Predictable Virtual Processors for Real-Time Systems

Evangelos P. Markatos        Thomas J. LeBlanc

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 396
August 1991

Abstract

Real-time systems often consist of tasks that have different computation characteristics and timing constraints. Tasks from different task models may have to share a processor, especially in uniprocessor and small-scale multiprocessor systems. Traditional methods for sharing a processor among different task models are limited to models with similar, overlapping characteristics, or result in intractable scheduling algorithms. In this paper we present a simple, general solution to this problem based on predictable virtual processors, an abstraction of a physical processor that can predictably schedule real-time tasks. A single task model can be scheduled on a predictable virtual processor as if it were running on a dedicated processor. Virtual processors are scheduled on the physical processor frequently enough and for sufficient duration to create the illusion of a dedicated processor. Predictable virtual processors, coupled with a two-level scheduling approach, separate the application problem of implementing a single scheduler for a given task model from the system problem of sharing a physical processor among several task models, thereby simplifying the design and development of real-time systems.

This material is based upon work supported by the National Science Foundation under Grant number CDA-8822724. The Government has certain rights in this material.
1 Introduction

A real-time system may consist of many different tasks with a variety of computation characteristics and timing constraints. Periodic tasks [14] are used for low-level control functions. Sporadic tasks with hard deadlines [8] are used for fast response to unexpected events. Imprecise computations [13], which are composed of tasks that follow a converging iterative pattern of computation, are used to trade time for accuracy, and are especially useful in real-time planning. Other task models represent the temporal constraints of the system using various forms of deadlines. Hard deadlines are used to specify actions that must be done by a certain time; earliest starting times are used to specify actions that have to start after a specific time. When deadlines are not strict, soft deadlines and value functions [9] can be used to describe the fact that tasks may be useful even if they complete after the specified deadline. Value functions and criticalities are also used to describe the importance of a task to the success of the real-time system.

Future real-time systems, especially those used in robotics applications, need the flexibility and abstraction provided by these various task models. For example, low-level leg, hand, and eye control functions might be based on periodic tasks with hard deadlines, ensuring that the hardware works correctly and in accordance with higher-level decision processes. Changes in the environment might trigger changes in behavior by increasing criticality of a task set or performing an aperiodic task in response to an external event. High-level planning decisions might be performed by tasks that compute the best possible plan within the time allocated. As can be seen from these examples, a complex robotics application is likely to incorporate a wide variety of task models, with differing task attributes and timing constraints.

Scheduling tasks from different task models on the same processor is complicated by the lack of common ground across all task models. For example, some task models have known worst-case completion times and deadlines, while others may only have criticalities. It is difficult to find a general scheduler that takes a large number of computational properties and timing constraints into account and schedules tasks in some predictable and optimal fashion. Moreover, even when such a scheduling algorithm is found, it is usually intractable, and approximate solutions must be found with heuristics.

In this paper, we advocate the use of multiple task models and schedulers for those cases where a general scheduling algorithm cannot be found for a set of task models, or where all known schedulers are intractable. In our approach [10], the system designer defines a new task model for each set of task attributes and timing constraints, and defines a scheduler unique to each task model. Each task model is allocated a predictable virtual processor, an abstraction of a physical processor that can predictably schedule real-time tasks. Operating system mechanisms such as real-time clocks, timers, and software interrupts make it possible to implement task model schedulers for a virtual processor in user space. The operating system kernel is responsible for multiplexing several virtual processors on a single physical processor.

This two-level scheduling scheme separates the application problem of implementing a single scheduler for a given task model from the system problem of sharing a physical processor among several task models. Each task model is allocated a virtual processor; real-time tasks are scheduled to run on the virtual processor by a model-specific scheduler (e.g., rate-monotonic scheduler[14], earliest-deadline-first scheduler, priority scheduler, or known techniques for combining task models under a single scheduler [3; 19; 22; 11]). The operating system multiplexes virtual processors on top of the physical processor so as to create the illusion of a dedicated physical processor for each task model. The illusion is preserved by frequently scheduling virtual processors on the physical processor.

This two-level scheduling approach has many attractive properties. It is a general solution to scheduling task models in that, if a feasible schedule exists, our approach will provide a conservative, tractable scheduler, when no alternative scheduler exists, or when the known alternatives are
intractable. It is extensible in that new scheduling algorithms for task sets can be implemented in user space, without affecting previously existing schedulers. It is conceptually appealing in that the scheduler for a task set can be implemented and debugged independently of other schedulers. It is also efficient: any scheduling algorithm known to be optimal for a task set can be used by a virtual processor dedicated to that task set.

