Fixed-Priority Preemptive Multiprocessor Scheduling: To Partition or not to Partition

Björn Andersson and Jan Jonsson

Department of Computer Engineering
Chalmers University of Technology
SE–412 96 Göteborg, Sweden
{*ba,janjo*}*@ce.chalmers.se*

Abstract

Traditional multiprocessor real-time scheduling partitions a task set and applies uniprocessor scheduling on each processor. For architectures where the penalty of migration is low, such as uniform-memory access shared-memory multiprocessors, the non-partitioned method becomes a viable alternative. By allowing a task to resume on another processor than the task was preempted on, some task sets can be scheduled where the partitioned method fails.

We address fixed-priority scheduling of periodically arriving tasks on m equally powerful processors having a non-partitioned ready queue. We propose a new priority-assignment scheme for the non-partitioned method. Using an extensive simulation study, we show that the priority-assignment scheme has equivalent performance to the best existing partitioning algorithms, and outperforms existing fixed-priority assignment schemes for the non-partitioned method. We also propose a dispatcher for the non-partitioned method which reduces the number of preemptions to levels below the best partitioning schemes.

1 Introduction

Shared-memory multiprocessor systems have recently made the transition from being resources dedicated for computing-intensive calculations to common-place general-purpose computing facilities. The main reason for this is the increasing commercial availability of such systems. The significant advances in design methods for parallel architectures have resulted in very competitive cost–performance ratios for off-the-shelf multiprocessor systems. Another important factor that has increased the availability of these systems is that they have become relatively easy to program.

Based on current trends [1] one can foresee an increasing demand for computing power in modern real-time applications such as multimedia and virtual-reality servers. Shared-memory multiprocessors constitute a viable remedy for meeting this demand, and their availability thus paves the way for cost-effective, high-performance real-time systems. Naturally, this new application domain introduces a new intriguing research problem of how to take advantage of the available processing power in a multiprocessor system while at the same time account for the real-time constraints of the application.

This paper contributes to solving this problem by addressing the issue of how to utilize processors to execute a set of application tasks in a dynamic operating environment with frequent mode changes. By mode changes we mean that the characteristics of the entire task set changes at certain points in time, for example, because new tasks enter or leave the system (dynamically arriving events), or because the existing tasks need to be re-scheduled with new real-time constraints (QoS negotiation). In particular, this paper elaborates on the problem to decide whether a *partitioned* (all instances of a task should be executed on the same processor) or *non-partitioned* (execution of a task is allowed to be preempted and resume on another processor) method should be used to schedule tasks to processors at run-time. The relevance of this problem is motivated by the fact that many multiprocessor versions of modern operating systems (for example, Solaris or Windows NT) today offer run-time support for both the partitioned and the non-partitioned method.

We study the addressed problem in the context of *preemptive, fixed-priority scheduling*. The reasons for this are the following. First, the fixed-priority scheduling policy is considered to be the most mature (in terms of theoretical framework) of all priority-based scheduling disciplines. As a consequence thereof, most (if not all) existing task partitioning schemes for multiprocessor real-time systems are based on a fixed-priority scheme. Second, most modern operating systems provide support for fixed-priority scheduling as part of their standard configuration, which makes it possible to implement real-time scheduling policies on

these systems.

It has long been claimed by real-time researchers [2, 3, 4, 5, 6] that tasks should be partitioned. The intuition behind this recommendation has been that a partitioning method (i) allows for the use of well-established uniprocessor scheduling techniques and (ii) prevents task sets to be unschedulable with a low utilization. In this paper, we show that the decision of whether to partition or not cannot solely be based on such intuition. In fact, we demonstrate in this paper that the partitioned method is *not the best approach to use in a dynamic operating environment*. To this end, we make two main research contributions.

C1. We propose a new fixed-priority scheme for the non-partitioned method which circumvents many of the problems identified with traditional fixed-priority schemes. We evaluate this new priority-assignment scheme together with the rate-monotonic scheme for the non-partitioned method and compare their performance with that of a set of existing bin-packing-based partitioning algorithms. The evaluation indicates that the new scheme clearly outperforms the non-partitioned rate-monotonic scheme and provides performance at least as good as the best bin-packing-based partitioning algorithms.

C2. We propose a dispatcher for the non-partitioned method which reduces the number of preemptions. The dispatcher combined with our proposed priority-assignment scheme causes the number of preemptions to be less than the number of preemptions generated by the best partitioning schemes.

We evaluate the performance of the scheduling algorithms for multiprocessor real-time systems in the context of *resource-limited* systems where the number of processors is fixed, and use as our performance metrics the success ratio and least system utilization when scheduling a population of randomly-generated task sets. Since existing partitioning algorithms have been proposed to be applied in a slightly different context — minimizing the number of used processors in a system with an unlimited amount of processors — their actual performance on a real multiprocessor system has not been clear.

The rest of this paper is organized as follows. In Section 2, we define our task model and review related work in fixed-priority multiprocessor scheduling. We then present the new priority-assignment scheme in Section 3, and evaluate its performance in Section 4. In Section 5, we propose and evaluate the new context-switch-aware dispatcher. We conclude the paper with a discussion in Section 6 and summarize our contributions in Section 7.

2 Background

We consider a task set $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of n independent, periodically-arriving real-time tasks. The characteristics of each task $\tau_i \in \tau$ is described by the pair (T_i, C_i). A task arrives periodically with a period of T_i and with a constant execution time of C_i. Each task has a prescribed deadline, which is the time of the next arrival of the task.

The *utilization* u_i of a task τ_i is $u_i = C_i/T_i$, that is, the ratio of the task's execution time to its period. The utilization U of a task set is the sum of the utilizations of the tasks belonging to that task set, that is, $U = \sum_i C_i/T_i$. Since we consider scheduling on a multiprocessor system, the utilization is not always indicative on the load of the system. This is because the original definition of utilization is a property of the task set only, and does not consider the number of processors. To also reflect the amount of processing capability available, we introduce the concept of *system utilization*, U_s, for a task set on m processors, which is the average utilization of each processor, that is, $U_s = U/m$.

We consider a multiprocessor system with m equally powerful processors. The system uses a run-time mechanism where each task is assigned a unique and fixed priority. There is a ready queue which stores tasks that currently do not execute but are permitted to execute. As soon as a processor becomes idle or a task arrives, a dispatcher selects the next task (the task with the highest priority) in the ready queue and schedules it on a suitable processor.

Recall that the focus of this paper is on preemptive scheduling on a multiprocessor architecture. Multiprocessor real-time scheduling differs from uniprocessor real-time scheduling in that we need to determine not only when a task should execute, but also on which processor to execute. In preemptive uniprocessor scheduling, a task can be preempted by another task, and resume its execution later at a different time on the same processor. In preemptive multiprocessor real-time scheduling, a task that is preempted by another task still needs to resume its execution later at a different time, but the task may resume its execution on a different processor. Based on how the system wants to resume a task's execution, two fundamentally different methods can be used to implement preemptive multiprocessor scheduling, namely, the partitioned method and the non-partitioned method[1].

