"

Figure 3.14 Measured response times of each approach with 3 CPU cores



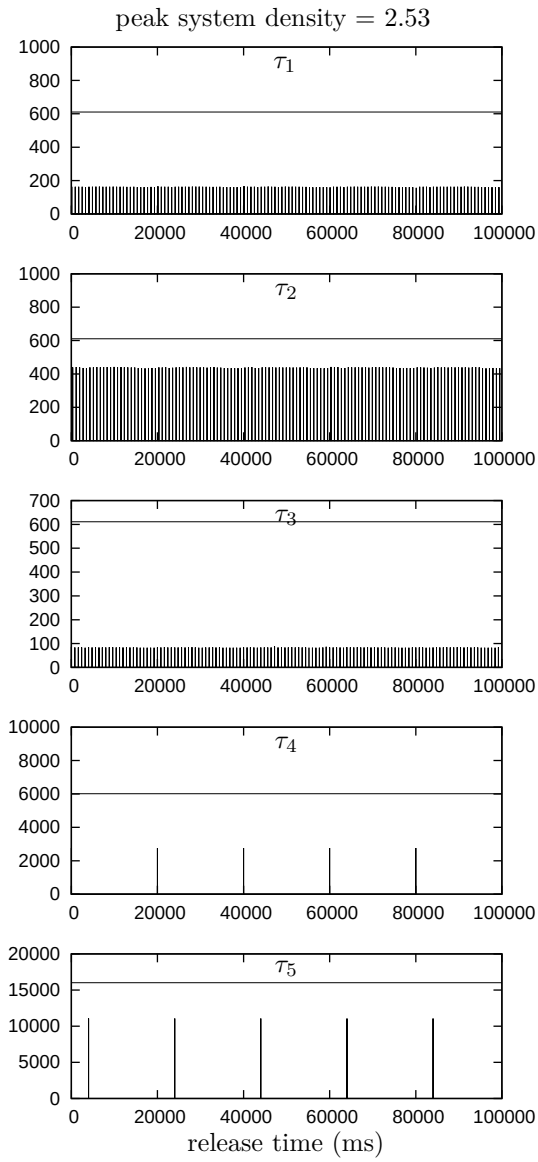
(b) "MaxThreads"

Figure 3.14 Measured response times of each approach with 3 CPU cores (cont'd)



(c) "Per-task"

Figure 3.14 Measured response times of each approach with 3 CPU cores (cont'd)



(d) "System-wide"

Figure 3.14 Measured response times of each approach with 3 CPU cores (cont'd)

Fig. 3.14 shows the same graphs when the number of CPU cores is three. In this figure, we can observe that “Per-task” misses some deadlines because its peak system density 3.29 is greater than the density bound 3. On the other hand, and“System-wide” with peak system density of 2.53 still makes the task set schedulable with three CPU cores.

Chapter 4
System-wide Time vs. Density Tradeoff with Parallelizable Periodic Multi-segment Tasks

4.1 Introduction

Chapter 3 presents per-task time vs. density tradeoff of multi-segment tasks and system-wide time vs. density tradeoff of single segment tasks. In this chapter, system-wide time vs. density tradeoff of multi-segment tasks is presented. For this, Section 4.2 describes a time vs. density tradeoff problem for multi-segment tasks. Section 4.3 explains how to extend single segment task model to multi-segment task model.

4.2 Problem Description

We consider a system with M homogeneous CPU cores and N independent tasks denoted by τ_i $(1 \leq i \leq N)$. Each task τ_i is periodically released with period P_i, and it should be finished within its relative deadline D_i as shown in Fig. 3.1. A task τ_i consists of consecutive K_i parallelizable segments denoted as s_{ik} $(1 \leq k \leq K_i)$. Just like other multi-segment model, only after s_{ik} finish, $s_{i(k+1)}$ can start. In this parallelizable periodic multi-segment task model, each segment s_{ik} can be parallelized into any number of threads. We define this

number of threads for segment s_{ik} as *parallelization option* denoted by O_{ik}. If $O_{ik} = 1$, s_{ik} is executed without parallelization. On the contrary, if $O_{ik} = 8$, s_{ik} is parallelized into 8 threads. The maximum number of O_{ik} is limited up to $O^{\max}$.

According to the O_{ik}, the threads in s_{ik} has different Worst Case Execution Time (WCET) as shown in WCET table in Fig. 4.1. $e_{ik}^{l}(O_{ik})$ is defined as an WCET of l-th thread in s_{ik} according to the parallelization option O_{ik}. For a given O_{ik}, the largest WCET of threads is denoted by $e_{ik}^{\max}(O_{ik})$, i.e.,

$$e_{ik}^{\max}(O_{ik}) = \max_{1 \le l \le O_{ik}} e_{ik}^{l}(O_{ik}).$$

Also, for a given O_{ik}, the *total computation amount* of s_{ik} which is the sum of WCET of all threads is denoted by $C_{ik}(O_{ik})$.

$$C_{ik}(O_{ik}) = \sum_{l=1}^{O_{ik}} e_{ik}^{l}(O_{ik}).$$

For the given N parallelizable periodic tasks which are scheduled on M homogeneous CPU cores, a problem to be solved is how to optimally exercise system-wide time vs. density tradeoff in terms of maximizing the overall schedulability of the entire tasks, i.e., minimizing the system peak density same as in Chapter 3.

For this problem, we consider a fluid scheduling algorithm such as PD^2 [60], LLREF [16], LRE-TL [25], DP-Fair [38] and U-EDF [47] as the underlying scheduling mechanism. In order to schedule s_{ik} based on the fluid schedule,