Two important questions must be answered before we can employ virtual processors in real-time systems:

1. Is it possible to do predictable real-time scheduling on virtual processors?

2. What information is needed by the operating system scheduler about a task model and how does the operating system use this information to multiplex virtual processors on a single physical processor?

In this paper we address both these questions. First, in section 2, we describe earlier attempts to schedule multiple task models using a single general task model. In section 3 we define the attributes of a task model that must be known in order to schedule the model on a virtual processor, and discuss how those attributes are computed. Section 4 describes how to make virtual processors predictable, and how to use predictable virtual processors to schedule many different task models on the same physical processor. In section 5 we use two examples to show how to use virtual processors to schedule more than one type of task. We discuss the advantages and disadvantages of our approach in section 6, and summarize our results in section 7.

2 Previous Work

Previous work has attempted to solve the problem of running tasks from different task models on the same processor by defining a general task model that can describe all the properties and constraints of all the models under consideration. In this way a scheduler for the general model can schedule all the tasks in the system. This approach is successful only when the different task models have similar computational properties and timing constraints.

Audsley et al. [1] schedule periodic and sporadic tasks on the same processor by assuming a minimum time between successive arrivals of sporadic tasks, and thus modeling sporadic tasks as periodic tasks. If tasks do not have a minimum inter-arrival time, but the inter-arrival time is a random variable, then probabilistic methods and probabilistic guarantees may apply [6].

Other researchers integrate periodic and sporadic tasks by having an explicit time schedule for the future, entering new tasks into the schedule as they arrive[3]. The schedule describes which tasks run on the processor at every time instant. Periodic tasks have static guarantees; sporadic tasks are given dynamic guarantees on a FCFS basis, provided that the already guaranteed tasks still meet their deadlines. This approach sometimes needs to keep extensive bookkeeping information, which may be exponential in the number of tasks that the system considers [3].

Sprunt et al. [22] and Lehoczky et al. [11] describe a way to schedule periodic and non-real-time aperiodic tasks using servers. Servers are high priority periodic tasks, that provide good response time to aperiodic tasks using unused processors cycles of periodic tasks.

Sha et al. [21] incorporate criticalities of periodic tasks within a rate-monotonic scheduler by artificially shortening the periods of tasks depending on the criticality of the task. In doing so, they force the rate-monotonic scheduler to consider both the criticality of a task and its period.

Biyabani et al. [2] describe a way to include criticalities in a deadline-based system. Their approach consists of maintaining an explicit set of guaranteed tasks that can run and meet their
timing constraints. If a new task arrives which cannot be accommodated in the set of guaranteed tasks, less critical tasks may be sacrificed (removed from the set) and their time allocated to the new task so it will meet its timing constraints.

All of these techniques produce a scheduler for two task models with similar computation properties and timing constraints. Unfortunately, these techniques cannot be easily generalized because there is no known algorithm for deriving a new scheduler (optimal or not) for a set of computation properties and timing constraints based on an optimal scheduler for some subset of those properties and constraints. Even in those cases where a scheduler can be found, the resulting scheduling algorithm is often intractable.

Given these difficulties, we propose a two-level scheduler that uses predictable virtual processors to separate essential real-time scheduling (done at the user-level as part of an application or in library software), from routine but predictable multiplexing (done inside the operating system kernel). Although the concept of a virtual processor is well known and widely used in uniprocessor and multiprocessor systems [7; 18], it has never been implemented as a predictable abstraction in conventional operating systems for the following reasons:

- Predictability limits the flexibility of the operating system, while providing little benefits to most conventional applications.
- Predictability is not important for most users, who are interested primarily in average throughput and have no specific timing constraints.
- The information required to implement predictable virtual processors is not available in conventional systems.

These arguments against predictable virtual processors do not hold in real-time systems however. Flexibility within the operating system is readily sacrificed for predictability, users are primarily interested in meeting timing constraints, and plenty of information about the computation characteristics and timing constraints of the applications is available. In the following sections we describe the information that is required to implement predictable virtual processors, and show how to use virtual processors to schedule many different task models on the same physical processor.

### 3 Definitions

We assume a uniprocessor real-time system with many different task models that do not explicitly interact. Each model is allocated a predictable virtual processor, on which the real-time tasks of the model are scheduled. All virtual processors are scheduled on top of the same physical processor.