With the partitioned method, the tasks in the task set are divided in such a way that a task can only execute on one processor. In this case, each processor has its own ready queue and tasks are not allowed to migrate between processors. With the non-partitioned method, all tasks reside in a global ready queue and can be dispatched to any processor. After being preempted, the task can resume its execution on any processor. The principle for the partitioned method and

[1] Some authors refer to the non-partitioned method as "dynamic binding" or "global scheduling".

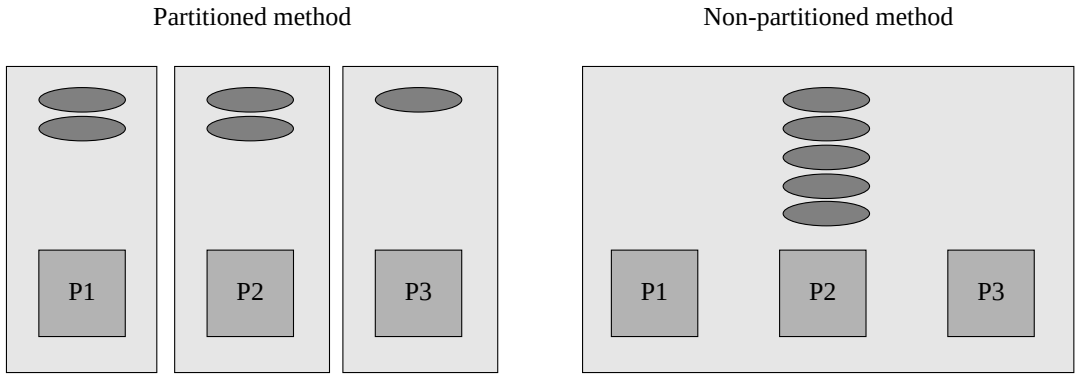

Figure 1: With the partitioned method a task can only execute on one processor. With the non-partitioned method a task can execute on any processor.

the non-partitioned method is illustrated in Figure 1.

For fixed-priority scheduling of periodically-arriving tasks on a multiprocessor system, both the partitioned method and the non-partitioned method have been addressed in previous research. Important properties of the real-time multiprocessor scheduling problem were presented in a seminal paper by Leung and Whitehead [7]. For example, they showed that the problem of deciding if a task set is schedulable (that is, all tasks will meet their deadlines at run-time) is NP-hard for both the partitioned method and the non-partitioned method. They also observed that the two methods are not comparable in effectiveness in the sense that there are task sets which are schedulable with an optimal priority assignment with the non-partitioned method, but cannot be scheduled with an optimal partitioning algorithm and conversely.

When comparing the partitioned method and the non-partitioned method, other researchers have listed the following disadvantages for the non-partitioned method. First, a task set may not be schedulable on multiple processors even though it has a system utilization that approaches zero [2, 3, 4, 5]. Second, the plethora of well-known techniques for uniprocessor scheduling algorithms (such as shared resource protocols), cannot be used [8]. Third, the run-time overhead is greater because the assignment of a processor to a task needs to be done each time the task arrives or resumes its execution after being preempted [9].

Among the two methods, the partitioned method has received the most attention in the research literature. The main reason for this is probably that the partitioned method can easily be used to guarantee run-time performance (in terms of schedulability). By using a uniprocessor schedulability test as the admission condition when adding a new task to a processor, all tasks will meet their deadlines at run-time. Now, recall that the partitioned method requires that the task set has been divided into several partitions, each having its own dedicated processor. Since an optimal solution to the problem of partitioning the tasks is computationally intractable, many heuristics for partitioning have been proposed, a majority of which are versions of the bin-packing algorithm[2] [2, 3, 4, 5, 8, 10, 11, 6, 9]. All of these bin-packing-based partitioning algorithms provide performance guarantees, they all exhibit fairly good average-case performance, and they can all be applied in polynomial time (using sufficient schedulability tests).

The non-partitioned method has received considerably less attention, mainly because of the disadvantages listed above, but also because of the following two reasons. First, no effective optimal priority-assignment scheme has been found for the non-partitioned method. It is a well-known fact that the rate-monotonic priority assignment scheme proposed by Liu and Layland [12] is an optimal priority assignment for the uniprocessor case. Unfortunately, it has been shown that the rate-monotonic priority-assignment scheme is no longer optimal for the non-partitioned method [3, 7]. Second, no effective schedulability tests exist for the non-partitioned method. Recall that, for the partitioned method, existing uniprocessor schedulability tests can be used. The only known exact (necessary and sufficient) schedulability test for the non-partitioned method has an exponential time-complexity [13]. Liu has proposed a sufficient schedulability test for the rate-monotonic priority-assignment scheme [14], which, unfortunately, becomes very pessimistic when the number of tasks increases. Recently, another sufficient schedulability test for the rate-monotonic priority-assignment scheme was proposed inde-

[2]The bin-packing algorithm works as follows: (1) sort the tasks according to some criterion; (2) select the first task and an arbitrary processor; (3) attempt to assign the selected task to the selected processor by applying a schedulability test for the processor; (4) if the schedulability test fails, select the next available processor; if it succeeds, select the next task; (5) goto step 3.

pendently by three research groups [15, 16, 17]. Unfortunately, this schedulability test has the disadvantage of becoming pessimistic for task sets running on a large number of processors [15, 16]. These schedulability tests all have a polynomial (for [17], a pseudo-polynomial) time-complexity. These research groups also found that the condition for a critical instant in uniprocessor scheduling cannot be applied in multiprocessor scheduling. This is an important observation since it indicates that completely new approaches must be used to devise a schedulability test for the non-partitioned method.

Since there currently exist no efficient[3] schedulability tests for non-partitioned method, it is tempting to believe that the non-partitioned method is inappropriate for real-time systems. However, our study of the non-partitioned method for real-time systems is motivated by the following two reasons. First, many real-time systems do not rely on a schedulability test, for example, those which employ feedback control scheduling [18, 19, 20] or QoS negotiation techniques [21]. Second, even if a real-time system does rely on a schedulability test, we believe that in developing better priority-assignment schemes, we may pave the way for effective schedulability tests in the future.

To focus on the core problem, we make the following assumptions in the remainder of this paper:

A1. Tasks require no other resources than the processors. This implies that tasks do not contend for an interconnect, such as a bus, or critical sections.

A2. Tasks are synchronous in the sense that their initial arrivals occur at the same instant of time and then the tasks arrive periodically.

A3. A task can always be preempted. In practice, though, one has to be aware of the fact that operating systems typically use tick-driven scheduling where the ready queue can only be inspected at a maximum rate.