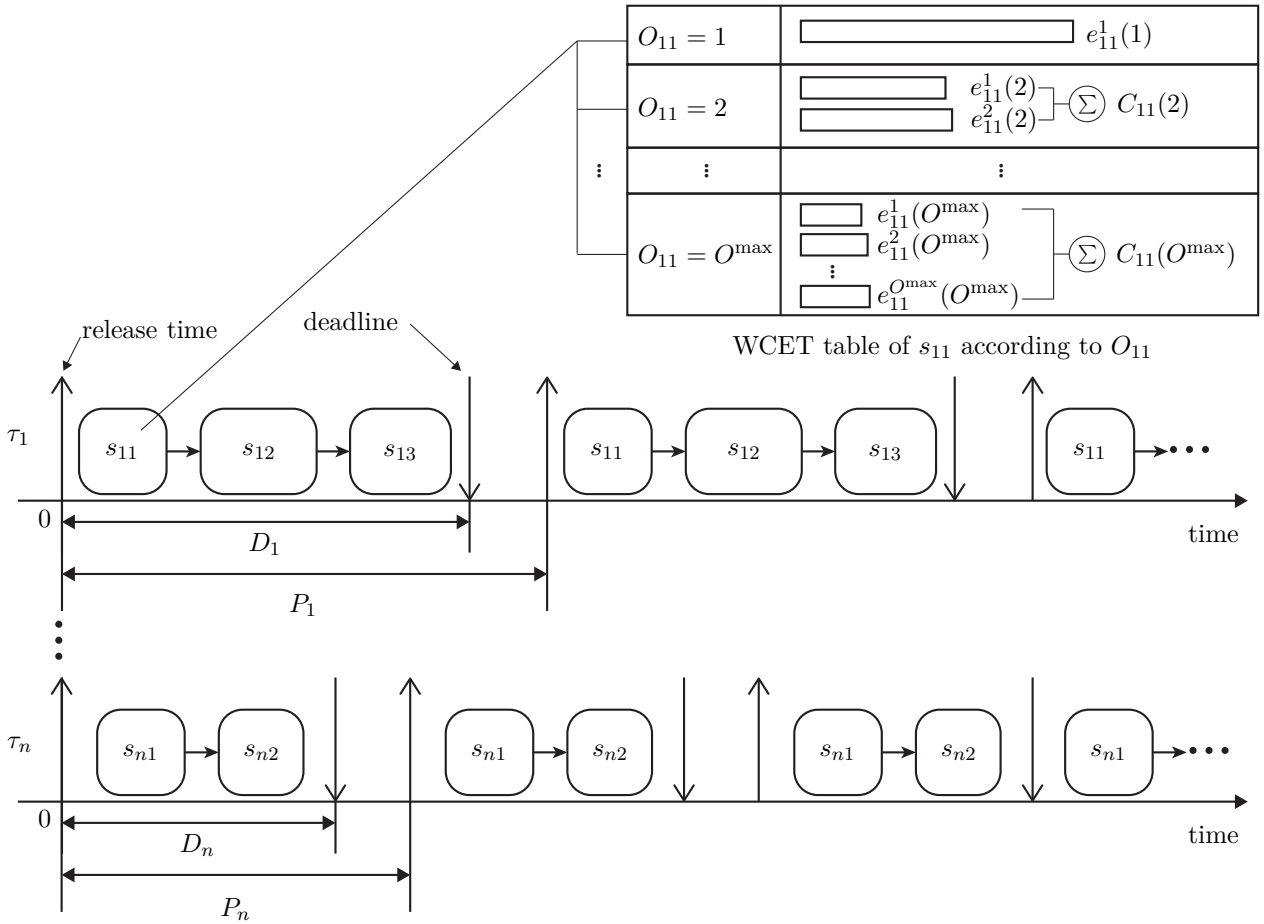


Figure 4.1 Parallelizable periodic task model

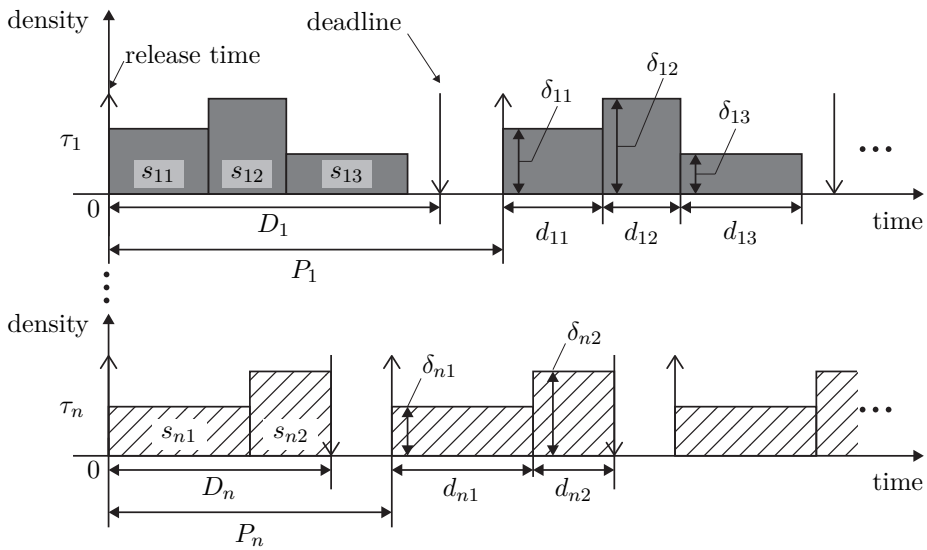


Figure 4.2 Intermediate deadline and density of segments on time-density graph.

each segment s_{ik} needs its deadline. However a task τ_i only has deadline for the task, not for its segments s_{ik}. Therefore, we introduce *intermediate deadline* which represents temporary deadline for s_{ik} denoted by d_{ik} as shown in Fig. 4.2. Since all the segments in a task should be finished before task's deadline, the sum of intermediate deadlines d_{ik}s should be shorter than the original deadline D_i. i.e.,

$$\sum_{k=1}^{K_i} d_{ik} \leq D_i. \tag{4.1}$$

When we extend the parallelizable periodic single-segmented task model in Chapter 3 to the parallelizable periodic multi-segment task model in this chapter, the controllable knobs for τ_i are depicted in Fig. 4.3, i.e., 1) τ_i's offset

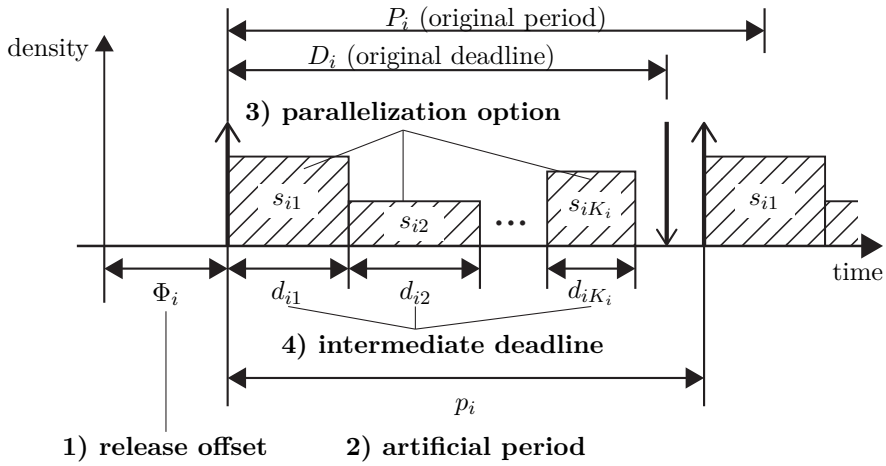


Figure 4.3 Four control values for τ_i used in system-wide time vs. density tradeoff.

Φ_i, 2) τ_i's artificial period p_i, 3) s_{ik}'s parallelization option O_{ik}, and 4) s_{ik}'s intermediate deadline d_{ik}. Note that τ_i's artificial deadline d_i in Chapter 3 is replaced with s_{ik}s' intermediate deadline d_{ik}s since the sum of intermediate deadlines represents the definition of the artificial deadline under condition of Eq. (4.1).