**Definition 1** A task (or process) model is a set of computation properties and timing constraints.

Examples of computation properties are computation time, synchronization constraints, and period. Timing constraints consist of properties and constraints on those properties. For example, a deadline that must be met is a timing constraint. A set of value functions that must be maximized is another timing constraint. Task models are combinations of computation properties and timing constraints. For example, sporadic tasks with known worst-case computation times and hard deadlines comprise one task model. Sporadic tasks with value functions define another task model. Periodic tasks with known computation times, periods, and hard deadlines shorter than their periods define another task model [24].

---

1 We will relax this assumption later, and consider both multiprocessor systems, and situations where task models do interact.
In fact, any set of computation properties and timing constraints for which there is a known scheduler can be considered a single task model. Although there is a modularity argument for dividing tasks into task models based solely on computation properties and timing constraints, a more efficient schedule often results when different task models (say, periodic and aperiodic tasks) are combined into a single, general task model. When there is a known scheduling algorithm for the general model, we allow the combined task set to be considered a single task model.

Each predictable virtual processor has two parameters that enable the operating system to do predictable scheduling: utilization and unit of time.

The utilization of each virtual processor specifies how long each virtual processor has to be scheduled on the physical processor in order to receive sufficient cycles. If a virtual processor needs many processor cycles, it has a high utilization.

**Definition 2** The utilization of a predictable virtual processor (with respect to some given physical processor P) is the ratio of the speed of the slowest physical processor that meets the timing constraints of the tasks scheduled on the virtual processor to the speed of the actual physical processor P.

The unit of time specifies how often a virtual processor must run so that its tasks receive timely processor cycles and thereby meet their timing constraints.

**Definition 3** The unit of time of a task model is defined as the smallest time unit the task model uses to express computation requirements and timing constraints.

For example, if all computation times and deadlines of a model are defined in multiples of 10 ms, then the unit of time for that model is 10 ms. In other words, the unit of time is the largest amount of time the task model can afford not to receive any processing power, and still meet its timing constraints. If the task model does not receive processor cycles within the specified unit of time, then the timing constraints may be violated. Typical values for the unit of time are on the order of several milliseconds to several seconds.

**How to find the utilization of a virtual processor:**

**Periodic task models:** Assume that we have two periodic tasks with computation time 1 and 2, and periods 5 and 8. Suppose also that they are scheduled by the earliest-deadline-first (EDF) algorithm. The utilization of both tasks is \(1/5 + 2/8 = 0.45\). The processor utilization of the model is 45%. If the tasks are to be scheduled by a rate-monotonic scheduler (RMS), the proposed utilization of the model would be different, since RMS cannot guarantee tasks will meet their constraints if processor utilization is 100%. RMS can schedule two tasks if their total utilization is less than 82%, so the model should request \(45/0.82 = 55\%\) of the processor's cycles. We can lower the 55% utilization for a specific set of tasks even more by using the improved bounds for rate monotonic scheduling in [19].

More general examples of \(N\) periodic tasks are treated in the same way. If we know the sum of the utilizations of the tasks is \(U\), then the utilization is \(U\) if the EDF scheduler is used, and \(\frac{U}{N(2^N - 1)}\) if RMS is used. Other periodic task models [24; 20] are treated in a similar way.

---

2Note that the unit of time cannot be less than the interrupt service time plus the context switch time because the system cannot turn its attention to some other job in less than that amount of time.

3Processor utilization must include any synchronization overhead and worst-case blocking time.
Non-periodic static task models: When the set of tasks to be scheduled and their timing constraints is known beforehand, then static schedulers are typically employed. These schedulers usually find optimal schedules for tasks. In such cases, the processor utilization of the task model is the minimum percentage of the processor's cycles that the model needs in order to meet its timing constraints. This percentage can be found using the following reasoning:

- If we give 0% of the physical processor's cycles to the model, the timing constraints will not be met (assuming a nontrivial task set).
- If we give 100% of the processor's cycles to the model, the timing constraints will be met (assuming a static schedule exists).
- If we give a specific percentage (say p) of the processor's cycles to the model, we can easily determine if the given percentage is enough for the task model to meet its timing constraints by applying the schedulability test of the model on a processor that is p times slower than the actual physical processor.

Thus, a simple binary search in the interval of percentages [0, 100], performing a schedulability test on each case, will give the processor utilization of the model.

Dynamic task models: Dynamic task models do not have complete information about the tasks in advance. Most models do not know when in the future tasks will appear and what their needs will be. Schedulers for these models try to meet all the timing constraints that can be met. Earliest-deadline-first is an example of such a scheduler. However, if the timing constraints cannot be met, these schedulers try to degrade as gracefully as possible, by missing as few deadlines as possible, or by trying to meet the deadlines of critical tasks. Unfortunately, most schedulers for dynamic models are intractable, and suboptimal schedulers are used. Those suboptimal schedulers run in polynomial time (or linear time, if possible), and miss as few deadlines as possible. In this context, the greater the percentage of a physical processor's cycles a model receives, the more timing constraints will be satisfied, either by dedicating more cycles to the application, or by running more sophisticated schedulers that will meet the timing constraints of more tasks.

