# 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. 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 preemptive scheduling of periodically arriving tasks on $m$ equally powerful processors. We compare the performance of the best algorithms of the partitioned and non-partitioned method, from two different aspects. First, an average-case comparison, using an idealized architecture, shows that, if a system has a small number of processors, then the non-partitioned method offers higher performance than the partitioned method. Second, an average-case comparison, using a realistic architecture, shows that, for several combinations of preemption and migration costs, the non-partitioned method offers higher performance.*

## 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.

Fixed-priority scheduling of tasks on such systems is typically solved using one of two different methods based on how tasks are assigned to the processors at run-time. In the *partitioned* method, all instances of a task are executed on the same processor. The processor used for the execution of a task is determined before run-time by a partitioning algorithm. In the *non-partitioned* method, a task is allowed to execute on any processor, even when resuming after having been preempted. For the partitioned method, a plethora of mature algorithms and analyses have been proposed, something that has contributed to the widespread use of the method in multiprocessor-based real-time systems. In contrast, the non-partitioned method has received much less attention, mainly because it is believed to suffer from scheduling- and implementation-related shortcomings, but also because it lacks support for more advanced system models, such as the management of shared resources.

Many operating systems for shared-memory multiprocessors support fixed-priority scheduling with both the partitioned and the non-partitioned method. In order to know which of the methods should be used in practice, it is important to know their ability to schedule an arbitrary task set on a fixed number of processors. So far, the only previous evaluation of fixed-priority preemptive multiprocessor scheduling claims that the average-case performance of the non-partitioned method is lower than the average-case performance of the partitioned method [2]. It has also been shown that the worst-case performance of the non-partitioned method using the rate-monotonic priority scheme is poor [3]. We believe that these comparisons are not complete, because (i) recently a novel priority assignment for the non-partitioned method [4] was proposed which alleviates the scheduling-related shortcoming, and (ii) the previous evaluation considered the cost of preemption and migration to be zero. It is not clear how preemption and migration costs will affect performance of the partitioned and non-partitioned methods.

In this paper, we compare the novel non-partitioned priority-assignment scheme against the best partitioning algorithms. We demonstrate that the partitioned method is *not necessarily the best approach.* To this end, we make two main research contributions.

C1. We show that, on an idealized architecture, the non-partitioned method offers higher average-case performance than the partitioned method. We also show that, even if a necessary and sufficient schedulability test is used for the partitioning algorithms, then the best partitioned method performs only slightly better than the non-partitioned method.

C2. We show that, on a realistic architecture, the non-partitioned method can provide higher performance than the partitioned method when both an improved dispatcher is used and the additional cost of a migration is no greater than the cost of a preemption.

The rest of this paper is organized as follows. In Section 2, we review previous work in fixed-priority preemptive multiprocessor scheduling and define concepts and system models used. Section 3 shows the average-case behavior of the partitioned and non-partitioned method, and in Section 4, we discuss and show the architectural impact on the average-case performance. We discuss other aspects of the scheduling problem in Section 5, and summarize our results in Section 6.

## 2 Background

Recall that there are two methods that are used to solve the multiprocessor scheduling problem, namely the partitioned and the non-partitioned method[1]. In this section, we first discuss in more detail the capabilities and problems of these methods, and then define concepts and system models used in this paper.

### 2.1 Previous work

For fixed-priority preemptive scheduling of periodically-arriving tasks on a multiprocessor system, both the partitioned and the non-partitioned method have been addressed in previous research. Important properties were presented in a seminal paper by Leung and Whitehead [5]. In particular, they showed that the problem of deciding whether 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 no method dominates the other in the sense that there are task sets which are schedulable with an optimal priority assignment with the non-partitioned method, but are unschedulable with an optimal partitioning algorithm and conversely.

Among the two methods, the partitioned method has received the most attention in the research literature. The main reason for this is 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 partitions, each having its own dedicated processor. Since an optimal solution to the problem of partitioning the tasks is believed to be computationally intractable, many heuristics for partitioning have been proposed (see for example [3, 6, 7, 8, 9, 10]). 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 following limitations. First, no efficient schedulability tests currently exist for the non-partitioned method. The only known necessary and sufficient schedulability test for the non-partitioned method has an exponential time-complexity [11]. The complexity can be reduced with sufficient schedulability tests to a polynomial [12, 2, 13, 14] or pseudo-polynomial [13] time complexity. However, the test in [14] becomes pessimistic when the number of tasks increases, and the tests in [12, 2, 13] become pessimistic when the number of processors increases. Second, no efficient optimal priority-assignment scheme has been found for the non-partitioned method. The rate-monotonic priority assignment (RM) [15], which is optimal on a uniprocessor, is not optimal for multiprocessors using the non-partitioned method [3, 5]. Even worse, task sets with a very low utilization can be unschedulable with RM [3]. We refer to the latter as *Dhall's effect.*

In a recent comparison of the partitioned and non-partitioned method, it was claimed that, even if one is willing to use a necessary and sufficient schedulability test for the non-partitioned method, the non-partitioned rate-monotonic is inferior to the partitioning algorithms [2, Section 3.5]. We believe that this comparison is not complete for two reasons. First, the previous evaluation did not consider a recently-proposed priority assignment scheme, *adaptiveTkC* [4], in which the highest priority is assigned to the task $\tau_i$ which has the least $T_i - k * C_i$, where $k = \frac{1}{2} \cdot \frac{m-1+\sqrt{5m^2-6m+1}}{m}$ ($m$ is the number of processors). As shown in [4], the value of $k$ is selected to counter two effects that occur at $k = 0$ and $k \rightarrow \infty$. The case

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

where $k = 0$ is equivalent to RM, which is known to suffer from Dhall's effect. The other case, $k \to \infty$, gives highest priority to the task with the longest execution time, which can make task sets unschedulable at arbitrary low utilization. The second reason why the previous evaluation is not complete is that it considered the cost of preemption and migration to be zero. It is not clear how preemption and migration costs will affect performance of the partitioned and non-partitioned methods. Because of these shortcomings, we believe that it is necessary to take a new look at the question of whether to partition or not to partition.