A4. The cost of preemption is zero. We will use this assumption even if a task is resumed on another processor than the task was originally preempted on. In a real system, though, the cost of context switch is typically in the order of $100\ \mu s$ [22].

3 Priority assignment

One of our major contributions in this paper is a new priority-assignment scheme for the non-partitioned method. In this section, we begin by recapitulate some known results concerning priority-assignment schemes for multiprocessor scheduling. We then proceed to motivate our new priority-assignment scheme and discuss how it can be optimized for the non-partitioned method.

While the partitioned method relies on well-known optimal uniprocessor priority-assignment schemes, such as the rate-monotonic scheme, it is not clear as to what priority-assignment scheme should be used for the non-partitioned method. To that end, Leung [13], and later Sáez *et al.* [9], evaluated dynamic-priority schemes and showed that the least-laxity-first strategy performs the best for the non-partitioned case. In the case of fixed-priority scheduling, adopting the idea used by Audsley [23] and Baruah [24] (testing for lowest priority viability) is unsuitable because existing schedulability tests for non-partitioned fixed-priority scheduling are too pessimistic. The only known results report that the rate-monotonic priority assignment scheme does not work well for the non-partitioned method. This is because, for rate-monotonic scheduling on a uniprocessor, a sufficient (but not necessary) condition for schedulability of a task set is that the utilization of the task set is less than or equal to $\ln 2$. For non-partitioned multiprocessor scheduling using the rate-monotonic priority assignment, no such property exists. Originally presented by Dhall [2, 3] (and later repeated in, for example, [4, 5, 7]), the basic argument is as follows. Assume that the task set $(T_1 = 1, C_1 = 2\epsilon), (T_2 = 1, C_2 = 2\epsilon), \ldots, (T_m = 1, C_m = 2\epsilon), (T_{m+1} = 1 + \epsilon, C_{m+1} = 1)$ should be scheduled using the rate-monotonic priority assignment scheme on m processors. The situation for $m = 3$ is shown in Figure 2. In this case, τ_{m+1} will have the lowest priority and will only be scheduled after all other tasks have executed in parallel. The task set is unschedulable and as $\epsilon \to 0$, the utilization becomes $U = 1$ no matter how many processors are used. Another way of formulating this is that the system utilization $U_s = U/m$ will decrease towards zero as m increases. We will refer to this observation as *Dhall's effect*.

Now, we observe that if τ_{m+1} could somehow be assigned a higher priority, the given task set would be schedulable. To see this, simply assign task priorities according to the difference between period and execution time of each task. Then, τ_{m+1} would be assigned the highest priority, and the task set would still be schedulable even if the execution time of any single task would increase slightly.

The fundamental problem demonstrated with the example above can be summarized as follows. In fixed-priority scheduling using the rate-monotonic scheme, many tasks with short period, but with relatively (with respect to period) short execution time, can block the available computing resources so that tasks with longer period, but with relatively long execution time, will miss their deadlines. This indicates that a more appropriate strategy to assign priorities for the non-partitioned method would be to reflect both time criticality and resource demands of each task.

[3] By "efficient" we mean that the schedulability test can be done in polynomial time (as a function of tasks, not task invocations) and that the schedulability test always deems task sets as schedulable if the task set has a utilization which is less than a fixed ratio of the number of processors.

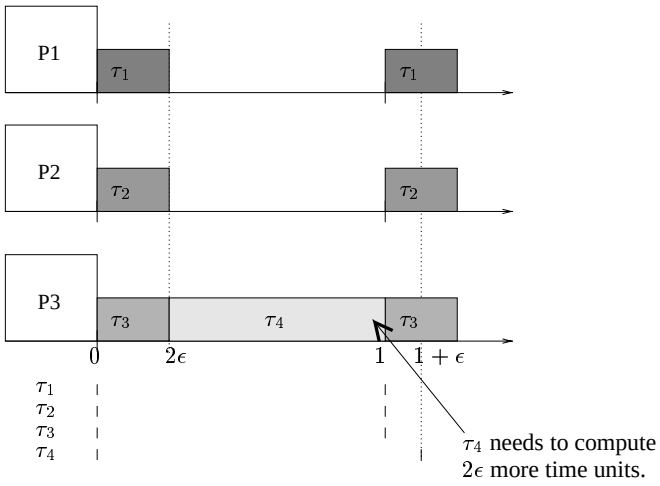

Figure 2: If the rate-monotonic priority assignment for the non-partitioned method is used, there are unschedulable task sets with a low utilization.

An example of such a strategy would be one that prioritizes tasks according to the difference between the period of a task and its execution time[4]. Based on this, we now propose a new priority assignment scheme, called *TkC*, where the priority of task τ_i is assigned according to the weighted difference between its period and its execution time, that is, $T_i - k \cdot C_i$, where k is a global *slack factor*. Note that, by using this model, we can represent two traditionally used priority assignment schemes, namely the rate monotonic (when $k = 0$) and the slack monotonic (when $k = 1$) schemes. Below, we will reason about what k should be selected to contribute to the best performance for the non-partitioned method.

3.1 Reasoning about k

It is clear from the discussion above that, in order to escape from Dhall's effect, we should select $k > 0$[5]. The remaining question is then what values of k yield the best performance. Even for all $0 < k \leq 1$, something similar to Dhall's effect can occur. Assume that the task set $(T_1 = 1, C_1 = \frac{1}{L}), (T_2 = 1, C_2 = \frac{1}{L}), \ldots, (T_m = 1, C_m = \frac{1}{L}), (T_{m+1} = L^2 + \frac{1}{L}, C_{m+1} = L^2 - \frac{1}{2} \cdot L)$ should be scheduled using the TkC priority-assignment scheme with $0 < k \leq 1$ on m processors. The situation for $m = 3$ is shown in Figure 3. Now, as $L \to \infty$, the task set is unschedulable with a utilization $U = 1$. On the other hand, selecting too large a k is a bad idea since task priorities will then be selected such that the tasks with the longest execution time obtains the highest priority. Assume that the task set $(T_1 = 1, C_1 = \epsilon), (T_2 = 1, C_2 = \epsilon), \ldots, (T_m = 1, C_m = \epsilon), (T_{m+1} = \epsilon, C_{m+1} = \epsilon^2)$ should be scheduled using the TkC priority-assignment scheme with $k \to \infty$ on m processors. The situation for $m = 3$ is illustrated in Figure 4. Now, as $\epsilon \to 0$, this task set is unschedulable with a utilization $U = 0$ no matter how many processors are used. Of course, the system utilization $U_s = U/m$ will also decrease toward 0 as m increases. Consequently, this effect is even worse than Dhall's effect.