There is obviously a tradeoff to be made between the speed of the processor used and the timing constraints that will be met. This tradeoff can be examined using probabilistic analysis [6], or simulation [25; 17; 23]. Probabilistic analysis makes some assumptions about the arrival rate of tasks, and calculates the probability that a specific scheduler will miss task deadlines. Extensive simulations, on the other hand, can also predict the effectiveness of a given scheduler by measuring the percentage of jobs that do not meet their deadlines under a given scheduler. Either analytic evaluation or simulation can bound the probability of failure of a scheduler for a given architecture. This approach, coupled with the binary search method described earlier, can determine the minimum percentage required by each model, so that the probability of failure is below a certain bound.

In the remainder of this paper we assume that all task models specify a unit of time and a required percentage of the processor they will use. This percentage can be computed directly from the model (as in the periodic tasks, or the statically defined tasks), or chosen after probabilistic analysis or simulation.

4 Multiplexing Predictable Virtual Processors

In this section we show how the operating system kernel shares a single physical processor among several predictable virtual processors, while ensuring that each virtual processor executes frequently enough and for sufficient duration to meet all of its timing constraints.
4.1 Two Task Models with the Same Unit of Time

We first consider the problem of scheduling two different task models on the same physical processor, where the task models have the same unit of time $u$. This means that deadlines, computation times, periods, and any other timing information for the two models is described as a multiple of $u$. The following theorem states that if we divide time in multiples of $u$, and if we give each model the computation time proportional to its utilization within each time unit, then the tasks can be scheduled as if they were scheduled on a dedicated processor.

**Theorem 4** Assume that two different task models with the same unit of time $u$ need to be scheduled on the same physical processor. If the models have utilizations $p_1$ and $p_2$ respectively ($p_1 + p_2 < 1$) and the context switch overhead is $c$ then the two models can be predictably scheduled if:

$$p_1 + p_2 + 2 \cdot c/u < 1$$

**Proof:** Each model receives a virtual processor and the virtual processors are scheduled on top of the physical processor periodically. We use a periodic scheduler with period $u$. As soon as a new unit of time starts, the scheduler will schedule the first virtual processor for $p_1 \cdot u$ time, and then it will schedule the second virtual processor for $p_2 \cdot u$ time. Each time the two models are scheduled in the same order. The rest of the interval $u$, the scheduler may stay idle, schedule one of the two virtual processors, or schedule any background work that may exist. The scheduling overhead per unit of time is two context switches ($2 \cdot c$), one to context switch from the first model to the second one, and one to context switch from the second model back to the first. If the total useful time the task models receive plus the context switch overhead is less than the time unit, the above periodic scheduler can be invoked every period. The scheduler does not overrun any period if:

$$p_1 \cdot u + p_2 \cdot u + 2 \cdot c < u$$

or

$$p_1 + p_2 + 2 \cdot c/u < 1$$

If the condition of the theorem holds, then the described scheduler has the following properties:

- Each model receives as many processor cycles as it requests
- There is never a gap larger than $u$ between two successive executions of the same task model

We will now show that there is a one-to-one mapping between the above schedule on the actual physical processor, and the schedule on a hypothetical physical processor where the tasks meet their timing constraints. We will focus on the first model only. Because the utilization of the model is $p_1$, we know that there exists a physical processor (referred to in figure 1 as "$p_1$'s hypothetical physical processor") that is $p_1$ times as fast as the actual physical processor wherein the tasks of the first task model meet their deadlines. Consider the time divided in multiples of $u$ as seen in figure 1. We map each time interval of $p_1$'s hypothetical physical processor to a time interval of the actual physical processor where the first task model is running, as shown in figure 1. We see that after each multiple of $u$, the task model $p_1$ will have received as many cycles on the actual physical processor as it would have received on its hypothetical physical processor. So, if the timing constraints of the model are met on the hypothetical physical processor, they will be met on the actual physical processor. A similar mapping holds for model $p_2$.

The above theorem can be easily generalized for many task models with the same units of time.