## 2.2 Concepts and System model

We consider the problem of scheduling a task set $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of $n$ independent[2], periodically-arriving real-time tasks on $m$ identical processors. A task arrives periodically with a period of $T_i$. Each time a task arrives, a new *instance* of the task is created. Each instance has 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 *meta period* of the task set is the least common multiple of $T_1, T_2, \ldots, T_n$.

For the partitioned method the system behaves as follows. Each task is assigned to a processor, and then assigned a local (for the processor), unique and fixed priority. With no loss of generality, we assume that the tasks on each processor are numbered in the order of decreasing priority, that is, $\tau_1$ has the highest priority. On each processor, the task with the highest priority of those tasks which has arrived, but not completed, is executed.

For the non-partitioned method the system behaves as follows. Each task is assigned a global, unique and fixed priority. With no loss of generality, we assume that the tasks in $\tau$ are numbered in the order of decreasing priority, that is, $\tau_1$ has the highest priority. Of all tasks that have arrived, but not completed, the $m$ highest-priority tasks are executed[3] in parallel on the $m$ processors.

The *utilization* $u_i$ of a task $\tau_i$ 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, $U = \sum_{i=1}^{n} C_i/T_i$. A task is *schedulable* if all its instances completes no later than their deadlines. A task set is schedulable if all its tasks are schedulable.

To understand the basic performance characteristics, we initially assume the following simple system model.

A1. Tasks are independent, arrive periodically, and can always be preempted. Hence, at every moment, a dispatcher determines which task to execute. In practice, some operating systems use tick driven scheduling, where the ready queue is only inspected at certain times.

A2. Tasks do not require exclusive access to any other resource than a processor.

A3. 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 (that is, the cost of migration is also assumed to be zero).

A4. The cost when a task arrives is zero because we consider cache misses that occur when a task arrives to be included in the execution time. We will use this assumption even if a task executes after arrival on a processor on which it has never executed. These assumptions should be reasonable since worst-case execution time analysis tools typically consider uninterrupted execution of a task that starts in an "empty state" (e.g., all cache lines are empty).

To investigate more realistic characteristics of the system, we will relax A3 later in this paper (in Section 4).

## 3 Average-Case Behavior

In this section, we will conduct an average-case performance evaluation of the non-partitioned and partitioned methods, assuming an idealized architecture with no overhead of task preemption or migration. Our evaluation methodology, which will be described in Section 3.1, is based on simulation experiments using randomly-generated task sets. The rationale for using simulation of synthetic task sets is that it more easily reveals the average-case performance and robustness of a scheduling algorithm than can be achieved by scheduling a single application benchmark. Section 3.2 presents the results from the simulations.

### 3.1 Experimental setup

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 $m = 4$ processors. The number of tasks, $n$, follows a uniform distribution with an expected value of $E[n] = 8$, a minimum of $0.5\,E[n]$, and a maximum of $1.5\,E[n]$. The period, $T_i$, of a task $\tau_i$ is taken from a set $\{100, 200, 300, 400, 500, \ldots, 1600\}$, each number having an equal probability of being selected. The utilization, $u_i$, of a task $\tau_i$ 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 $\tau_i$ is computed from the generated utilization of the task, and the execution time is rounded down to the next lower integer. If the execution time becomes zero, then the task is generated again.

[2]That is, there are no precedence constraints on the arrival time of the tasks.

[3]At each instant, the processor chosen for each of the $m$ tasks is arbitrary. If less than $m$ tasks should be executed simultaneously, some processors will be idle.

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 one meta period[4]. Scheduling decisions are only taken when a task arrives or completes.

We use *success ratio* as the performance measure for average-case performance. The success ratio is the fraction of all generated task sets that are successfully scheduled with respect to an algorithm. The success ratio is computed for each point in a plot as an average of 2,000,000 task sets[5]. In the evaluation of the partitioned method, we consider a task set as successfully scheduled if and only if the number of processors required according to the bin-packing algorithm is no greater than $m$. In the evaluation of the non-partitioned method, we consider a task set as successfully scheduled if the task set is schedulable, that is all task instances in the task set simulated during a meta period completed no later than their deadlines.

Two non-partitioned priority-assignment schemes are evaluated, namely RM [15] and adaptiveTkC [4]. Two bin-packing-based partitioning algorithms are studied, namely RM-FFDU [8], and R-BOUND-MP [9]. The reason for selecting these two latter algorithms is that we have found that they are the partitioning algorithms which provide the best performance in our experimental environment (other algorithms, such as RRM-BF [7] and RMGT [6] offer lower performance). For all partitioning algorithms, we use the corresponding sufficient schedulability test.