The density of the segment s_{ik} at a certain time instant t, denoted by δ_{ik},

is formulated as follows:

$$\delta_{ik}((\Phi_i, \vec{\boldsymbol{O}}_{\boldsymbol{i}}, \vec{\boldsymbol{d}}_{\boldsymbol{i}}, p_i), t) = \begin{cases} C_{ik}(O_{ik})/d_{ik} & (t - (\Phi_i + \phi_{ik})) \bmod p_i \leq d_{ik} \\ 0 & (t - (\Phi_i + \phi_{ik})) \bmod p_i > d_{ik} \end{cases} \tag{4.2}$$

$$\text{,where} \quad \vec{\boldsymbol{O}}_{\boldsymbol{i}} = (O_{i1}, \cdots, O_{iK_i})$$

$$\vec{\boldsymbol{d}}_{\boldsymbol{i}} = (d_{i1}, \cdots, d_{iK_i})$$

$$\phi_{ik} = \begin{cases} 0 & k = 1 \\ \sum_{j=1}^{k-1} d_{ij} & k > 1 \end{cases}$$

meaning that the density of the segment is $C_{ik}(O_i)/d_{ik}$ while the task is active and zero otherwise. Note that phi_{ik} in Eq. (4.2) represents the offset of segment s_{ik} within τ_i. Since the segments s_{ik}s in τ_i are sequentially executed, the active interval of s_{ik} does not overlap others. Thus the density of the tau_i denoted by δ_i is simply sum of δ_{ik}s as follow:

$$\delta_i((\Phi_i, \vec{\boldsymbol{O}}_{\boldsymbol{i}}, \vec{\boldsymbol{d}}_{\boldsymbol{i}}, p_i), t) = \sum_{k=1}^{K_i} \delta_{ik}((\Phi_i, \vec{\boldsymbol{O}}_{\boldsymbol{i}}, \vec{\boldsymbol{d}}_{\boldsymbol{i}}, p_i), t)$$

With this density function of each task τ_i, the system-side time vs. density tradeoff problem can be formulated as the following problem of minimizing the peak of the system density, i.e., the sum of all N task density functions, within the hyper period HP while satisfying the time bound condition for

each tread:

$$\text{minimize} \quad \max_{0 \le t < HP} \sum_{i=1}^{N} \delta_i((\Phi_i, \vec{\boldsymbol{O}}_{\boldsymbol{i}}, \vec{\boldsymbol{d}}_{\boldsymbol{i}}, p_i), t)$$

subject to

$$\begin{aligned}
0 \le \quad & p_i & \le P_i \quad & (1 \le i \le N) \\
0 \le \quad & \Phi_i & < p_i \quad & (1 \le i \le N) \\
1 \le \quad & O_i & \le O^{\max} \quad & (1 \le i \le N) \\
& \sum_{k=1}^{K_i} d_{ik} & \le D_i \quad & (1 \le i \le N) \\
e_{ik}^{\max}(O_{ik}) \le \quad & d_{ik} & & (1 \le i \le N)
\end{aligned}$$

4.3 Extension to Parallelizable Periodic Multi-segment Task Model

Based on the parallelizable periodic single segment task model in Chapter 3, in this section, we consider extension to the time vs. density tradeoff of parallelizable periodic multi-segment task model.

For the convenience explanation in following sections, let define function $\delta^{\,\mathrm{PTO}}(\tau_i)$ as the minimum peak density of τ_i derived by per-task optimization in [36]. In the same way, functions $d_{ik}^{\mathrm{PTO}}(\tau_i)$ and $O_{ik}^{\mathrm{PTO}}(\tau_i)$ are defined by intermediate deadline and parallelization option of s_{ik} when τ_i has minimum peak density.

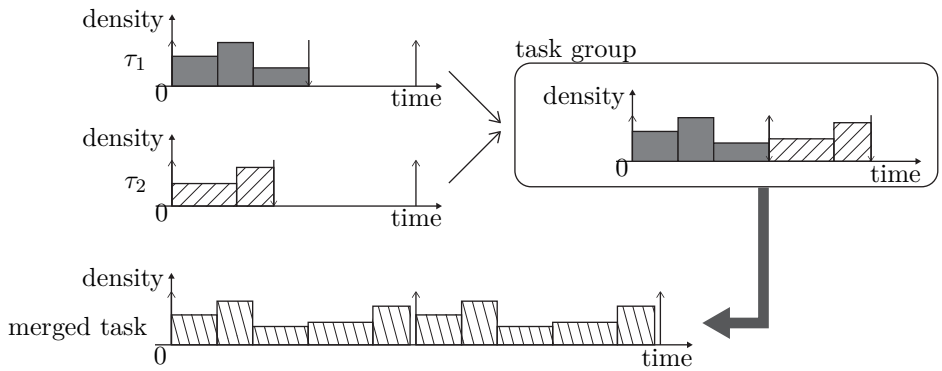


Figure 4.4 Example of merged task of τ_1 and τ_2.

4.3.1 Peak Density Minimization for a Task Group of Multi-segment Tasks with Same Period

As described in Section 3.4.2, task groups are mutually exclusive subset of task set, and the tasks in a task group do not overlap each other vertically on time-density graph. Since all tasks in the group have same period, by controlling the release offset, tasks can be horizontally placed on entire time-density graph.

The basic idea of the proposed method in this section is converting a problem of minimizing the peak density of a task group into the problem of minimizing the peak density of a task, i.e., the per-task minimization problem described in [36]. More specifically, we can convert multi-segment tasks in a task group with same period in into one merged multi-segment task which includes all segments of the tasks in the group. As an example, Fig. 4.4 shows the merged task when τ_1 and τ_2 are in the group.

When G is a set of all tasks in the task group whose period are identical as

P^G, let's define τ^G as "merged task" which contains all segments of the group, i.e., $\forall_{i \in \{i|\tau_i \in G\}} s_{ik}$ $(1 \leq k \leq K_i)$. Then the properties of τ^G is as follows:

1) **The number of segments (K^G):** The number of τ^G's segments K^G is the sum of the number of segments in G. Thus, K^G is defined as $K^G = \sum_{i \in \{i|\tau_i \in G\}} K_i$.

2) **The period of τ^G (P^G):** Since all the tasks in the group are released with same period P^G, τ^G is also repeated with period of P^G. Thus, we can say that the period of τ^G is P^G.

3) **The deadline of τ^G (D^G):** In order to prevent segments in τ^G vertically overlap[1], τ^G should be finished before its next release. Therefore, the deadline of τ^G, i.e., D^G, is same as its period P^G.