**Corollary 5** Assume that $n$ task models with the same unit of time $u$ need to be run on the same physical processor. If model $i$ has utilization $p_i$, and the context switch overhead is $c$, then the models can be predictably scheduled if:

$$\sum_{i=1}^{n} (p_i + c/u) < 1$$

6
Figure 1: Mapping the running time of a slow physical processor to a predictable virtual processor
4.2 Many Task Models with Different Units of Time

Different task models may have different units of time. For example, low-level control systems may have small units of time corresponding to the interrupt arrival rate of fast devices. On the other hand, high-level, long-running, but time-critical activities may have large units of time, because they deal with events that happen over a range of seconds. The unit of time of a task model depends on the rate of change of the physical system that the model controls. As a simple example consider the unit of time of an automatic pilot control system. If the automatic pilot controls a plane, then it needs to run and update its information and control signals every few milliseconds. If the automatic pilot controls a ship, then running it every few seconds may suffice.

Suppose that each task model (virtual processor) \( i \) has a unit of time \( u_i \) and utilization \( p_i \). This means that within \( u_i \) time this model has to receive \( p_i \% \) of the processor's cycles. Moreover no two successive runs of the task model should be more than \( u_i \) time apart. If we model each virtual processor \( i \) as a periodic task with period \( u_i/2 \), worst-case computation time \( p_i \cdot u_i/2 \), and deadline equal to the arrival of the next instance of the periodic task, then each model will receive enough processor cycles, and no two successive runs of the same model will be more than \( u_i \) time apart. The problem of scheduling periodic tasks on a uniprocessor has received a lot of attention and two basic algorithms for its solution have been developed: earliest-deadline-first and rate monotonic scheduling. Liu and Layland [14] proved that a set of \( n \) tasks with computation times \( C_i \) and periods \( P_i \) can meet their deadlines under the EDF scheduler iff their total utilization is less than 1. They also proved that if total utilization is less than \( n \left(2^{\frac{1}{n}} - 1\right) \) then RMS can schedule the tasks to meet their deadlines.

Earliest-deadline-first scheduling

**Theorem 6** Suppose that we have \( n \) virtual processors, where model \( i \) has utilization \( p_i \) and unit of time \( u_i \). All \( n \) virtual processors can be scheduled on top of one physical processor using the earliest-deadline-first scheduler iff:

\[
\sum_{i=1}^{n} \left(\frac{p_i}{u_i} + \frac{c}{u_i/2}\right) < 1
\]

**Proof:** This is a direct result of [14]. We assume that each virtual processor \( i \) is a periodic process with period \( u_i/2 \) and worst-case computation time \( (u_i/2) \cdot p_i \). The sufficient condition given by Liu and Layland states that if the total utilization of the periodic tasks is less than 1, then the earliest-deadline-first scheduler meets the deadlines of all the tasks. The utilization of the periodic tasks we consider is \( U = \sum_{i=1}^{n} \left(\frac{p_i}{u_i} + \frac{c}{u_i/2}\right) \), and so the earliest-deadline-first scheduler meets all the deadlines if \( U < 1 \). A mapping similar to the one used in the proof of theorem 4 completes the proof.

Rate monotonic scheduling

**Theorem 7** Suppose that we have \( n \) virtual processors, where model \( i \) has utilization \( p_i \) and unit of time \( u_i \). All \( n \) virtual processors can be scheduled on top of one physical processor using the rate
monotonic scheduler if:

\[ \sum_{i=0}^{n} \left( p_i + \frac{c}{(u_i/2)} \right) < n \left( 2^{1/n} - 1 \right) \]

Proof: Same as above.

It is possible to improve the above bound for rate monotonic scheduling based on more precise criteria [19].

4.3 Rate Monotonic vs. Earliest-Deadline-First Scheduling

Both rate monotonic and earliest-deadline-first scheduling can be used to multiplex virtual processors on top of a single physical processor. Although the scheduling bound for rate monotonic scheduling is low in the worst case (69%), this bound is pessimistic and can often be exceeded [22]. Rate monotonic scheduling is simple, and can be implemented using static priorities. If the hardware supports enough interrupt priority levels, then scheduling decisions can be implemented directly in hardware.

Earliest-deadline-first scheduling can use up to 100% of the processor's cycles for real-time processing, but it introduces greater overhead. The relative priorities of processes do not stay constant over time, and therefore a dynamic priority scheduler must be implemented in software. However, recent performance experiments demonstrate that earliest-deadline-first scheduling can be efficiently implemented in software using priority queue operations that cost only a few microseconds each. In our experiments on a BBN Butterfly multiprocessor, the context switch time between user tasks in the same address space (including the cost of priority queue manipulation) is about 50 μs [15]. Of course, this cost will increase as the number of tasks in the system increases, but the cost is proportional to the logarithm of the size of the priority queue. Therefore, even if a scheduling decision has to be made every 10 ms, the overhead of priority queue operations will typically be no more than 0.5%.