We have also evaluated a hybrid partitioned/non-partitioned algorithm, which we will call *RM-FFDU+adaptiveTkC*. The reason for considering a hybrid solution is that we may be able to increase processor utilization with the use of non-partitioned tasks, without jeopardizing the guarantees given to partitioned tasks. The RM-FFDU+adaptiveTkC scheme operates in the following manner. First, as many tasks as possible are partitioned with RM-FFDU on the given number of processors, and are given local priorities. Then, the remaining tasks (if any) are assigned global priorities according to the adaptiveTkC priority-assignment scheme. Each processor has a local ready queue for the partitioned tasks and there is a global ready queue for the non-partitioned tasks. A processor executes a task from its local ready queue, if the local ready queue of that processor is non-empty. A processor executes a task from the global ready queue, if the local ready queue of that processor is empty.

[4]The reason for scheduling during a meta period is because for the non-partitioned method, the response time of a task is not necessarily maximized when a task arrives at the same time as its higher priority tasks [4]. We therefor select small values of $E[n]$ to avoid that the meta period grows too large, causing simulations to take too long time. We also select small values of $m$ since $m$ must be less than $n$ to make the scheduling problem non-trivial.

[5]With 95% confidence, we obtain an error of the success ratio that is less than 0.1%.

## 3.2 Performance comparison

Figure 1 shows the results of the simulation experiment. We make three observations.

First, RM offers the worst performance. However, it is not as bad as suggested by previous studies [3]. The reason for this is of course that Dhall's effect, although it exists, does not occur frequently. This observation also corroborates a recent study [2].

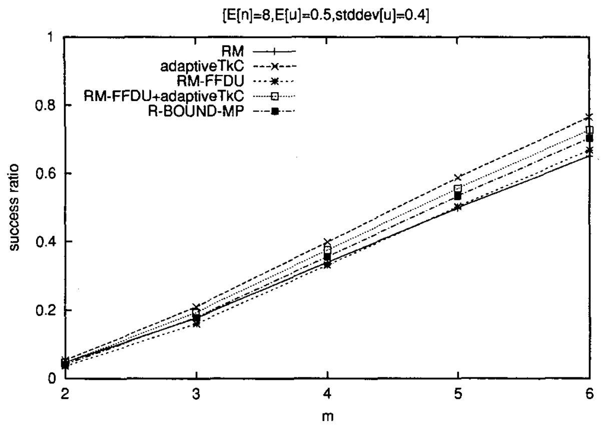
Second, adaptiveTkC offers the best performance, no matter how we vary the parameters (the number of processors, the expected value of the number of tasks, the expected value of task utilization and the standard deviation of the task utilization). The reason for this is that adaptiveTkC takes advantage of the salient property of the non-partitioned method, namely the ability to schedule tasks in slots of unused time on different processors (as RM does).

Third, the hybrid partitioned/non-partitioned algorithm consistently outperforms the corresponding partitioning algorithms. This indicates that such a hybrid scheme is a viable alternative to use in multiprocessor systems that mixes real-time tasks of different criticality. The reason for the good performance is that the hybrid partitioned/non-partitioned algorithm can schedule all task sets that the corresponding partitioned method can schedule, but the hybrid partitioned/non-partitioned algorithm can also schedule some tasks in slots of unused time on different processors.

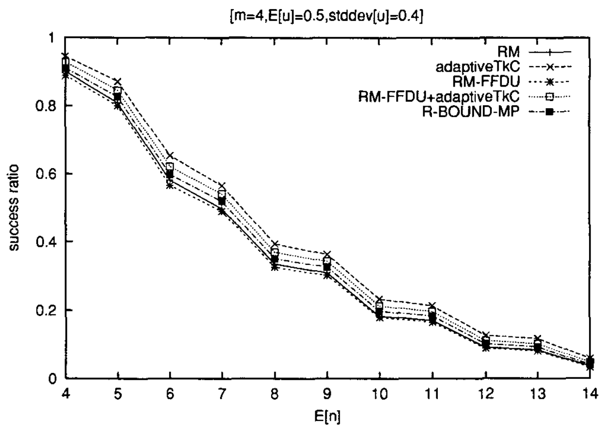
For all partitioning algorithms, sufficient schedulability tests were originally proposed to be used. Now, if we instead use a necessary and sufficient schedulability test based on response-time analysis [16] for the partitioning algorithms, the success ratio can be expected to increase, albeit at the 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. Figure 2 shows the simulation results when the partitioning algorithms use response-time analysis. Here, we make the following observation. RM-FFDU with response-time analysis only provides a slightly higher success ratio than adaptiveTkC while other partitioning algorithms still have a lower success ratio than adaptiveTkC. Note that this means that, even if the best partitioning algorithm (R-BOUND-MP) use response-time analysis, it still performs worse than adaptiveTkC. This should not come as a surprise, since for R-BOUND-MP, the performance bottleneck is the partitioning and not the schedulability test [9].