In conclusion, we observe that k should be greater than 1, but not too large. Lauzac *et al.* [16] evaluated the performance of the non-partitioned fixed-priority scheduling. Unfortunately, they only considered $k = 0$ (that is, the rate-monotonic scheme), which clearly is not the best k. In Section 4.2, we simulate scheduling with different values of k to determine which k is best. In Section 4.3, we show that the non-partitioned method using the TkC priority-assignment scheme outperforms all existing fixed-priority schemes for the non-partitioned method and performs at least as good as the best practical online partitioning schemes.

4 Performance evaluation

In this section, we will conduct a performance evaluation of the non-partitioned and partitioned method. Our evaluation methodology is based on simulation experiments using randomly-generated task sets. The reasons for this are that (i) simulation using synthetic task sets more easily reveal the average-case performance and robustness of a scheduling algorithm than can be achieved by scheduling a single application benchmark, and (ii) with the use of simulation we can compare the best k from simulation with the derived interval of k from our reasoning in Section 3.1.

[4]This is what is typically referred to as the *slack* of the task.

[5]Selecting $k < 0$ can also cause Dhall's effect.

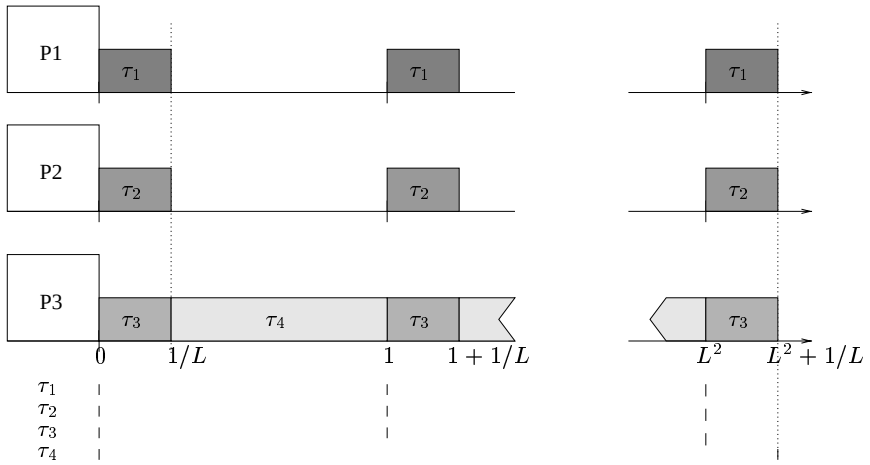

Figure 3: If the TkC priority assignment ($0 < k \leq 1$) for the non-partitioned method is used, there are unschedulable task sets with a low utilization.

4.1 Experimental setup

Below, we describe our performance evaluation. Unless otherwise stated, we conduct the performance evaluation using the following experimental setup.

Task sets are randomly generated and their scheduling is simulated with the respective method on four processors. The number of tasks, n, follows a uniform distribution with an expected value of $E[n] = 8$ and a standard deviation of $0.5\,E[n]$. The period, T_i, of a task τ_i is taken from a set $\{100, 200, 300, \ldots, 1600\}$, each number having an equal probability of being selected. The utilization, u_i, of a task τ_i follows a normal distribution with an expected value of $E[u_i] = 0.5$ and a standard deviation of $stddev[u_i] = 0.4$. If $u_i < 0$ or $u_i > 1$ then a new u_i is generated. The execution time, C_i, of a task τ_i is computed from the generated utilization of the task, and rounded down to the next lower integer. If the execution time becomes zero, then the task is generated again.

The priorities of tasks are assigned with the respective priority-assignment scheme, and, if applicable, tasks are partitioned and assigned to processors. All tasks arrive at time 0 and scheduling is simulated during $\mathrm{lcm}(T_1, \ldots, T_n)$. The reason for selecting small values of m and $E[n]$ is that $\mathrm{lcm}(T_1, \ldots, T_n)$ grows rapidly as n is increased, causing simulations to take too long time.

Two performance metrics are used for evaluating the simulation experiments, namely *success ratio* and *least system utilization*. The success ratio is the fraction of all generated task sets that are schedulable with respect to an algorithm[6] . The success ratio is a recognized performance metric in the real-time community and measures the probability that an arbitrary task set is schedulable with respect to a certain algorithm. The least system utilization is defined as the minimum of the system utilization of all the task sets that we simulated and found to be unschedulable. The least system utilization is primarily used to see if an algorithm is likely to suffer from Dhall's effect and similar effects, that is, if a task set will be unschedulable even for a low utilization. The plots showing the least system utilization will be less smooth than those from the simulations of success ratio. This is because the least system utilization reflects a minimum of numbers rather than an average.

Two non-partitioned priority-assignment schemes are evaluated, namely *RM* which is the rate-monotonic priority-assignment scheme, and *Tk1.1C* which is the TkC priority-assignment scheme with $k = 1.1$ (in Section 4.2, we will show that this is indeed the best k). Four bin-packing-based partitioning schemes are studied, namely RRM-BF [10], RM-FFDU [11], RMGT [8], and R-BOUND-MP [6][7]. The reason for selecting these algorithms is that we have found that they are the partitioning algorithms which provide the best performance in our experimental environment. Since partitioning schemes use a schedulability test as a part of the partitioning, the success ratio is here also a guarantee ratio. Note that this property does not apply for scheduling algorithms using the non-partitioned method. We have also evaluated a hybrid partitioned/non-partitioned algorithm, which we will call *RM-FFDU+Tk1.1C*. The reason for considering a hybrid solution is that we may be able to increase

[6]Since the number of task sets for each point differs between plots (5 000 000 in Figure 5 and 2 000 000 in Figures 6 and 7, we obtain different estimates of the error. With 95% confidence, we obtain errors that are less than 0.0006 for Figure 5 and 0.0014 for Figures 6 and 7.

[7]To ensure correct operation of our simulator, we repeated previous experiments [10, pages 235–236] and [8, pages 1440–1441] with identical results, and [11, pages 36–37] with similar results.

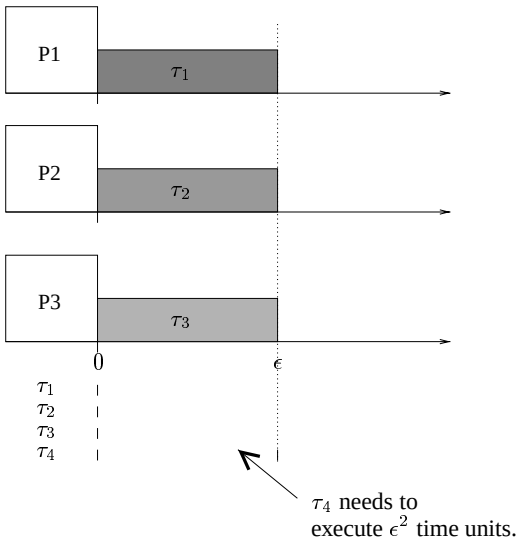

Figure 4: If the TkC priority assignment ($k \to \infty$) for the non-partitioned method is used, there are unschedulable task sets with a low utilization.

processor utilization with the use of non-partitioned tasks, without jeopardizing the guarantees given to partitioned tasks. The RM-FFDU+Tk1.1C scheme operates in the following manner. First, as many tasks as possible are partitioned with RM-FFDU on the given number of processors. Then, the remaining tasks (if any) are assigned priorities according to the Tk1.1C priority-assignment scheme. In order not to jeopardize the partitioned tasks, the priorities of the non-partitioned tasks are set to be lower than the priority of any partitioned task.