4.4 Interactions Between Virtual Processors

Another factor to consider when deciding on a kernel scheduling algorithm is the extent to which interactions between virtual processors should be supported by the kernel. Until now we have assumed that tasks in different models do not interact (e.g., via synchronization), and therefore each virtual processor can be scheduled in isolation. This assumption is not a serious constraint, since there are no known general protocols for synchronization among tasks with different computation properties and timing constraints that preserve scheduler guarantees4. Nonetheless, if tasks in different models do interact, either the user-level scheduler or the kernel scheduler must take these interactions into account.

If a small subset of models interact in a known specific way, then a single scheduler for those models can be implemented in user space and assigned to a virtual processor. If multiple virtual processors are involved, the kernel can provide a mechanism that ensures the appropriate virtual processor executes as required by the user-level synchronization protocol. We will describe such a mechanism based on global priorities.

4 Synchronization protocols have been developed for specific task models and schedulers (such as periodic tasks using rate monotonic scheduling), but these protocols do not generalize to other task models or schedulers.
Previous work has considered periodic processes that are scheduled under the rate monotonic scheduler and synchronize using critical sections. Critical sections may cause priority inversion, where a high priority task is forced to wait, possibly indefinitely, for a low priority task that is preempted inside a critical region. The problem of priority inversion can be avoided or minimized using a priority inheritance protocol or the priority ceiling protocol [20; 19; 16].

These synchronization protocols can be used with rate monotonic scheduling in the kernel to control priority inversion between virtual processors. Rather than associating priorities with individual tasks, the kernel associates priorities with virtual processors. These priorities are established by the kernel scheduler, and are independent of any priorities implemented by a user-level scheduler. If a task is in a critical section, the kernel can use a priority inheritance protocol or the priority ceiling protocol to ensure that the corresponding virtual processor will execute whenever necessary to avoid priority inversion. However, the user-level scheduler for that virtual processor must ensure that the task in the critical section is the one that actually runs.

Any modifications to the basic kernel scheduler, such as support for global priorities, must be carefully considered. Incorporating a priority-based protocol within the kernel scheduler complicates the kernel implementation, and compromises our modular approach to scheduling. However, in many systems priority inversion is a particularly common and nasty problem. There are several well-known protocols that work well with rate monotonic scheduling, so functional gains in user programs may be worth minor complications in the kernel.

5 Discussion

The novel aspect of our approach to scheduling several different task models simultaneously is to divide the problem into two separate problems: real-time task scheduling on virtual processors and multiplexing a physical processor among virtual processors. Rather than attempt to develop a new general scheduler for every set of task models, we implement a single general scheduler for virtual processors in the operating system, and allow model-specific real-time task schedulers to be implemented for each virtual processor. The distinctive features of this approach include:

- **Simplicity** - Each task model, represented by a different virtual processor, can be scheduled by well-known and well-understood algorithms designed for that specific model. No general-purpose scheduler needs to be developed for each set of task models, and new task models can be added to the system without complicating the scheduling algorithm for those models already in place.

- **Modularity** - The schedulers for different task models can be tested and debugged in isolation; scheduling errors in one task model do not make tasks in other models miss their deadlines, since each model executes as if it were running on dedicated hardware. As a result, any task model can execute with any other mix of models, provided that each model receives a predictable virtual processor with the appropriate parameters.

- **Tractability** - Given a set of task models and associated schedulers, our method can always schedule them on a single processor, provided that the appropriate conditions are met. Previous approaches to this general problem provide no such guarantee. For each set of task models, a new scheduler has to be found, and it is often the case that the resulting scheduler is intractable. If an intractable scheduler is found, approximate schedulers have to be developed, often with questionable success.

- **Multiprocessing** - Our approach can be adopted in multiprocessor systems by assigning each virtual processor to a physical processor and never migrating it. All virtual processors on the
same physical processor are scheduled according to the same principles used in the uniprocessor case. The initial assignment of virtual processors to physical processors is somewhat problematic, in that any attempt to minimize the number of physical processors used will result in an intractable algorithm; however, simple heuristic algorithms for this assignment problem have been developed [5].

- Policy/mechanism separation - Our approach is based on policy/mechanism separation [12] in operating system design. The operating system provides the virtual processor abstraction, along with the predictable scheduling of virtual processors on the physical processor. Real-time task scheduling is done outside the operating system kernel, within each individual task model implementation. Our general kernel scheduler can be used in any real-time system, as it is independent of any specific computation characteristics or timing constraints. New task models, and their schedulers, can be easily added, since they are implemented in user space and don't interfere with other task models.