4) **The intermediate deadline of the segments in τ^G:** The segments in τ^G are originated from the tasks in G. The intermediate deadlines of s_{ik} $(1 \leq i \leq K_i)$ in $\tau_i \in G$ have the constraint that their sum is less or equal to the deadline of τ_i, i.e., D_i. However, with the merged task τ^G, this constraint is applied to the entire segments in τ^G not to the segments in individual task τ_i. Thus the additional constraints for the segments originated from individual task is needed. In order to distinguish the segment s_{ik} in τ_i from the segment originated from τ_i but in τ^G, additional terms are introduced. In this section, for the segments in τ^G, s^G_{ik}, d^G_{ik}, and O^G_{ik} is defined as the segment in G originated from

[1]This is according to the definition of task group in Section 3.4.2.

s_{ik}, the intermediate deadline of s_{ik}^G, and parallelization option of s_{ik}^G, respectively.

5) **The WCET of segments in τ^G:** Even if segments are recombined inside τ^G, their WCET according to the parallelization option $e_{ik}^l(O_{ik})$, i.e., WCET table in Fig. 4.1, are not changed.

With this properties, the per-task time vs. density tradeoff of τ^G finds the optimal peak density, i.e., $\delta^{\,\mathrm{PTO}}(\tau^G)$, with the corresponding intermediate deadlines of each segment, i.e., $d_{ik}^{\mathrm{PTO}}(\tau^G)$ $(i \in \{i \mid \tau_i \in G\},\ 1 \le k \le K_i)$, and parallelization options, i.e., O_{ik}^{PTO} $(i \in \{i \mid \tau_i \in G\},\ 1 \le k \le K_i)$. However, as described above, this solution for τ^G does not consider the intermediate deadline constraints of tasks in the task group G, i.e., $\sum_{k=1}^{K_i} d_{ik} \le D_i$ $(i \in \{i \mid \tau_i \in G\})$.

The proposed approach in this section finds the peak density of the task group G, i.e., the peak density among $\tau_i \in G$, by comparing the result of the per-task optimization of τ^G with the result of the per-task optimization of τ_i. First of all, the following lemma says that if the per-task optimization of τ^G cannot find a solution due to too tight time bound, then the solution for the task group G also cannot be found.

Lemma 4.1. *The task group G does not have a solution if the per-task optimization of τ^G does not have a solution. And this impossible solution only happens when time bound is smaller than the sum of maximum WCET of the thread of all segments in G when they are maximally parallelized, i.e.,*

$\sum_{i\in\{i|\tau_i\in G\}}\sum_{k=1}^{K_i} e_{ik}^{max}(O^{max}) < D^G$.

Proof. We can prove this lemma with Fig. 3.8. During the per-task optimization of τ^G, the optimal density in continuous domain is found by applying D^G into x-axis of Fig. 3.8(c). In this graph, the optimal density always can be found if the sum of intermediate deadlines ($\sum_{i=1}^{n} d_i$ in Fig. 3.8(c)) is greater than the sum of maximum WCET among threads when maximally parallelized ($\sum_{k=1}^{K_i} e_{ik}^{\max}$ in the case of Fig. 3.8). Thus, for the per-task optimization of τ^G, it does not have the solution only when $\sum_{i\in\{i|\tau_i\in G\}}\sum_{k=1}^{K_i} e_{ik}^{\max}(O^{\max}) < D^G$. In this case, the task group G does not have the solution since the minimum time requirement of each segment which is placed horizontally is $e_{ik}^{\max}(O^{\max})$ and their sum is greater than time bound D^G. Thus the task group G cannot have a solution. □

Contrast to the Lemma 4.1, the following two lemmas (Lemmas 4.2 and 4.3) say that if the per-task optimization finds a solution then the task group G also can find a solution. The following lemma says if the per-task optimization of τ^G finds a solution that satisfies each task's intermediate constraints, then this solution makes the peak density of task group G optimal.

Lemma 4.2. *If the per-task optimization of τ^G finds a solution and its solution satisfies the following constraint:*

$$\sum_{k=1}^{K_i} d_{ik}^{PTO}(\tau^G) \le D_i \qquad \text{,where } i \in \{i \mid \tau_i \in G\}$$

for all tasks in G, i.e., $\tau_i \in G$, then the peak density of τ^G, i.e., $\delta^{PTO}(\tau^G)$, is the peak density of the task group G.

Proof. Since all segments in τ_G are horizontally placed in the task group G, if the per-task optimization of τ^G finds optimal solution that satisfies above intermediate deadline constraint for the all segments, then the per-task optimization problem of τ^G becomes exactly same problem that minimizes the peak density of task group G. Therefore the peak density of the per-task optimization of τ^G is the peak density of the task group G. □

For the case where the solution found by the per-task optimization of τ^G does not meet the intermediate deadline constraint in Lemma 4.2, the following lemma says that the per-task optimization of the individual task is an optimal solution.

Lemma 4.3. *For the τ^G's segments originated from a certain task $\tau_i \in G$, i.e., $s_{ik}^G (1 \leq k \leq K_i)$, if $\sum_{k=1}^{K_i} d_{ik}^{PTO}(\tau^G)$ is greater than D_i then $d_{ik}^{PTO}(\tau_i)$ and $O_{ik}^{PTO}(\tau_i)$ are the optimal intermediate deadline and parallelization option for s_{ik}^Gs that make the peak density of task group minimum.*

Proof. If $\sum_{k=1}^{K_i} d_{ik}^{\mathrm{PTO}}(\tau^G)$ is greater than D_i, as the result of per-task optimization of τ^G, then that means the per-task optimization of τ^G excessively utilizes the time up to its time bound, i.e., $\sum_{\{j|\tau_j \in G\}} \sum_{k=1}^{K_j} d_{jk}^{\mathrm{PTO}}(\tau^G) = D^G$. Thus applying smaller than $d_{ik}^{\mathrm{PTO}}(\tau^G)$ to d_{ik}^G still meets the time bound of τ_G. Since the peak density of τ_i, i.e., δ_i, monotonically decreases when $\sum_{k=1}^{K_i} d_{ik}$ increases (see Fig. 3.8), under the condition of $\sum_{k=1}^{K_i} d_{ik}^G \leq D_i$, the minimum

peak density of s_{ik}^G $(1 \leq k \leq K_i)$ is found when $\sum_{k=1}^{K_i} d_{ik}^G = D_i$ with $d_{ik}^{\mathrm{PTO}}(\tau_i)$ and $O_{ik}^{\mathrm{PTO}}(\tau_i)$. Therefore, for the given τ_i, when $\sum_{k=1}^{K_i} d_{ik}^{\mathrm{PTO}}(\tau^G)$ is greater than D_i, $d_{ik}^{\mathrm{PTO}}(\tau_i)$ and $O_{ik}^{\mathrm{PTO}}(\tau_i)$ are solutions for d_{ik}^G and O_{ik}^G $(1 \leq k \leq K_i)$, respectively. □

If the sum of $d_{ik}^{\mathrm{PTO}}(\tau^G)$ is longer than D_i, as the result of the per-task optimization of τ^G, it cannot be a solution for the task group G although it is a solution for the merged task τ^G. The Lemma 4.3 means that, in this case, applying the result of the per-task optimization of τ_i to the segments originated from τ_i makes the peak density of the task group G optimal without changing densities of other tasks's segments. At the view point of individual task τ_i in G, the per-task optimization of τ_i is an optimal solution when the time bound is fixed as D_i which means the peak density cannot be reduced under the constraints $\sum_{k=1}^{K_i} d_{ik} < D_i$. Thus, even if the per-task optimization of τ^G finds the peak density smaller than the per-task optimization of τ_i, it is because that the per-task optimization gives more time bound than D_i which cannot be happened in the task group G.