## 4 Architectural Impact

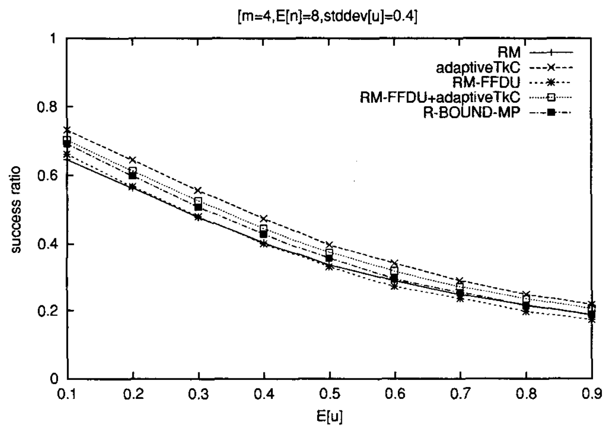
From the results in Section 3, we observed that adaptiveTkC performed well under the assumption of an ideal-



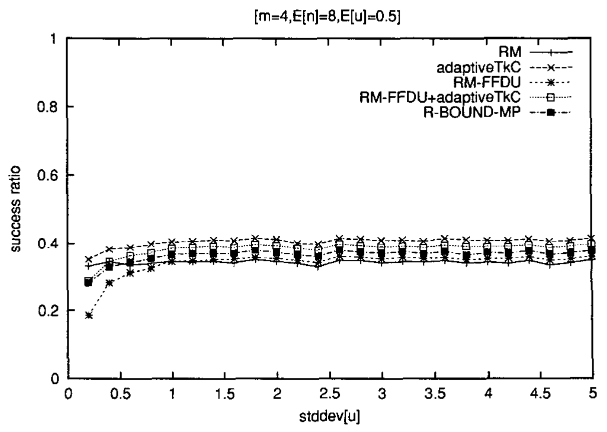
(a) Success ratio as a function of the number of processors.



(b) Success ratio as a function of the number of tasks.



(c) Success ratio as a function of the expected value of utilization of tasks.



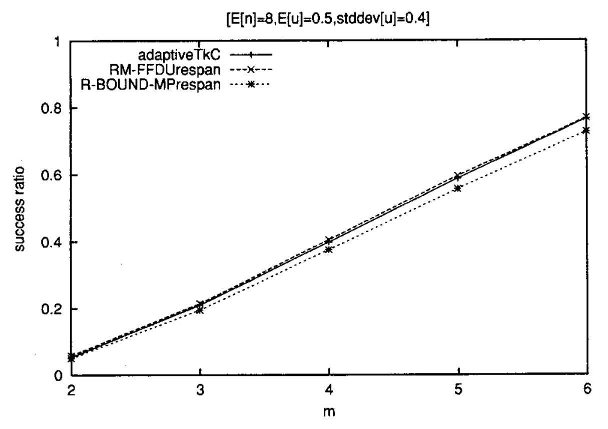
(d) Success ratio as a function of the standard deviation of utilization of tasks.

Figure 1: Success ratio for different scheduling algorithms when the partitioning algorithms use a sufficient schedulability test (polynomial time complexity).

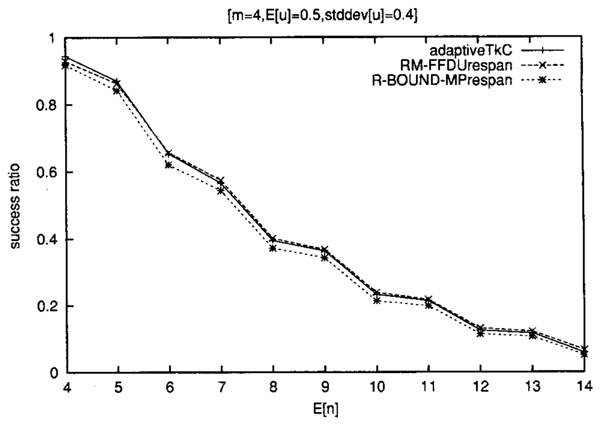
ized architecture. Now, in order to evaluate the performance of a realistic system we must incorporate some architectural costs. In this section, we will study the impact of two architectural costs, namely *preemption* and *migration*.

It is tempting to believe that the non-partitioned method causes a larger amount of (and more costly) preemptions[6] than the partitioned method, and that this makes more task sets schedulable with the partitioned method than with the non-partitioned method. We will now show that such statements cannot easily be made. First, we will propose a dispatcher for the non-partitioned method that reduces the number of preemptions by analyzing the current state of the schedule. We will then show that the average number of preemptions generated by the best non-partitioned method using the new dispatcher is significantly less than that of the best partitioning algorithm. Finally, we will evaluate the schedulability of the non-partitioned and partitioned method when the costs of preemption and migration are accounted for.

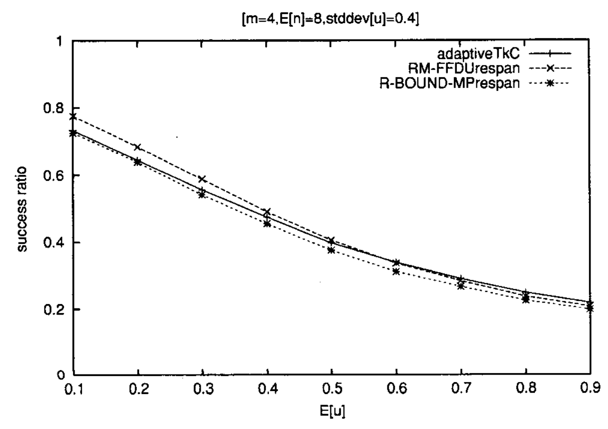
[6]A task is preempted if it has remaining execution time but does not immediately continue to execute on the same processor.