There are disadvantages to our approach as well, including:

- Context switch overhead is introduced by frequent scheduling of virtual processors on the physical processor. We do not believe that this overhead will be substantial however, since the typical unit of time in most real-time systems is much larger than the context switch overhead. Worst-case values of the granularity of timing constraints are in the area of tens of milliseconds, while context switch overhead is usually less than a couple hundred microseconds, resulting in less than 1% overhead.

- A conservative schedule is employed by the kernel. To ensure that the worst-case load for a task model can be scheduled, the utilization for the model must be based on the peak demand of all tasks in the model. If the peak demand is not the average demand, more cycles are allocated to a set of task models than is strictly necessary. Such a conservative scheduler either wastes processor cycles or refuses to guarantee a set of task models that could be guaranteed.

Our method is not the only approach that produces a conservative scheduler. For example, Audsley et al. have suggested modelling aperiodic tasks as periodic tasks with a period equal to the minimum interarrival time [1]. Our approach is a generalization of this same idea. In both cases, the scheduler devotes processor cycles to aperiodic tasks that may not arrive every period.

In general it is difficult to quantify the extent to which our conservative scheduler causes problems, primarily due to the lack of alternative schedulers against which a comparison could be made. We don’t believe this problem is a major one however, for two reasons. First, any general scheduler capable of producing a schedule that does not need to allocate resources for worst-case situations is likely to be intractable. Second, virtual processors can be created dynamically to deal with unusual variance in load. For example, a virtual processor with low utilization may be created for a task model during normal operation, and replaced by a virtual processor with high utilization (effectively more processor cycles) during peak operation. The dynamic nature of predictable virtual processors helps the system to assign the cycles to the task models that really need them. This dynamic adjustment does not impose significant overhead, since it involves only a re-evaluation of the simple test stated in theorems 4, 6, and 7.

6 Examples

We will now give two examples of how to apply predictable virtual processors to the problem of scheduling task models with different characteristics.
6.1 Periodic and Sporadic Tasks

Consider a set of periodic tasks with known computation times, and with deadlines equal to the beginning of the next period. Assume that these periodic tasks never miss a deadline when scheduled using the earliest-deadline-first policy. (That is, the set of periodic tasks utilize less than 100% of the processor's cycles.) Now suppose that in addition to these periodic tasks we also need to schedule sporadic tasks that arrive unpredictably and must execute before a deadline. We would like to guarantee that sporadic tasks meet their deadline, or give an early warning if a deadline will be missed. Sporadic tasks are guaranteed on a FCFS basis.

Chetto and Chetto [4] proposed an algorithm very similar to EDF that guarantees all periodic tasks, and as many sporadic task as is possible. If a sporadic task can not be guaranteed, it is notified as soon as possible. Unfortunately this scheduling algorithm is inefficient; in the worst case, where the periods of the periodic tasks are relatively prime, the algorithm requires exponential time.

Using predictable virtual processors, we can assign one virtual processor to the set of periodic tasks, and another virtual processor to the sporadic tasks. The utilization of the first virtual processor is equal to the utilization of the periodic tasks. The second virtual processor is assigned the remaining utilization (minus the necessary context switch overhead). The two virtual processors run on top of the physical processor using the scheduler of theorem 6.

Having separated the periodic and sporadic tasks, we can design simple schedulers for both models. The periodic tasks run in isolation using an earliest-deadline-first scheduler. The sporadic tasks are scheduled using a separate earliest-deadline-first scheduler, and are guaranteed as they arrive using a feasibility test that involves only sporadic tasks. The feasibility test checks to see if there is sufficient unclaimed execution time on the virtual processor assigned to sporadic tasks to meet the deadline of the newly arrived task. If so, then the new task can be guaranteed. Using a balanced tree data structure, the feasibility test can be performed in time logarithmic in the number of active sporadic tasks. As a result, the cost of using virtual processors is a logarithmic test each time a sporadic task arrives, and the relatively small overhead of context switching virtual processors.

In this example, our method reduces a frequent run-time schedulability test from an exponential algorithm to a logarithmic one. We sacrifice a small number of processor cycles for context switching in order to isolate the two task models, which admits a simple schedulability test for each model. The separation of the two models does not result in an excessively conservative schedule (when compared to the exponential algorithm) because we assume a fixed computation time for each periodic task. Since the peak utilization of periodic tasks is equal to the average utilization, there are no excess cycles available for sporadic tasks.