From the Lemmas 4.1, 4.2, and 4.3, the following theorem states the optimal solution for task group G.

Theorem 4.1. *For the task group G, there exist a solution only if the per-task optimization of τ^G has solution. In this case, for the segments s_{ik}^G, if $\sum_{k=1}^{K_i} d_{ik}$ $PTO(\tau^G)$ is smaller or equal to D_i then the result of the per-task optimization of τ_G is the optimal. Otherwise, the result of the per-task opti-*

mization of τ_i is the optimal.

Proof. If the per-task optimization of τ^G does not have a solution the task group also does not have a solution as said in Lemma 4.1. If the per-task optimization of τ^G has a solution, according to the range of the calculated intermediate deadline, this solution is used as the optimal solution for task group G (Lemma 4.2 or the optimal solution of the individual task is used (Lemma 4.3. Since Lemmas 4.1, 4.2, and 4.3 give the optimal solution for task group G in all possible cases according to the result of the per-task optimization of τ^G, the above procedure finds an optimal solution. □

Thanks to Theorem 4.1, according to the result of the per-task optimization of τ^G, the intermediate deadlines and parallelization options that make the peak density of task group G optimal, i.e., d_{ik} and O_{ik}, are as follow:

$$
\begin{aligned}
&\text{if } \sum_{k=1}^{K_i} d_{ik}^{\mathrm{PTO}}(\tau^G) \le D_i, \\
&\qquad d_{ik} = d_{ik}^G \;=\; d_{ik}^{\mathrm{PTO}}(\tau^G) \\
&\qquad O_{ik} = O_{ik}^G \;=\; O_{ik}^{\mathrm{PTO}}(\tau^G) \\
&\text{if } \sum_{k=1}^{K_i} d_{ik}^{\mathrm{PTO}}(\tau^G) < D_i, \\
&\qquad d_{ik} = d_{ik}^G \;=\; d_{ik}^{\mathrm{PTO}}(\tau_i) \\
&\qquad O_{ik} = O_{ik}^G \;=\; O_{ik}^{\mathrm{PTO}}(\tau_i).
\end{aligned}
$$

After deciding d_{ik}^Gs and O_{ik}^Gs for all τ_is in G, we can get the release offset Φ_i for each τ_i. Since τ_i starts after the prior tasks in G finishes, Φ_i is sum of

the response times of the prior tasks.

$$\Phi_i = \sum_{j \in \{j | \tau_j \in G, j < i\}} \sum_{k=1}^{K_j} d_{jk}$$

The peak density of the task group is the highest peak density of the task. If τ_i adopts the result of per-task optimization of τ^G, i.e., in case of Lemma 4.2, the peak density of τ_i is $\delta^{\,\mathrm{PTO}}(\tau^G)$. In this case, $\delta^{\,\mathrm{PTO}}(\tau^G)$ is always greater than $\delta^{\,\mathrm{PTO}}(\tau_i)$ because s_{ik}s are placed within shorter time bound. Otherwise, i.e., in case of Lemma 4.3, the peak density of τ_i is $\delta^{\,\mathrm{PTO}}(\tau_i)$ which is always greater than $\delta^{\,\mathrm{PTO}}(\tau^G)$. Therefore the peak density of the task group denoted by δ^G is :

$$\delta^G = \max\Big(\max_{\tau_i \in G} \ \delta^{\,\mathrm{PTO}}(\tau_i),\ \delta^{\,\mathrm{PTO}}(\tau^G)\Big).$$

4.3.2 Heuristic Algorithm for System-wide Time vs. Density Tradeoff

As in Chapter 3, this section proposes a heuristic algorithm for a system-wide tradeoff of time and density using the peak density minimization of the task group in Section 4.3.1. Basically, this section extends the heuristic algorithm in Section 3.4.3 to the parallelizable periodic multi-segment task model described in Section 4.2.

Heuristic Algorithm: For each task τ_i in the given task set $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_N\}$,

find the artificial period $p_i(\leq P_i)$, the offset Φ_i, the intermediate deadlines d_{ik}, and the parallelization option O_{ik} such that the peak system density can be maintained as low as possible while satisfying the time bound conditions of all the tasks.

Input: Given set of tasks, i.e., $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_N\}$.

Output: Partitioned harmonic period task groups, i.e., $\Gamma[1], \Gamma[2], \cdots$.

begin procedure

1. initialized $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_N\}$
2. initialized harmonic period task group index $x = 0$
3. **while** $(\Gamma \neq \emptyset)$
4. **PeriodHarmonization**(Γ), $n = |\Gamma|$

   // $\tau_i \in \Gamma$ is sorted in ascending order of artificial period p_i
5. $x = x + 1$, $\Gamma[x] = \emptyset$, $\delta^{\text{peak}}[x] = 0$
6. **for** $i = 1$ **to** n
7. $\delta^{\text{peak}}_{\text{new}}$ = **TaskGroupOptimization** $(\Gamma[x] \cup \tau_i)$
8. **if** $(\delta^{\text{peak}}_{\text{new}} \leq \delta^{\text{peak}}[x] +$ **PerTaskOptimization**$(\tau_i))$
9. $\Gamma = \Gamma - \{\tau_i\}$, $\Gamma[x] = \Gamma[x] \cup \{\tau_i\}$
10. $\delta^{\text{peak}}[x] = \delta^{\text{peak}}_{\text{new}}$
11. **end if**
12. **end for**
13. finalize $\Gamma[x]$
14. recover the original periods of tasks left in Γ
15. **end while**

end procedure

Chapter 5
Conclusion

5.1 Summary

Motivated by the observation of the time vs. density tradeoff of real-time tasks with parallelization freedom, this dissertation proposes a system-wide tradeoff of time and density for maximizing the schedulability of multicore fluid scheduling. For this, two optimal approaches are proposed, i.e., an optimal per-task tradeoff of time and density and an optimal horizontal placement of a task group with the same period. Using those optimal methods as building blocks, a heuristic algorithm is proposed for a system-wide tradeoff of time and density by controlling artificial periods, artificial deadlines, offsets, and parallelization options for all the tasks all together. The simulation study shows that the proposed approach well balances the time and the density and, as a result, significantly improves the schedulability, and the actual implementation justifies its practical feasibility.