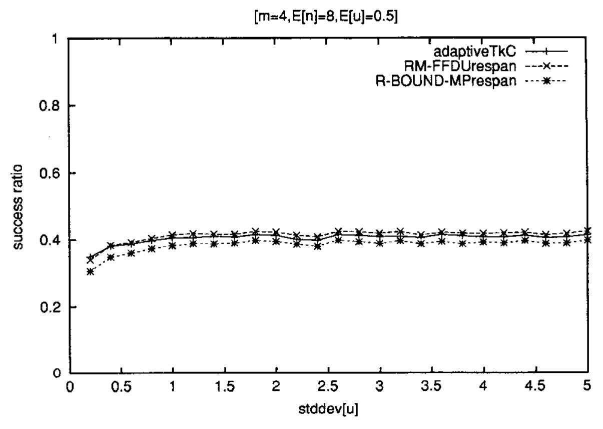
(a) Success ratio as a function of the number of processors.



(b) Success ratio as a function of the number of tasks.



(c) Success ratio as a function of the expected value of utilization of tasks.



(d) Success ratio as a function of the standard deviation of utilization of tasks.

Figure 2: Success ratio for different scheduling algorithms when the partitioning algorithms use a necessary and sufficient schedulability test (pseudo-polynomial time complexity).

## 4.1 Improved Dispatcher

Recall that the non-partitioned method using fixed-priority preemptive scheduling only requires 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. As long as the cost of a preemption is zero, the task-to-processor assignment at each instant does not affect schedulability. However, on real computers the time required for a preemption is non-negligible. If the task-to-processor assignment is selected arbitrarily, it could happen 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 same ones that executed last. With the original dispatcher, this would have serious consequences for the schedulability of the system. Hence, to reduce the risk of unnecessary preemptions, 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 preemptions is minimized.

We now propose a heuristic for the task-to-processor assignment that will reduce the number of preemptions for the non-partitioned method. The basic idea of the task-to-processor assignment algorithm is to determine which of the

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

**Input**: Let $\tau_{before}$ 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 selected for execution.

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

```
1:  $E = \{p_i : (\tau_{ij,before} \neq \text{non}-\text{existing}) \wedge (\tau_{ij,before} \in \tau_{highest})\}$
2:  for each $p_i \in E$
3:      remove $\tau_{ij,before}$ from $\tau_{highest}$
4:      $\tau_{ij,after} \leftarrow \tau_{ij,before}$
5:  for each $p_i \notin E$
6:    $if\ \tau_{highest} \neq \emptyset$
7:      select an arbitrary $\tau_j$ from $\tau_{highest}$
8:      remove $\tau_j$ from $\tau_{highest}$
9:      $\tau_{ij,after} \leftarrow \tau_j$
10:   $else$
11:     $\tau_{ij,after} \leftarrow \text{non}-\text{existing}$
```

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 in their previous execution. The algorithm for this is described in Algorithm 1. In the remainder of this paper, we will refer to this dispatcher as the *preemption-aware* dispatcher. We let *adaptiveTkCunaware* denote adaptiveTkC together with a dispatcher where the task with the highest priority is assigned to the processor with the least index, unaware of the preemptions it will generate. Correspondingly, we let *adaptiveTkCaware* denote adaptiveTkC together with the preemption-aware dispatcher. In the remainder of this section, we will assume that, whenever the non-partitioned method is used, adaptiveTkCaware is used. AdaptiveTkCunaware will only be used as a baseline algorithm.

## 4.2 Number of Preemptions

To demonstrate that the preemption-aware dispatcher is effective, we will start by showing its performance in terms of the number of preemptions generated. To this end, it would be natural to simulate scheduling and count the total number of preemptions generated during a meta period and use that number as a measure of the number of preemptions. However, if many task sets are simulated, and we take the sum of the preemptions in all task sets, then task sets with a large meta period would be biased in that they would have a higher impact on the total number of preemptions. To make each task set equally important, we have therefore chosen to use *preemption density* as the measure of the number of preemptions. Preemption density of a scheduled task set is defined as the number of preemptions generated during a meta period divided by the length of the meta period.

To reveal which of the two methods (partitioned or non-partitioned) that generates the highest number of preemptions, we have simulated scheduling of randomly-generated task sets and computed the preemption density for each of these task sets. Since, in this subsection, we are only interested in the number of preemptions — not their impact on schedulability — we assume that the cost of preemption and the cost of migration is zero. We simulate scheduling using adaptiveTkCaware, adaptiveTkCunaware and R-BOUND-MP because they are the best (as demonstrated in Section 3.2) schemes for the non-partitioned method and the partitioned method, respectively. We use the same experimental setup as described in Section 3.1. We varied the number of tasks and simulated 5,000 task sets for each point in a plot. We then computed the preemption density only for those task sets for which both adaptiveTkCaware, adaptiveTkCunaware and R-BOUND-MP were schedulable.

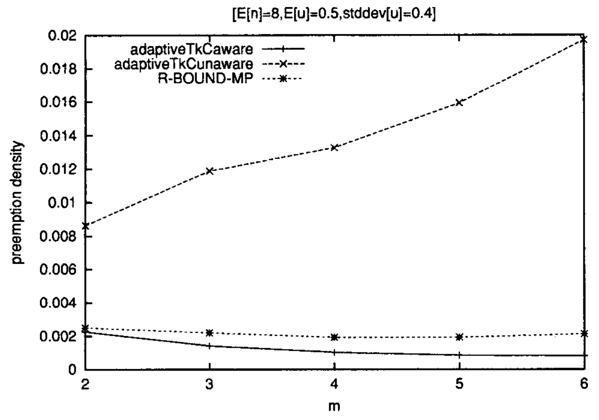
The results from the simulations are shown in Figure 3. We observe that, on average, the preemption density of the best non-partitioned method (adaptiveTkCaware) is lower than the preemption density of the best partitioned method (R-BOUND-MP). The reason for this is that, for the partitioned method, an arriving higher-priority task must always preempt an executing lower-priority task on the same processor. With the non-partitioned method, a higher priority task can sometimes execute on another idle processor, thereby avoiding a preemption. Figure 4 illustrates a situation where the non-partitioned method with the preemption-aware dispatcher causes the number of preemptions to be less than the number of preemptions of a partitioned method.