6.2 Periodic and Aperiodic Tasks

Consider a set of periodic tasks scheduled according to the rate monotonic scheduler, and assume the existence of aperiodic non-real-time tasks that need fast response from the system. There are several known solutions to this scheduling problem[22]:

Background Service: When the processor is idle (because RM can not utilize the processor 100%), run aperiodic tasks until a periodic task arrives.

Polling: Create a periodic polling task whose function is to execute any aperiodic tasks in the system. If the polling task is scheduled for execution when there are no aperiodic tasks in the system, the task suspends itself and loses its cycles for that period.

[Most previous work, including [4], makes this same assumption.]
Server: Create a high-priority bandwidth-preserving periodic server task to execute aperiodic tasks.

If the server task is scheduled for execution when there are no aperiodic tasks in the system, the server suspends itself, but it retains the unused cycles until the end of its period. If the server task has high priority, it can often schedule aperiodic tasks as soon as they arrive, and thereby provide good response time for those tasks.

Using predictable virtual processors, we can assign one virtual processor to the set of periodic tasks, and another virtual processor to the aperiodic tasks. Once again, the second virtual processor is assigned any utilization unused by the periodic tasks. Since the virtual processor assigned to the aperiodic tasks is scheduled for execution periodically by the underlying operating system scheduler, the resulting system is analogous to polling, where the frequency of polling is the frequency of virtual processor scheduling.

Although polling performs much better than background service, it is often inferior to servers. The bandwidth preservation property of servers makes it possible to service an aperiodic task as soon as it arrives, whereas a polling task is scheduled for execution independently of aperiodic task arrival. However, polling performs about as well as servers when the polling period is small [11], as is the case with our virtual processors.

These two examples demonstrate that predictable virtual processors can be competitive from a performance perspective with the best known scheduling algorithms. Nonetheless, we do not advocate that a separate predictable virtual processor always be used for every task model. In those cases where there exists an efficient scheduling algorithm for two task models, a single virtual processor can be used for both models. In all other cases however, where an appropriate scheduling algorithm is either unknown or intractable, predictable virtual processors offer a simple, general solution to scheduling multiple task models.

7 Summary

In this paper we introduced the concept of predictable virtual processors for real-time systems. Through frequent and sufficiently long scheduling, we create the illusion of a dedicated processor for each virtual processor. Real-time tasks are scheduled on top of a virtual processor as if it were a physical processor. Predictable virtual processors are scheduled on top of the physical processor using earliest-deadline-first or rate monotonic scheduling.

This two-level scheduling approach is both simple and efficient. A task model can be implemented and debugged in isolation using any appropriate scheduler, and then incorporated into the system without affecting the timing constraints of other tasks in the system. The modular organization of schedulers can result in sub-optimal virtual processor algorithms, but the improvement in the running time of the schedulers may more than compensate for the sub-optimal schedules.

In some cases our method is inferior to other known techniques. Since our operating system scheduler does not mix virtual processor allocations, unused cycles of one virtual processor cannot be used by another. Scheduling algorithms that can allocate processor cycles left unused by one task model to another task model will be able to guarantee some task sets that a strict separation of models would reject. Those algorithms could be used within a single virtual processor to improve the utilization of a group of tasks however, or within the kernel scheduler to improve overall system utilization (although to do so would significantly complicate the system scheduler). In general, separate virtual processors should be used for different task models whenever:

- A single optimal scheduler for both models cannot be found. If we consider any two task models, each with different computational properties and timing constraints, we see many cases where
there is no known optimal scheduling algorithm to guarantee the tasks in both models, even though there may be an optimal algorithm for each individual model. Such combinations of task models can be directly scheduled using separate predictable virtual processors.

- An optimal scheduler for both models is intractable (or inefficient), while the individual schedulers for the two models are tractable (or efficient). Although the scheduler that results from the use of separate virtual processors may be sub-optimal, it is practical.

- Development and debugging time is at a premium. In many cases it is more important to optimize the programmer's time, rather than optimizing the use of processor cycles. In those cases, separate virtual processors admit modular development, synthesizing new systems by multiplexing known and tested scheduling algorithms.

Different task models should be combined within a single virtual processor whenever:

- There exists a known efficient scheduler that can schedule both models. One such example is a set of periodic tasks scheduled by the rate monotonic scheduler and a set of aperiodic tasks that require fast response time. Many server algorithms have been proposed in the literature[22]; one of these server algorithms could be used for a single virtual processor dedicated to both task models.
References