The extension from time vs. density tradeoff for single segment tasks to multi-segment tasks also proposed. Based on the per-task optimization and group optimization of single segment, it is shown that multi-segment tasks also can be optimized through time vs. density tradeoff without inventing additional method.

5.2 Future Work

Although this dissertation proposes system-wide tradeoff between time and density, it has several limitations. The followings are the future work that possibly covers these limitations.

- **Limitation from using fluid schedule:** Even tough fluid schedule is an optimal scheduler on multicore system, in practical use, it has larger context switch overhead comparing to other multicore scheduler such as G-EDF. Basically, the time vs. density tradeoff is based on the tradeoff between deadline and total computation amount, which also existing problem in other multicore scheduler. Therefore this tradeoff problem of other multicore scheduler should be dealt in the future.

- **Optimality of proposed algorithm:** Although per-task time vs. density tradeoff and group time vs. density tradeoff are optimal, the heuristic algorithm which uses theses as build blocks is not optimal. This is an irrefutable limitation of the proposed method. However, considering the solution space of the problem, it is clear that designing an algorithm that finds optimal solution is not easy or impossible. Thus, as a future work, identifying the property of the problem and solution space, and improving the algorithm are planned.

- **Memory access interference** The one of the important practical issues in real-time multicore system is shared memory access interference of each core. In this dissertation, bank partitioning and cache disabling

are used for actual implementation. In order to overcome this limitation, the research about optimally allocating DRAM bank to each segment is planned.

With above future planned researched, we expect that more practical and efficient solution can be found for the time vs. density tradeoff problem.

References

[1] Ieee standard for information technology - portable operating system interfaces (posix(r)) - part 1: System application program interface (api) - amendment 1: Realtime extension [c language]. *IEEE Std 1003.1b-1993*, 1994.

(Referenced on page 10.)

[2] Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Real-Time Systems Symposium, 1998. Proceedings. The 19th IEEE*, pages 4–13. IEEE, 1998.

(Referenced on page 12.)

[3] AMD. `http://developer.amd.com/tools-and-sdks/opencl-zone/`, last visited on may, 2017.

(Referenced on page 12.)

[4] James H Anderson and John M Calandrino. Parallel Real-time Task Scheduling on Multicore Platforms. In *27th IEEE Real-Time Systems Symposium (RTSS)*, 2006.

(Referenced on pages 6 and 8.)

[5] Benjamin Bado, Laurent George, Pierre Courbin, and Joël Goossens. A Semi-partitioned Approach for Parallel Real-time Scheduling. In *20th ACM International Conference on Real-Time and Network Systems*,

2012.

(Referenced on page 7.)

[6] Sanjoy Baruah. Partitioned edf scheduling: a closer look. *Real-Time Systems*, 49(6):715–729, 2013.

(Referenced on page 7.)

[7] SK Baruah, NK Cohen, CG Plaxton, and DA Varvel. Proportionate Progress: a Notion of Fairness in Resource Allocation. In *25th annual ACM symposium on Theory of computing*, 1993.

(Referenced on pages 2, 3, 6, 7, 8, 15, and 17.)

[8] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. Improved schedulability analysis of edf on multiprocessor platforms. In *17th Euromicro Conference on Real-Time Systems (ECRTS)*, 2005.

(Referenced on pages 7 and 8.)

[9] Giorgio Buttazzo and Luca Abeni. Adaptive workload management through elastic scheduling. *Real-Time Systems*, 23(1):7–24, 2002.

(Referenced on page 9.)

[10] Giorgio C Buttazzo and Luca Abeni. Smooth rate adaptation through impedance control. In *ECRTS*, pages 3–10, 2002.

(Referenced on page 9.)

[11] Giorgio C Buttazzo, Giuseppe Lipari, Marco Caccamo, and Luca Abeni. Elastic scheduling for flexible workload management. *IEEE Transactions*

on Computers, 51(3):289–302, 2002.

(Referenced on page 9.)

[12] Marco Caccamo, Giorgio Buttazzo, and Lui Sha. Capacity sharing for overrun control. In *Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE*, pages 295–304. IEEE, 2000.

(Referenced on pages 12 and 13.)

[13] Marco Caccamo, Giorgio C Buttazzo, and Deepu C Thomas. Efficient reclaiming in reservation-based real-time systems with variable execution times. *IEEE Transactions on Computers*, 54(2):198–213, 2005.

(Referenced on pages 12 and 13.)

[14] John M Calandrino, Hennadiy Leontyev, Aaron Block, UC Devi, and James H Anderson. LITMUS^{RT}: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *27th IEEE Real-Time Systems Symposium (RTSS)*, 2006.

(Referenced on pages 10 and 51.)

[15] Jian-Jia Chen and Samarjit Chakraborty. Partitioned packing and scheduling for sporadic real-time tasks in identical multiprocessor systems. In *24th Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

(Referenced on page 7.)

[16] Hyeonjoong Cho, Binoy Ravindran, and E Douglas Jensen. An Optimal Real-time Scheduling Algorithm for Multiprocessors. In *27th IEEE Real-Time Systems Symposium (RTSS)*, 2006.

(Referenced on pages 2, 3, 6, 7, 8, 15, and 65.)

[17] Sangyeun Cho and Lei Jin. Managing distributed, shared l2 caches through os-level page allocation. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 455–468. IEEE Computer Society, 2006.

(Referenced on page 13.)

[18] Hoon Sung Chwa, Jinkyu Lee, KieuMy Phan, Arvind Easwaran, and Insik Shin. Global EDF Schedulability Analysis for Synchronous Parallel Tasks on Multicore Platforms. In *25th Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

(Referenced on pages 7 and 8.)

[19] Henry Cook, Miquel Moreto, Sarah Bird, Khanh Dao, David A. Patterson, and Krste Asanovic. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 308–319, New York, NY, USA, 2013. ACM.

(Referenced on page 13.)

[20] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering,*

IEEE, 5(1):46–55, 1998.

(Referenced on page 1.)

[21] Sudarshan K Dhall and CL Liu. On a Real-time Scheduling Problem. *Operations Research*, 26(1):127–140, 1978.

(Referenced on page 6.)

[22] Leonardo Ecco, Sebastian Tobuschat, Selma Saidi, and Rolf Ernst. A mixed critical memory controller using bank privatization and fixed priority scheduling. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on*, pages 1–10. IEEE, 2014.

(Referenced on page 13.)

[23] Ming Fan and Gang Quan. Harmonic semi-partitioned scheduling for fixed-priority real-time tasks on multi-core platform. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 503–508. EDA Consortium, 2012.

(Referenced on page 9.)

[24] Frédéric Fauberteau, Serge Midonnet, and Manar Qamhieh. Partitioned Scheduling of Parallel Real-time Tasks on Multiprocessor Systems. *ACM SIGBED Review*, 8(3):28–31, 2011.