Note that, in Figure 3(a), increasing the number of processors gives a different effect on the preemption density for adaptiveTkCaware than for adaptiveTkCunaware. It can be explained as follows. When the number of processors increases, there will be more tasks executing in parallel and it is more likely that, during a time interval, a task completes its execution. Consider which processor a low-priority task executes on when a higher-priority task completes it execution. For adaptiveTkCaware, the lower-priority task will continue to execute on the same processor as it did before the higher priority task completed. Hence no preemption occurs. However, for adaptiveTkCunaware (as defined in Section 4.1), the lower priority task will continue its execution on another processor with a lower index. That is, with our definition, a preemption.

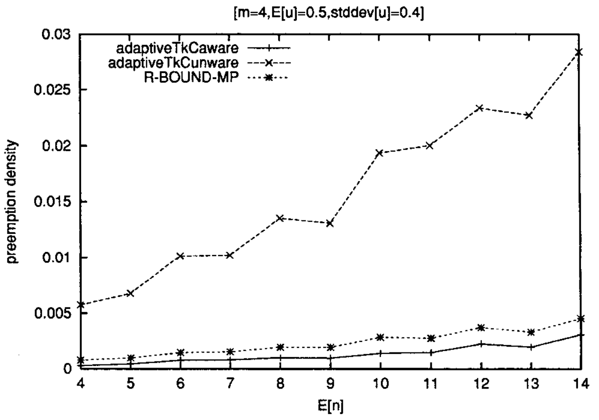
## 4.3 Average-Case Behavior

With the new dispatcher at hand, we are now in position to assess the schedulability of a system with non-negligible preemption and migration cost. We will model the preemption cost and migration cost of a uniform-memory access shared-memory multiprocessor with caches.

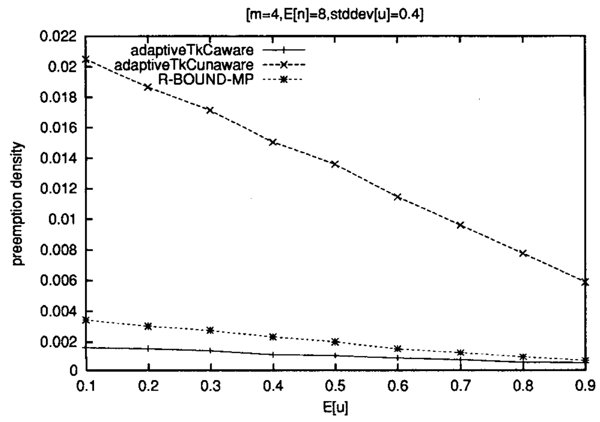
For the assumed system, the cost of a preemption includes the following terms: time to acquire the ready queue, time to save and restore a task's environment (e.g., registers,



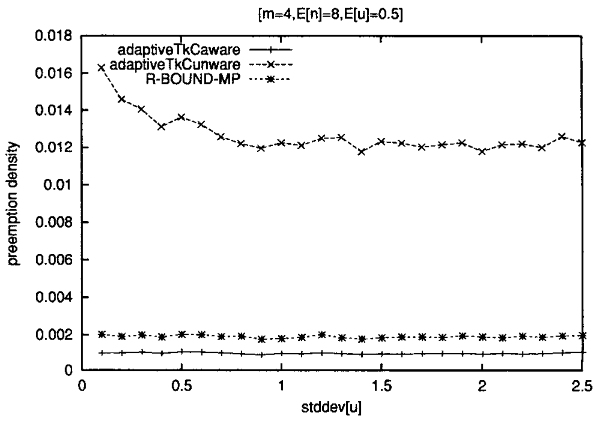
(a) Preemption density as a function of the number of processors.



(b) Preemption density as a function of the number of tasks.

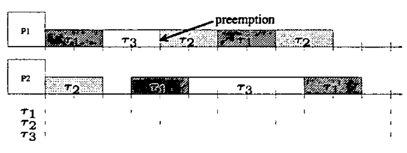


(c) Preemption density as a function of the expected value of utilization of tasks.

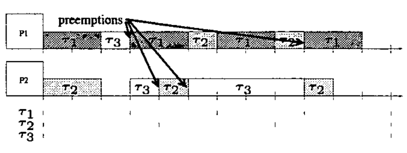


(d) Preemption density as a function of the standard deviation of utilization of tasks.

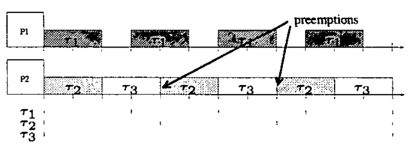
Figure 3: Preemption density of different scheduling algorithms.