Below, we make two major experiments. First, we will vary k to find the best k and compare the results with our reasoning in Section 3.1. Second, we will compare the performance of the Tk1.1C method with that of the partitioning algorithms.

4.2 Finding the best k

Figure 5 shows the success ratio and the least system utilization as a function of the slack factor k. From the plot we make two important observations. First, the least system utilization and the success ratio are correlated. This implies that selecting an appropriate k to maximize the least system utilization will provide a good (though not necessary optimal) k for the success ratio. Second, choosing $k = 1.1$ provides the best success ratio. This corroborates that our reasoning in Section 3.1 was accurate.

4.3 Performance comparison

In Figures 6 and 7, we show the performance as a function of the number of processors. The success ratio associated with the Tk1.1C priority-assignment scheme is higher than that of any other scheduling algorithm. This clearly indicates that our new parametrized priority-assignment scheme works as predicted. In particular, we notice that the traditional RM scheme is outperformed by more than 10 percentage units. The least system utilization associated with Tk1.1C is equivalent to that of the best existing partitioning algorithms. Note how the least system utilization of RM decreases considerably as the number of processors increases, simply because of Dhall's effect. Also observe that Tk1.1C does not appear to suffer from Dhall's effect.

In a complementary study [25], we have also investigated the robustness of the scheduling algorithms by varying the other parameters ($E[n]$, $E[u_i]$ and $stddev[u_i]$) that determine the task set. Here, we observed that the relative ranking of the algorithms does not change; the Tk1.1C priority-assignment scheme still offers the highest success ratio. Furthermore, Tk1.1C continues to provide a least utilization which is equivalent to that of the best existing partitioning algorithms. This further supports our hypothesis that Dhall's effect does not occur for Tk1.1C.

From the plots in Figures 6 and 7, we can also make the following two observations. First, the hybrid partitioning/non-partitioning scheme consistently outperforms the corresponding partitioning schemes. This indicates that such a hybrid scheme is a viable alternative to use in multiprocessor systems that mixes real-time tasks of different criticality. Second, the success ratio of RM is not as bad as suggested by previous studies [2, 3, 4, 5, 7]. The reason for this is of course that Dhall's effect, although it exists, does not occur frequently. This observation corroborates a recent study [16].

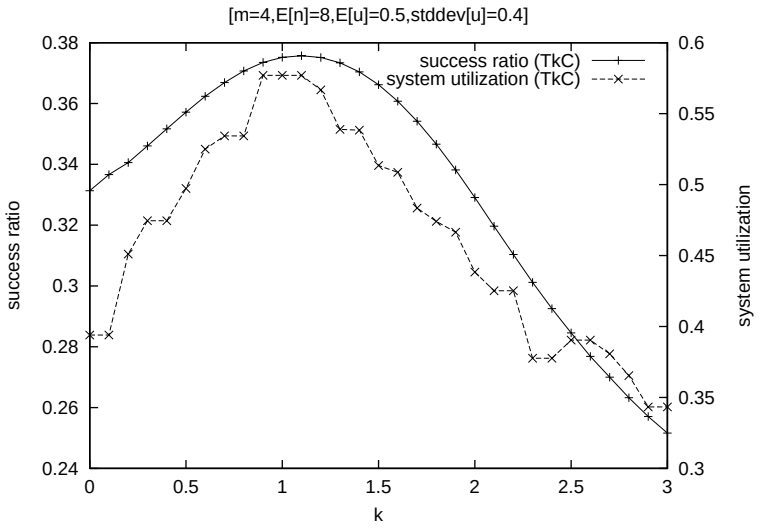

Figure 5: Success ratio and least system utilization as a function of the slack factor k.

For all partitioning algorithms, sufficient schedulability tests were originally proposed to be used. Now, if we instead use a schedulability test based on response-time analysis for the partitioning schemes, the success ratio can be expected to increase, albeit at the expense of a significant increase in computational complexity. Recall that response-time analysis has pseudo-polynomial time complexity, while sufficient schedulability tests typically have polynomial time-complexity. We have simulated the partitioning algorithms using response-time analysis [25] and made the following observation. The FFDU algorithm now occasionally provides a slightly higher success ratio than Tk1.1C, while other partitioning algorithms did not improve their performance that much, and still had a lower success ratio than the Tk1.1C. Note that this means that, even if the best partitioning algorithm (R-BOUND-MP) were to use response-time analysis, it would still perform worse than Tk1.1C. This should not come as a surprise since, for R-BOUND-MP, the performance bottleneck is the partitioning and not the schedulability test [6].

5 Context switches

It is tempting to believe that the non-partitioned method causes a larger amount of (and more costly) context switches than the partitioned method. In this section, we will propose a dispatcher for the non-partitioned method that reduces the number of context switches by analyzing the current state of the scheduler. We will also compare the number of context switches generated by the best non-partitioned method using our dispatcher with that of the best partitioning scheme.

5.1 Dispatcher

The non-partitioned method using fixed-priority scheduling does only require that, at each instant, the m tasks with the highest priorities are executed; it does not require a task to run on a specific processor. Hence, as long as the cost for a context switch is zero, the task-to-processor assignment at each instant does not affect schedulability. However, on real computers the time required for a context switch is non-negligible. If the task-to-processor assignment is selected arbitrarily, it could happen (in theory) that all m highest-priority tasks execute on another processor than they did the last time they were dispatched, even if these m tasks were the ones that executed last. Hence, to reduce the risk of unnecessary context switches, we need a dispatcher that not only selects the m highest-priority tasks, but also selects a task-to-processor assignment such that the number of context switches is minimized.

We now propose a heuristic for the task-to-processor assignment that will reduce the number of context switches for the non-partitioned method. The heuristic is intended to be used as a subroutine in the dispatcher in an operating system. The basic idea of the task-to-processor assignment algorithm is to determine which of the tasks that must execute now (that is, have the highest priority) have recently executed, and then try to execute those tasks on the same processor as their previous execution. The algorithm for this is described in Algorithm 1.