(Referenced on pages 6 and 7.)

[25] Shelby Funk. LRE-TL: An Optimal Multiprocessor Algorithm for Sporadic Task Sets with Unconstrained Deadlines. *Real-Time Systems*, 46(3):332–359, 2010.

(Referenced on pages 2, 3, 7, 8, 15, and 65.)

[26] Fei Guo, Yan Solihin, Li Zhao, and Ravishankar Iyer. A framework for providing quality of service in chip multi-processors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 343–355, Washington, DC, USA, 2007. IEEE Computer Society.

(Referenced on page 13.)

[27] C-C Han and H-Y Tyan. A better polynomial-time schedulability test for real-time fixed-priority scheduling algorithms. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 36–45. IEEE, 1997.

(Referenced on page 9.)

[28] Dan Hildebrand. An architectural overview of qnx. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, Berkeley, CA, USA, 1992. USENIX Association.

(Referenced on page 10.)

[29] Intel. `https://software.intel.com/en-us/intel-opencl`, last visited on may, 2017.

(Referenced on page 12.)

[30] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '07, pages 25–36, New York, NY, USA, 2007. ACM.

(Referenced on page 13.)

[31] Pekka Jääskeläinen, Carlos Sánchez de La Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, and Heikki Berg. pocl: A Performance-Portable OpenCL Implementation. *International Journal of Parallel Programming*, 2014.

(Referenced on pages 12 and 51.)

[32] Min Kyu Jeong, Doe Hyun Yoon, Dam Sunwoo, Mike Sullivan, Ikhwan Lee, and Mattan Erez. Balancing dram locality and parallelism in shared memory cmp systems. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12. IEEE, 2012.

(Referenced on page 13.)

[33] Shinpei Kato and Yutaka Ishikawa. Gang EDF Scheduling of Parallel Task Systems. In *30th IEEE Real-Time Systems Symposium (RTSS)*, 2009.

(Referenced on pages 6, 7, and 8.)

[34] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. Snucl: An opencl framework for heterogeneous cpu/gpu clusters. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 341–352, New York, NY, USA, 2012. ACM.

(Referenced on page 12.)

[35] Junsung Kim, Hyoseung Kim, Karthik Lakshmanan, and Ragunathan Raj Rajkumar. Parallel Scheduling for Cyber-Physical Systems: Analysis and Case Study on a Self-Driving Car. In *4th ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, 2013.

(Referenced on page 6.)

[36] Jihye Kwon, Kang-Wook Kim, Sangyoun Paik, Jihwa Lee, and Chang-Gun Lee. Multicore scheduling of parallel real-time tasks with multiple parallelization options. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.

(Referenced on pages 7, 8, 15, 46, 70, and 71.)

[37] Karthik Lakshmanan, Shinpei Kato, and Ragunathan Rajkumar. Scheduling Parallel Real-time Tasks on Multi-core Processors. In *31st IEEE Real-Time Systems Symposium (RTSS)*, 2010.

(Referenced on pages 6, 7, and 8.)

[38] Greg Levin, Shelby Funk, Caitlin Sadowski, Ian Pye, and Scott Brandt. DP-FAIR: A Simple Model for Understanding Optimal Multiprocessor Scheduling. In *22nd Euromicro Conference on Real-Time Systems*

(ECRTS), 2010.

(Referenced on pages 2, 3, 7, 8, 15, and 65.)

[39] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 367–378. IEEE, 2008.

(Referenced on page 13.)

[40] Giuseppe Lipari and Sanjoy Baruah. Greedy reclamation of unused bandwidth in constant-bandwidth servers. In *Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on*, pages 193–200. IEEE, 2000.

(Referenced on pages 12 and 13.)

[41] Ernesto Massa, George Lima, Paul Regnier, Greg Levin, and Scott Brandt. Quasi-partitioned scheduling: optimality and adaptation in multiprocessor real-time systems. *Real-Time Systems*, pages 1–32, 2016.

(Referenced on page 7.)

[42] Nasro Min-Allah, Ishtiaq Ali, Jiansheng Xing, and Yongji Wang. Utilization bound for periodic task set with composite deadline. *Computers & Electrical Engineering*, 36(6):1101–1109, 2010.

(Referenced on page 9.)

[43] Aaftab Munshi et al. The OpenCL Specification. *Khronos OpenCL Working Group*, 1:l1–15, 2009.

(Referenced on pages 1 and 51.)

[44] Mitra Nasri and Gerhard Fohler. An efficient method for assigning harmonic periods to hard real-time tasks with period ranges. In *27th Euromicro Conference on Real-Time Systems (ECRTS)*, 2015.

(Referenced on pages 9, 40, 100, and 101.)

[45] Mitra Nasri, Gerhard Fohler, and Mehdi Kargahi. A framework to construct customized harmonic periods for real-time systems. In *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, pages 211–220. IEEE, 2014.

(Referenced on page 9.)

[46] Geoffrey Nelissen, Vandy Berten, Joël Goossens, and Dragomir Milojevic. Techniques Optimizing the Number of Processors to Schedule Multi-threaded Tasks. In *24th Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

(Referenced on pages 6, 7, 8, and 15.)

[47] Geoffrey Nelissen, Vandy Berten, Vincent Nélis, Joël Goossens, and Dragomir Milojevic. U-EDF: An Unfair but Optimal Multiprocessor Scheduling Algorithm for Sporadic Tasks. In *24th Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

(Referenced on pages 2, 3, 7, 8, 15, and 65.)

[48] NVIDIA. `https://developer.nvidia.com/opencl`, last visited on may, 2017.

(Referenced on page 12.)

[49] OpenMP Architecture Review Board. OpenMP fortran application program interface version 1.0, October 1997.

(Referenced on page 10.)

[50] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.

(Referenced on page 11.)

[51] OpenMP Architecture Review Board. OpenMP application program interface version 4.0, 2013.

(Referenced on page 11.)

[52] Manar Qamhieh, Serge Midonnet, and Laurent George. Dynamic Scheduling Algorithm for Parallel Real-time Graph Tasks. *ACM SIGBED Review*, 9(4):25–28, 2012.

(Referenced on page 7.)

[53] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 423–432, Dec 2006.

(Referenced on page 13.)

[54] Ragunathan Rajkumar, Kanaka Juvva, Anastasio Molano, and Shuichi Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Photonics West'98 Electronic Imaging*, pages 150–164. International Society for Optics and Photonics, 1997.

(Referenced on page 12.)

[55] RTAI = the RealTime Application Interface fo Linux. `https://www.rtai.org/`, last visited on Oct. 18, 2014.

(Referenced on page 10.)

[56] Abusayeed Saifullah, David Ferry, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D Gill. Parallel real-time scheduling of dags. *IEEE Transactions on Parallel and Distributed Systems*, 25(12):3242–3252, 2014.