(a) AdaptiveTkCaware



(b) AdaptiveTkCunaware



(c) R-BOUND-MP

Figure 4: Preemptions for the non-partitioned method and partitioned method when scheduling the task set $\{(T_1 = 3, C_1 = 2), (T_2 = 4, C_2 = 2), (T_3 = 12, C_3 = 6)\}$ on $m = 2$ processors. The non-partitioned method, with adaptiveTkCaware, generates the least number of preemptions.

memory mapping) and the cost of cache misses due to cache reloading when the task resumes. In the evaluation in this section, we will assume that the time to acquire the ready queue and time to save and restore a task's state is zero. We do this for three reasons, namely (i) these costs depend heavily on the implementation (hardware/software) and mechanism to protect the ready queue (fine-grained/course-grained locking/wait-free/lock-free), (ii) it has been found that the time to execute kernel code constitute only a fraction of the total cost of a context switch [17], and (iii) the trend of faster processors and (relatively) slower memories will make the cache reload effect an even more dominant term in the cost of preemption.

Our simulated system model is now changed as follows. When a task is resumed, we add a cost of $preempt.ratio \cdot E[u] \cdot E[T]$ to its remaining execution time because of cache reloading. If a task is resumed on another processor than it was preempted on we add yet another cost of $mig.ratio \cdot E[u] \cdot E[T]$ to its remaining execution time because of cache reloading due to the migration. We use the same experimental setup as described in Section 3.1, but simulated 5,000 task sets for each possible combination of preemption ratio, migration ratio, and the number of processors.

The results from our simulations are shown in Figure 5. We observe that, when the migration and preemption costs are high, adaptiveTkCaware has higher success ratio than adaptiveTkCunaware. This indicates that the preemption-aware dispatcher is effective. We can also see that the partitioned method becomes more favorable when the migration ratio is large, because no migrations can occur for that method. On the other hand, the non-partitioned method becomes more favorable when the preemption ratio increases, because the preemption-aware dispatcher can reduce the number of preemptions. However, when there are only two processors available (see Figure 5(a)), the preemption-aware dispatcher becomes less effective, because it has only one alternative processor to try to schedule a task on to reduce the number of preemptions.

## 5 Discussion

From the previous sections, we have learned that the relative performance of the partitioned and non-partitioned method varies depending on the system models and performance measures used. However, the decision whether to partition or not to partition may also need to consider other aspects than comparing success ratio. Below we list some interesting aspects.

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 [2]. 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 execution 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 hence offers one solution for exploiting processors efficiently, 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 alternative approach would be to partition only the tasks that have hard real-time constraints, and then let the tasks with soft constraints be scheduled by the non-partitioned method.

The partitioned method has the advantage of having a low computational complexity of the schedulability test, something that is important for, e.g., online admission tests and QoS negotiation [18]. There is also a complexity associated with the algorithm that reschedules a task set that has changed. The non-partitioned method has lower computational complexity for rescheduling than the partitioned method if the partitioned method use the response-time analysis. However, even if sufficient schedulability tests are used for the partitioned method, the non-partitioned method has a slightly lower computational complexity of rescheduling ($O(n \log n)$ for adaptiveTkC) than the partitioned method ($O(nm + n \log n)$ for R-BOUND-MP and $O(nm+n \log n)$ for RM-FFDU). For systems that reschedule frequently, but do not rely on schedulability tests, e.g., feedback control scheduling [19], the non-partitioned method, with its slightly higher success ratio and its slightly lower computational complexity of rescheduling, becomes a viable alternative. Also, since these systems are likely to be used when the execution time varies, the advantage of reclaiming resources in the non-partitioned method gives a further performance increase.

## 6 Conclusions

It has been believed that the non-partitioned method is inferior to the partitioned method. In this paper, we have shown through two evaluations that by using a better priority assignment scheme and an improved dispatcher, such a belief is not necessarily true. First, an average-case comparison using an idealized architecture, showed that, if a system has a small number of processors, then the non-partitioned method offers higher performance than the partitioned method. Second, an average-case comparison using a realistic architecture, showed that, for several combinations of preemption and migration costs, the non-partitioned method offers higher performance.

## Acknowledgement

This work is funded by the national Swedish Real-Time Systems research initiative ARTES (www.artes.uu.se), supported by the Swedish Foundation for Strategic Research.

|  | preempt. ratio |  |  |  |  |
|---|---|---|---|---|---|
| mig. ratio | 1 % | 5 % | 10 % | 20 % | 50 % |
| 1 % | 0.05<br>0.05<br>0.04 | 0.04<br>0.03<br>0.04 | 0.03<br>0.02<br>0.03 | 0.02<br>0.01<br>0.02 | 0.01<br>0.00<br>0.01 |
| 5 % | 0.05<br>0.03<br>0.04 | 0.04<br>0.02<br>0.04 | 0.03<br>0.01<br>0.03 | 0.02<br>0.00<br>0.02 | 0.01<br>0.00<br>0.01 |
| 10 % | 0.05<br>0.02<br>0.04 | 0.04<br>0.01<br>0.04 | 0.03<br>0.01<br>0.03 | 0.02<br>0.00<br>0.02 | 0.01<br>0.00<br>0.01 |
| 20 % | 0.04<br>0.01<br>0.04 | 0.03<br>0.01<br>0.04 | 0.02<br>0.00<br>0.03 | 0.02<br>0.00<br>0.02 | 0.01<br>0.00<br>0.01 |
| 50 % | 0.02<br>0.00<br>0.04 | 0.02<br>0.00<br>0.04 | 0.01<br>0.00<br>0.03 | 0.01<br>0.00<br>0.02 | 0.01<br>0.00<br>0.01 |