5.2 Comparison of the number of context switches

Two terms contribute to the time for context switches: (i) operating system overhead, including register save and restore and time to acquire a lock for the ready queue, and (ii)

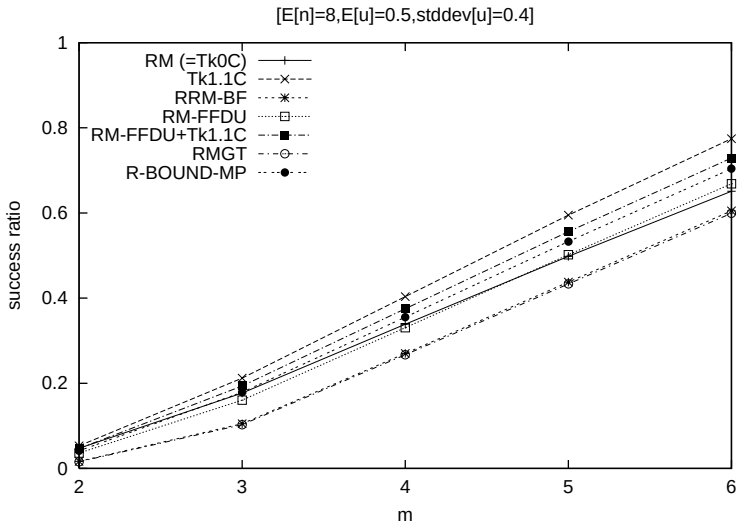

Figure 6: Success ratio as a function of the number of processors.

Algorithm 1 Task-to-processor assignment algorithm for the non-partitioned method.

Input: Let τ_{before} be the set of tasks that just executed on the m processors. On each processor p_i, a (possibly non-existing) task $\tau_{ij,before}$ executed. Let $\tau_{highest}$ be the set of tasks that are in the ready queue and has the highest priority.

Output: On each processor p_i, a (possibly non-existing) task $\tau_{ij,after}$ should execute.

$E = \{p_i : (\tau_{ij,before} \neq \text{non} - \text{existing}) \wedge (\tau_{ij,before} \in \tau_{highest})\}$
for each $p_i \in E$
    remove $\tau_{ij,before}$ from $\tau_{highest}$
    $\tau_{ij,after} \leftarrow \tau_{ij,before}$
for each $p_i \notin E$
  if $\tau_{highest} \neq \emptyset$
    select an arbitrary τ_j from $\tau_{highest}$
    remove τ_j from $\tau_{highest}$
    $\tau_{ij,after} \leftarrow \tau_j$
  else
    $\tau_{ij,after} \leftarrow \text{non} - \text{existing}$

an increase in execuction time due to cache reloading. We assume that the penalty for cache reloading when a task is preempted, and resumes on another processor, is the same as if the task would resume on the same processor after being preempted. If the preempting task (that is, the new task) has a large working set, this should be a reasonable assumption. We also assume that the operating system overhead to resume a preempted task on the same processor is the same as when the task resumes on another processor. Under these assumptions, the cost of a context switch is the same for the partitioned method and the non-partitioned method. Hence, we can count the number of preemptions during an interval of $\text{lcm}(T_1, T_2, \ldots, T_n)$ and use that as a measure of the impact of context switches on the performance of the partitioned method and the non-partitioned method. Note, however, that this does not measure the impact of context switches on schedulability, but gives an indication on the amount of overhead introduced.

To reveal which of the two methods (partitioned or non-partitioned) that suffers the highest penalty due to context switches, we have simulated the scheduling of randomly-generated task sets and counted the number of preemptions. We simulate scheduling using Tk1.1C and R-BOUND-MP because they are the best (as demonstrated in Section 4.3) schemes for the non-partitioned method and the partitioned method, respectively. We use the same experimental setup as described in Section 4.1. We varied the number of tasks and counted the number of preemptions only for those task sets for which both Tk1.1C and R-BOUND-MP were schedulable. Since different task sets have different $\text{lcm}(T_1, \ldots, T_n)$, the impact of task sets with large $\text{lcm}(T_1, \ldots, T_n)$ will be too large. To make each task set equally important, we select to use *preemption density* as a measure of the context-switch overhead. Preemption density of a task set is defined as the number of preemptions during a $\text{lcm}(T_1, \ldots, T_n)$ divided by the $\text{lcm}(T_1, \ldots, T_n)$ itself. We then take the average of the preemption density over a set of 100 000 simulated task sets[8].

The results from the simulations are shown in Figure 8. We observe that, on average, the preemption density for the best non-partitioned method (Tk1.1C) is lower than the preemption density for the best partitioned method (R-BOUND-MP). The reason for this is that, for the partitioned

[8]Hence we obtain an error of 0.0004 with 95% confidence.

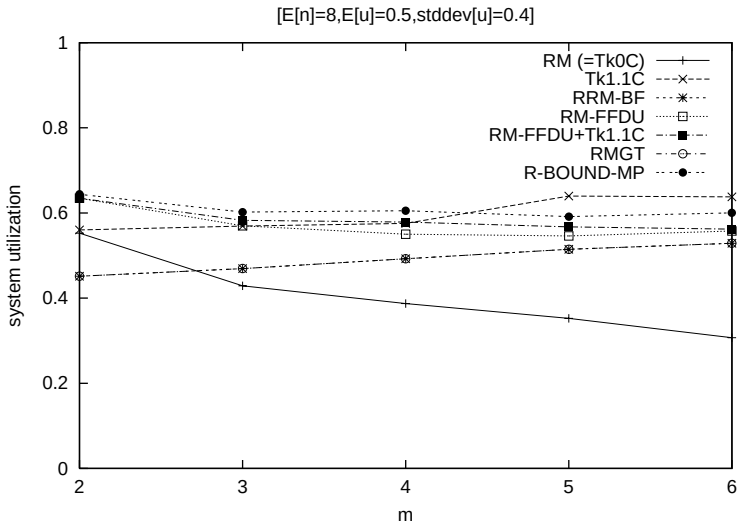

Figure 7: Least system utilization as a function of the number of processors.

method, an arriving higher-priority task must preempt a lower-priority task. With the non-partitioned method, a higher priority task can sometimes execute on another idle processor, thereby avoiding a preemption. Figure 9 illustrates a situation where the non-partitioned method with a context-switch-aware dispatcher causes the number of preemptions to be less than the number of context switches for a partitioned method.

The reasoning and evaluations in this section indicate that the cost of context switches for the non-partitioned method can be made significantly less than for the partitioned method. This means that it is should be possible to demonstrate even more significant performance gain in terms of schedulability for the non-partitioned method relative to the partitioned method on a real multiprocessor system.

6 Discussion

The simulation results reported in Section 4.3 indicate that the non-partitioned method in fact performs better than its reputation. Besides the advantages demonstrated through our experimental studies, there are also other benefits in using the non-partitioned method. Below, we discuss some of those benefits.