(Referenced on pages 7 and 8.)

[57] 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.

(Referenced on pages 6, 7, and 8.)

[58] Saeed Senobary and Mahmoud Naghibzadeh. Ss-drm: Semi-partitioned scheduling based on delayed rate monotonic on multiprocessor platforms. *Journal of Computing Science and Engineering*, 8(1):43–56, 2014.

(Referenced on page 7.)

[59] Chi-Sheng Shih, Sathish Gopalakrishnan, Phanindra Ganti, Marco Caccamo, and Lui Sha. Scheduling real-time dwells using tasks with synthetic periods. In *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE*, pages 210–219. IEEE, 2003.

(Referenced on page 9.)

[60] Anand Srinivasan and James H Anderson. Fair Scheduling of Dynamic Task Systems on Multiprocessors. *Journal of Systems and Software*, 77(1):67–80, 2005.

(Referenced on pages 2, 3, 6, 7, 8, 15, and 65.)

[61] Anand Srinivasan and Sanjoy Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Information Processing Letters*, 84(2):93–98, 2002.

(Referenced on pages 7 and 8.)

[62] Anand Srinivasan, Philip Holman, James H Anderson, and Sanjoy Baruah. The case for fair multiprocessor scheduling. In *International Symposium on Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

(Referenced on pages 7 and 8.)

[63] G Edward Suh, Srinivas Devadas, and Larry Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *High-Performance Computer Architecture, 2002. Proceedings. Eighth In-*

ternational Symposium on, pages 117–128. IEEE, 2002.

(Referenced on page 13.)

[64] David Tam, Reza Azimi, Livio Soares, and Michael Stumm. Managing shared l2 caches on multicore systems in software. In *Workshop on the Interaction between Operating Systems and Computer Architecture*, pages 26–33. Citeseer, 2007.

(Referenced on page 13.)

[65] Ryoji Tsuchiyama, Takashi Nakamura, Takuro Iizuka, Akihiro Asahara, Satoshi Miki, and Satoru Tagawa. The opencl programming book. *Fixstars Corporation*, 63, 2010.

(Referenced on page 11.)

[66] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. Openacc: First experiences with real-world applications. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 859–870, Berlin, Heidelberg, 2012. Springer-Verlag.

(Referenced on page 11.)

[67] Wind River VxWorks. `https://www.windriver.com/products/vxworks/`, last visited on Oct. 18, 2014.

(Referenced on page 10.)

[68] Mingli Xie, Dong Tong, Kan Huang, and Xu Cheng. Improving system throughput and fairness simultaneously in shared memory cmp systems

via dynamic bank partitioning. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 344–355. IEEE, 2014.

(Referenced on page 13.)

[69] H. Yun, R. Mancuso, Z. P. Wu, and R. Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166, April 2014.

(Referenced on pages 13, 16, and 51.)

[70] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, April 2013.

(Referenced on pages 12, 13, and 16.)

[71] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory bandwidth management for efficient performance isolation in multi-core platforms. *IEEE Transactions on Computers*, 65(2):562–576, 2016.

(Referenced on pages 12 and 13.)

[72] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. Cpi2: Cpu performance isolation for shared compute

clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 379–391, New York, NY, USA, 2013. ACM.

(Referenced on page 13.)

Appendix A
Period Harmonization

The task group minimization in Chapters 3 and 4 considers the tasks with same period. If tasks have different periods, there is lower chance to decrease peak density by controlling the release offset. Because the system peak density is calculated according to every release pattern of job instances from system start time to the hyper period of tasks. Therefore, if the periods of tasks are reduced to be same or integer multiple of others, although the periods become shorter than original ones, overall system peak density can be reduced. This section explains the method called *period harmonization* which makes periods of multiple tasks harmonic.

The definition of harmonic period is as follow. For a given set of n tasks τ_is $(1 \leq i \leq n)$ with increasing order of period P_i, i.e., $P_i \leq P_{i+1}$, the task set has harmonic periods if P_{i+1} is integer multiple of P_i, i.e.,

$$\underset{1 \leq i \leq n-1}{\forall} P_i \cdot z_i = P_{i+1}, \qquad \text{where } z_i \text{ is positive integer.}$$

Note that by the definition of harmonic period, $P_{i+2}, P_{i+3}, \cdots$ are also integer multiple of P_i.

The work in [44], the authors proposed the period harmonization method when the period of task is given as the possible period range. Within this

period range, the method in [44] finds harmonic periods for all tasks.

The proposed method in Chapters 3 and 4 applies this period harmonization method into the system-wide time vs. density problem to find the artificial period p_i. Although there is no given period range in the problem, possible artificial period has upper bound and lower bound. Since the artificial period p_i should not be longer than the original period P_i, the upper bound of p_i is P_i. Also, the artificial period p_i can not be shorter than the longest WCET of the task τ_i. In the case of parallelizable periodic single segment task model in Chapter 3, the longest WCET is same as the execution time of thread when the task is not parallelized, i.e.,

$$e_i^{\max}(1). \tag{A.1}$$

In the case of parallelizable periodic multi-segment task model in Chapter 4, the longest WCET of the task is the sum of the longest WCETs of segments, i.e., the sum of WCET of a thread when the parallelization option O_{ik} is one, i.e.,

$$\sum_{k=1}^{K_i} e_{ik}^{\max}(1). \tag{A.2}$$

Since Eq. (A.1) is a special case of Eq. (A.2) when K_i is one, the generalized

the range of p_i is as follow:

$$\sum_{k=1}^{K_i} e_{ik}^{\max}(1) \leq p_i \leq P_i.$$

요약(국문초록)

최근 멀티코어 시스템에서의 실시간 태스크들은 OpenCL이나 OpenMP와 같은 병렬 프로그래밍 프레임워크를 통해 자유로운 형태로 병렬화 할 수 있게 되었다. 이러한 자유로운 병렬화는 태스크를 더 많은 수의 쓰레드로 병렬화 할 수록 각 쓰레드의 수행시간은 줄어드는 반면 전체 쓰레드 수행시간의 총량은 증가하는 트레이드오프 문제를 발생시키며, 특히 플루이드 스케줄링 기법을 사용하는 시스템에서는 시간과 밀도간의 트레이드오프가 발생하게된다. 이에 본 논문에서는 전체 시스템차원에서 트레이드오프를 활용하여 멀티코어 시스템의 스케줄 가능성을 최대화하도록 실시간 태스크의 매개변수를 결정하는 기법에 대해 제안한다. 또한 시뮬레이션과 실제 구현 실험을 통해 제안하는 기법이 시간과 밀도 사이에서 균형을 유지하며 최대 80%의 스케줄 가능성을 향상시키는 것을 보인다.

주요어 : 실시간 시스템, 멀티코어 스케줄링, 플루이드 스케줄링, 시간과 밀도간의 트레이드오프

학 번 : 2009-20755