(a) $m = 2$

|  | preempt. ratio |  |  |  |  |
|---|---|---|---|---|---|
| mig. ratio | 1 % | 5 % | 10 % | 20 % | 50 % |
| 1 % | 0.21<br>0.19<br>0.18 | 0.19<br>0.11<br>0.16 | 0.16<br>0.07<br>0.13 | 0.13<br>0.02<br>0.09 | 0.08<br>0.00<br>0.05 |
| 5 % | 0.20<br>0.12<br>0.18 | 0.18<br>0.08<br>0.16 | 0.15<br>0.04<br>0.13 | 0.12<br>0.01<br>0.09 | 0.08<br>0.00<br>0.05 |
| 10 % | 0.18<br>0.07<br>0.18 | 0.16<br>0.05<br>0.16 | 0.14<br>0.02<br>0.13 | 0.12<br>0.01<br>0.09 | 0.07<br>0.00<br>0.05 |
| 20 % | 0.15<br>0.02<br>0.18 | 0.14<br>0.01<br>0.16 | 0.13<br>0.01<br>0.13 | 0.10<br>0.00<br>0.09 | 0.07<br>0.00<br>0.05 |
| 50 % | 0.11<br>0.00<br>0.18 | 0.10<br>0.00<br>0.16 | 0.09<br>0.00<br>0.13 | 0.08<br>0.00<br>0.09 | 0.05<br>0.00<br>0.05 |

(b) $m = 3$

|  | preempt. ratio |  |  |  |  |
|---|---|---|---|---|---|
| mig. ratio | 1 % | 5 % | 10 % | 20 % | 50 % |
| 1 % | 0.40<br>0.36<br>0.35 | 0.37<br>0.23<br>0.32 | 0.33<br>0.17<br>0.27 | 0.29<br>0.12<br>0.22 | 0.21<br>0.12<br>0.16 |
| 5 % | 0.38<br>0.24<br>0.35 | 0.35<br>0.18<br>0.32 | 0.32<br>0.14<br>0.27 | 0.28<br>0.12<br>0.22 | 0.21<br>0.12<br>0.16 |
| 10 % | 0.36<br>0.17<br>0.35 | 0.33<br>0.14<br>0.32 | 0.31<br>0.13<br>0.27 | 0.26<br>0.12<br>0.22 | 0.20<br>0.12<br>0.16 |
| 20 % | 0.31<br>0.13<br>0.35 | 0.30<br>0.12<br>0.32 | 0.27<br>0.12<br>0.27 | 0.24<br>0.12<br>0.22 | 0.20<br>0.12<br>0.16 |
| 50 % | 0.24<br>0.12<br>0.35 | 0.23<br>0.12<br>0.32 | 0.22<br>0.12<br>0.27 | 0.21<br>0.12<br>0.22 | 0.18<br>0.12<br>0.16 |

(c) $m = 4$

|  | preempt. ratio |  |  |  |  |
|---|---|---|---|---|---|
| mig. ratio | 1 % | 5 % | 10 % | 20 % | 50 % |
| 1 % | 0.76<br>0.67<br>0.69 | 0.72<br>0.45<br>0.62 | 0.67<br>0.36<br>0.54 | 0.60<br>0.34<br>0.46 | 0.49<br>0.34<br>0.38 |
| 5 % | 0.73<br>0.45<br>0.69 | 0.69<br>0.37<br>0.62 | 0.65<br>0.35<br>0.54 | 0.58<br>0.34<br>0.46 | 0.49<br>0.34<br>0.38 |
| 10 % | 0.69<br>0.36<br>0.69 | 0.66<br>0.35<br>0.62 | 0.62<br>0.34<br>0.54 | 0.56<br>0.34<br>0.46 | 0.48<br>0.34<br>0.38 |
| 20 % | 0.63<br>0.34<br>0.69 | 0.60<br>0.34<br>0.62 | 0.57<br>0.34<br>0.54 | 0.53<br>0.34<br>0.46 | 0.47<br>0.34<br>0.38 |
| 50 % | 0.52<br>0.34<br>0.69 | 0.50<br>0.34<br>0.62 | 0.49<br>0.34<br>0.54 | 0.47<br>0.34<br>0.46 | 0.44<br>0.34<br>0.38 |

(d) $m = 6$

Figure 5: Success ratio for scheduling algorithms as a function of the number of processors, preemption costs and migration costs. Each square shows adaptiveTkCaware (upper), adaptiveTkCunaware (middle), R-BOUND-MP (lower). The shaded regions indicate where the non-partitioned method (with adaptiveTkCaware) performs better than the partitioned method (R-BOUND-MP).

## 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. 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.

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

[4] B. Andersson and J. Jonsson. Some insights on fixed-priority preemptive non-partitioned multiprocessor scheduling. In *Proc. of the IEEE Real-Time Systems Symposium – Work-in-Progress Session*, Orlando, Florida, November 27–30, 2000. Also in TR-00-10, Dept. of Computer Engineering, Chalmers University of Technology.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[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] 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.

[16] M. Joseph and P. Pandya. Finding response times in a real-time system. *Computer Journal*, 29(5):390–395, October 1986.

[17] V. Padmanabhan and D. Roselli. The real cost of context switching. Technical report, CS Division, University of California at Berkeley, November 1994.

[18] T. F. Abdelzaher, E. M. Atkins, and K. G. Shin. QoS negotiation in real-time systems and its application to automated flight control. In *Proc. of the IEEE Real-Time Technology and Applications Symposium*, pages 228–238, Montreal, Canada, June 9–11, 1997.

[19] 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, 1999.