The non-partitioned method is the best way of maximizing the resource utilization when a task's actual execution time is much lower than its stated worst-case execution time [16]. This situation can occur when the execution time of a task depends highly on user input or sensor values. Since the partitioned method is guided by the worst-case excution time during the partitioning decisions, there is a risk that the actual resource usage will be lower than anticipated, and thus wasted if no dynamic exploitation of the spare capacity is made. Our suggested hybrid approach offers one solution to exploiting system resources effectively, while at the same time providing guarantees for those tasks that require so. The hybrid solution proposed in this paper applies the partitioned method to the task set until all processors have been filled. The remaining tasks are then scheduled using the non-partitioned approach. An alternativ approach would be to partition only the tasks that have strict (that is, hard) real-time constraints, and then let the tasks with less strict constraints be scheduled by the non-partitioned method. Since it is likely that shared-memory multiprocessors will be used to schedule mostly tasks of the latter type, we expect even better performance (in terms of success ratio) for the hybrid solution.

The non-partitioned method can perform mode changes faster than the partitioned method since tasks are not assigned to dedicated processors. For the partitioned method it may be necessary to repartition the task set during the mode change, something which significantly decreases the number of instants in time that a mode change can take place at. In a complementary study, we have observed pairs of task sets between which a mode change can only be allowed to take place at a few instants in time, otherwise deadlines will be missed. The lack of capability to perform fast mode changes limits the degree of flexibility in the system, which is a serious drawback when working with dynamic application environments.

As shown in this paper, the non-partitioned method can be extended to account for processor affinity with the aid of our proposed context-switch-aware dispatcher. The net result of this extension is that the non-partitioned method incurs fewer context switches at run-time than the partitioned method on the average. However, it is important to real-

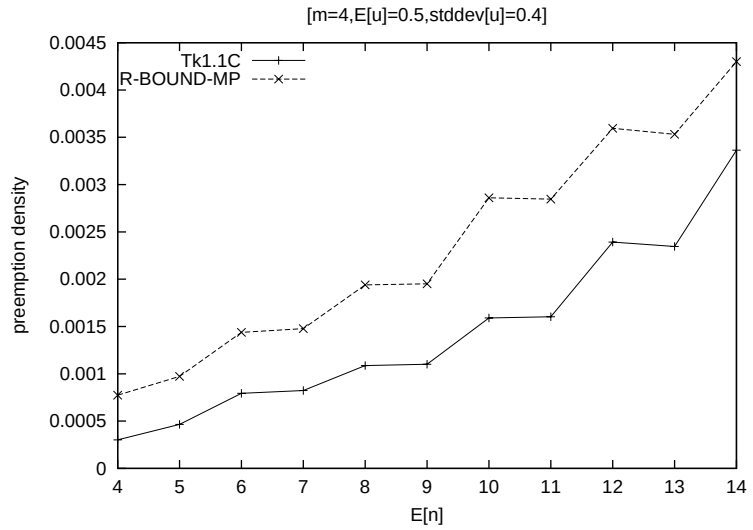

Figure 8: Preemption density as a function of the number of tasks in the system.

ize that this new dispatcher requires more synchronization between processors. The overall effect of this extra synchronization remains to be seen and is consequently a subject for future research. Also, it will be necessary to assess the real costs for context switches and cache reloading and its impact on schedulability. However, this warrants evaluation of real applications on real multiprocessor architectures, which introduces several new system parameters to consider.

Finally, it should be mentioned that the results obtained for the TkC priority-assignment scheme constitute a strong conjecture regarding the existence of a fixed-priority assignment scheme for the non-partitioned method that does not suffer from Dhall's effect. The results also indicate that there is a need to configure TkC for each application and system size. However, in order to use TkC in a system with exclusively strict real-time constraints, it is also important to find an effective schedulability test. In our future research we will therefore focus on (i) proving that TkC does in fact not suffer from Dhall's effect, (ii) constructing a (most likely, sufficient) schedulability test for TkC, and (iii) devising a method to derive the best slack factor k for any given task set and multiprocessor system.

7 Conclusions

In this paper, we have addressed the problem of scheduling tasks on a multiprocessor system with changing workloads. To that end, we have made two major contributions. First, we proposed a new fixed-priority assignment scheme that gives the non-partitioned method equal or better performance than the best partitioning schemes. Since the partitioned method can guarantee that deadlines will be met at run-time for the tasks that are partitioned, we have also evaluated a hybrid solution that combined the resource-effective characteristics of the non-partitioned method with the guarantee-based characteristics of the partitioned method. The performance of the hybrid solution was found to be at least as good as any partitioned method. Second, we proposed a context-switch-aware dispatcher for the non-partitioned method that contributes to significantly reducing the number of context switches taken at run-time. In fact, we show that the number of context switches taken is less than that of the partitioned method for similar task sets.

References

[1] K. Diefendorff and P. K. Dubey. How multimedia workloads will change processor design. *IEEE Computer*, 30(9):43–45, September 1997.

[2] S. Dhall. *Scheduling Periodic-Time-Critical Jobs on Single Processor and Multiprocessor Computing Systems*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champain, 1977.

[3] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, January/February 1978.

[4] S. Davari and S.K. Dhall. On a real-time task allocation problem. In *19th Annual Hawaii International Conference on System Sciences*, pages 8–10, Honolulu, Hawaii, 1985.

[5] S. Davari and S.K. Dhall. An on-line algorithm for real-time task allocation. In *Proc. of the IEEE Real-Time Systems Symposium*, volume 7, pages 194–200, New Orleans, LA, December 1986.

[6] S. Lauzac, R. Melhem, and D. Mossé. An efficient RMS admission control and its application to multiprocessor scheduling. In *Proc. of the IEEE Int'l Parallel Processing Symposium*, pages 511–518, Orlando, Florida, March 1998.

[7] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250, December 1982.

[8] A. Burchard, J. Liebeherr, Y. Oh, and S.H. Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Transactions on Computers*, 44(12):1429–1442, December 1995.

[9] S. Sáez, J. Vila, and A. Crespo. Using exact feasibility tests for allocating real-time tasks in multiprocessor systems. In *10th Euromicro Workshop on Real Time Systems*, pages 53–60, Berlin, Germany, June 17–19, 1998.

[10] Y. Oh and S. H. Son. Allocating fixed-priority periodic tasks on multiprocessor systems. *Real-Time Systems*, 9(3):207–239, November 1995.

[11] Y. Oh and S. H. Son. Fixed-priority scheduling of periodic tasks on multiprocessor systems. Technical Report 95-16, Department of Computer Science, University of Virginia, March 1995. Available at ftp://ftp.cs.virginia.edu/pub/techreports/CS-95-16.ps.Z.

[12] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, January 1973.

[13] J. Y.-T. Leung. A new algorithm for scheduling periodic, real-time tasks. *Algorithmica*, 4(2):209–219, 1989.

[14] C. L. Liu. Scheduling algorithms for multiprocessors in a hard real-time environment. In *JPL Space Programs Summary 37-60*, volume II, pages 28–31. 1969.

[15] B. Andersson. Adaption of time-sensitive tasks on shared memory multiprocessors: A framework suggestion. Master's thesis, Department of Computer Engineering, Chalmers University of Technology, January 1999. Available at http://www.ce.chalmers.se/staff/ba/master_thesis/ps/thesis.ps and http://www.docs.uu.se/snart/prizes.shtml#1999.

[16] S. Lauzac, R. Melhem, and D. Mossé. Comparison of global and partitioning schemes for scheduling rate monotonic tasks on a multiprocessor. In *10th Euromicro Workshop on Real Time Systems*, pages 188–195, Berlin, Germany, June 17–19, 1998.

[17] L. Lundberg. Multiprocessor scheduling of age contraint processes. In *5th International Conference on Real-Time Computing Systems and Applications*, Hiroshima, Japan, October 27–29, 1998.

[18] J. A. Stankovic, C. Lu, and S. H. Son. The case for feedback control real-time scheduling. Technical Report 98-35, Dept. of Computer Science, University of Virginia, November 1998. Available at ftp://ftp.cs.virginia.edu/pub/techreports/CS-98-35.ps.Z.

[19] J. A. Stankovic, C. Lu, S. H. Son, and G. Tao. The case for feedback control real-time scheduling. In *Proc. of the EuroMicro Conference on Real-Time Systems*, volume 11, pages 11–20, York, England, June 9–11, 1999.

[20] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Design and evaluation of a feedback control EDF scheduling algorithm. In *Proc. of the IEEE Real-Time Systems Symposium*, Phoenix, Arizona, December 1–3, 1995.

[21] S. Brandt and G. Nutt. A dynamic quality of service middleware agent for mediating application resource usage. In *Proc. of the IEEE Real-Time Systems Symposium*, volume 19, pages 307–317, Madrid, Spain, 1998.

[22] J. C. Mogul and A. Borg. Effect of context switches on cache performance. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 75–84, Santa Clara, CA USA, April 8–11, 1991.

[23] N. C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical Report YCS 164, Dept. of Computer Science, University of York, York, England Y01 5DD, December 1991.

[24] S. K. Baruah. Static-priority scheduling of recurring real-time tasks. Technical report, Department of Computer Science, The University of North Carolina at Chapel Hill, 1999. Available at http://www.cs.unc.edu/~baruah/Papers/staticdag.ps.

[25] B. Andersson and J. Jonsson. Fixed-priority preemptive scheduling: To partition or not to partition. Technical Report No. 00-1, Dept. of Computer Engineering, Chalmers University of Technology, S–412 96 Göteborg, Sweden, January 2000. Available at http://www.ce.chalmers.se/staff/ba/TR/TR-00-1.ps.

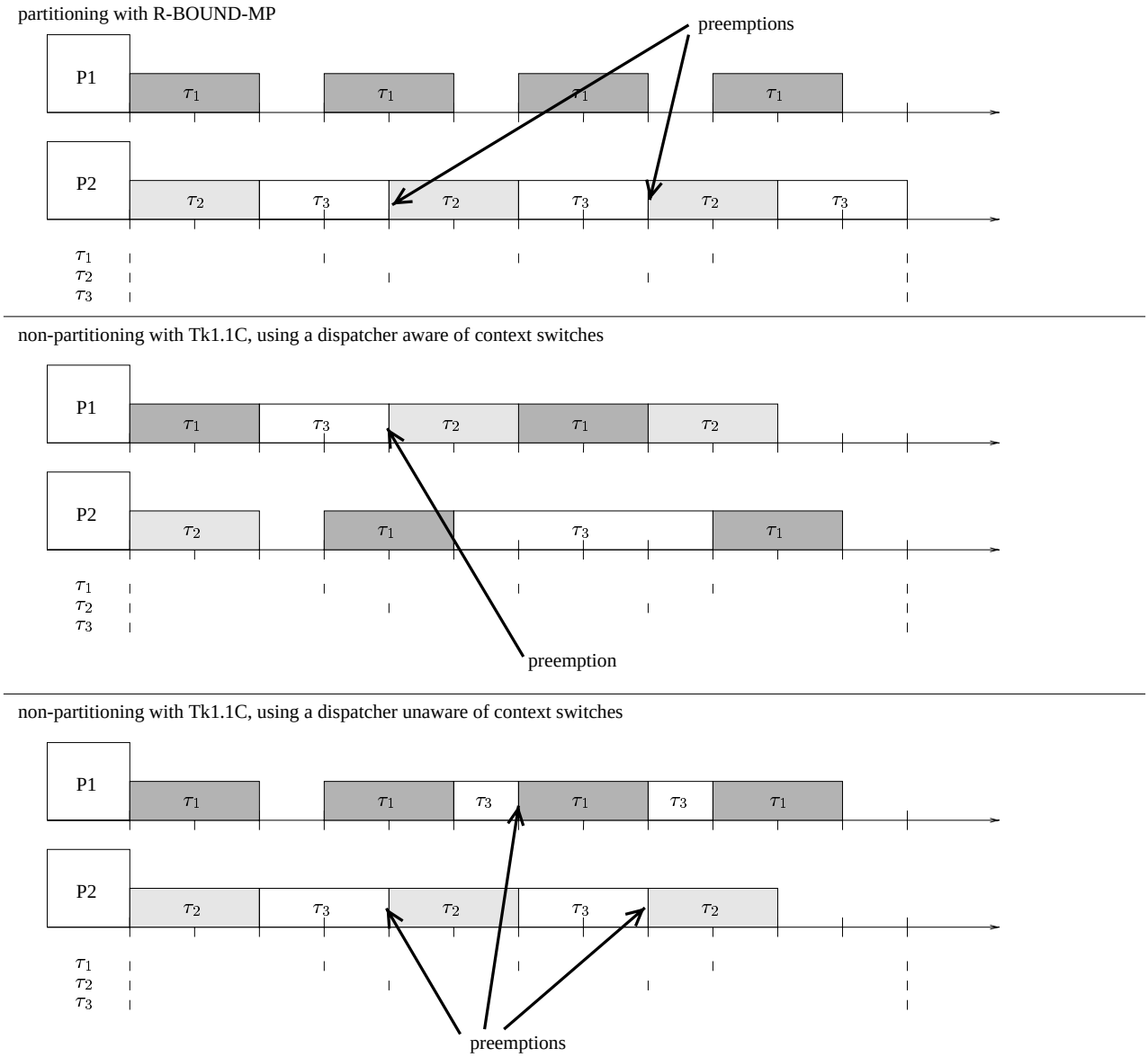

Figure 9: Preemptions for non-partitioned method and partitioned method.