OSPERT 2008

Proceedings of the Fourth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications

Prague, Czech Republic

July 1, 2008

In conjunction with:
The 20th Euromicro International Conference on Real-Time Systems, Prague, Czech Republic, July 2-4, 2008

Supported by:
The ARTIST2 Network of Excellence on Embedded Systems Design

FOREWORD

Welcome to Prague and the Fourth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT). Although OSPERT is a relatively new workshop, it has quickly become an important venue for exchanging ideas about operating-systems issues related to real-time and embedded systems. This volume contains the nine papers to be presented at the workshop. This year's workshop will also feature an invited talk by Peter Zijlstra of Red Hat, entitled "Linux-rt: Turning a General Purpose OS into a Real-Time OS." Peter's talk will focus on the linux-rt project, its history, current state, and future.

The contributed papers were read and evaluated by the program committee, but were not formally refereed; it is expected that more polished versions of many of these papers will appear in conferences and fully refereed scientific journals. The Program Committee would like to thank all authors who submitted papers for consideration. The efforts of three external reviewers, Dario Faggioli, Su Fang Hsiao, and Po-Liang Wu, who aided in the review process, are also acknowledged.

After the workshop, a final workshop proceedings will be published as a technical report at the University of North Carolina. Authors are encouraged to revise their papers, if needed, to reflect issues and suggestions raised during the workshop. The final proceedings will be made available on-line at the conference website (`http://www.cs.unc.edu/~anderson/meetings/ospert08/OSPERT.html`).

Partial funding for the workshop was provided by the ARTIST2 Network of Excellence on Embedded Systems Design. This support is gratefully acknowledged. The efforts of Alan Burns, Gerhard Fohler, Zdenek Hanzáek, and Giuseppe Lipari in helping to organize this event are also appreciated.

Program Committee

INVITED PRESENTATION

Linux-rt: Turning a General Purpose OS into a Real-Time OS

Peter Zijlstra, *Red Hat*

Talk Abstract

This talk will be about the linux-rt project, its history, current state and future. It will cover the design decisions we made and highlight some of the practical techniques like run-time lock validation and latency tracers that helped us to quickly spot and fix problems.

It will touch upon some of the recent developments in linux-rt such as rwlocks with full priority inheritance support and adaptive spins for mutexes as well as talk about some of the work that is currently in progress.

The talk will also focus on current and future challenges and how academic real-time researchers can help solving those and by doing so bring our two communities closer together.

Speaker Bio

Peter Zijlstra is a professional Linux kernel hacker who has made significant contributions to various core Linux subsystems including the VM and scheduler. He is one of the maintainers of the lockdep/lockstat infrastructure and an active contributor to the linux-rt effort. He is currently employed by Red Hat.

TABLE OF CONTENTS

Real-Time on Linux

Coordinated Management of Multiple Resources or Components

Scheduling-Related Issues

An efficient implementation of the BandWidth Inheritance protocol for handling hard and soft real-time applications in the Linux kernel *

Dario Faggioli, Giuseppe Lipari and Tommaso Cucinotta
e-mail: d.faggioli@sssup.it, g.lipari@sssup.it, t.cucinotta@sssup.it
Scuola Superiore Sant'Anna, Pisa (Italy)

Abstract

This paper presents an improvement of the Bandwidth Inheritance Protocol (BWI), the natural extension of the well-known Priority Inheritance Protocol (PIP) to resource reservation schedulers. The modified protocol allows for a better management of nested critical section, removes unneeded overheads in the management of task block and unblock events, and introduces a run-time deadlock detection mechanism at no cost.

Also, an implementation of the new protocol on the Linux kernel is presented, along with experimental results gathered while running some synthetic application load. Presented results prove the effectiveness of the proposed solution in reducing latencies due to concurrent use of resources and in improving temporal isolation among groups of independent tasks. Also, we show that the introduced overhead is low and negligible for the applications of interest.

1 Introduction

Embedded systems are nowadays part of our everyday life. As their popularity and pervasiveness increases, these devices are required to provide more and more functionality, thus their complexity grows higher and higher. In particular, embedded systems now not only find application in the domain of self-contained, hard real-time, safety critical systems, but their applicability is undergoing a tremendous growth in the range of soft real-time applications, with various degrees of time-sensitiveness and QoS requirements.

The requirements on the real-time operating system platform on which such applications are implemented increases in parallel. The RTOS must be robust (also to timing faults), secure (also to denial of service attacks) and dependable. Finally, it must support open and dynamic applications with QoS requirements.

For these reasons, Linux is becoming the preferred choice for a certain class of embedded systems. In fact, it already provides many of the needed services: it has an open source license and an huge base of enthusiastic programmers as well as a lot of software running on it. Furthermore, it presents a standard programming interface.

Due to the increasing interest from the world of embedded applications, Linux is also being enriched with more and more real-time capabilities [13], usually proposed as separate patches to the kernel, that are progressively being integrated into the main branch. For example, a small group of developers has proposed the PREEMPT_RT patch which greatly reduces the non preemptable sections of code inside the kernel, thus reducing the worst-case latencies. The support for priority inheritance can be extended to in-kernel locking primitives and a great amount of interrupt handling code has been moved to schedulable threads. From a programming point of view, Linux now supports almost entirely the Real-Time POSIX extensions, but the mainstream kernel still lacks support for such extensions like sporadic servers or any Resource Reservation techniques, that would allow the kernel to provide temporal isolation.

Resource Reservation (RR) is an effective technique to schedule hard and soft real-time applications and to provide temporal isolation among them in open and dynamic systems. According to this technique, the resource bandwidth is partitioned between the different applications, and an overrun in one of them cannot influence the temporal behavior of the others.

In standard Resource Reservation theory, tasks are considered as independent. In practical applications, instead, tasks may interact due to the concurrent access to a shared resource, which commonly requires the use of a mutex semaphore in order to serialize those accesses. For example, a Linux application may consist of one multi-threaded process, and threads may interact and synchronize each other through pthread mutexes. In this case, it is necessary to use appropriate resource access protocols in order to bound the priority inversion phenomenon [20]. While implementations of scheduling policies for QoS guarantees on Linux exist [19, 1], they do not provide support for appropriate management of interactions among threads.

*This work has been partially supported supported by the European FRESCOR FP6/2005/IST/5-034026 project.

Contributions of this paper. In a previous work [14], the Bandwidth Inheritance (BWI) Protocol has been proposed as the natural extension of the Priority Inheritance Protocol [20] in the context of Resource Reservations.

In this paper, we extend that work with three important contributions: first, we propose a simplification of the BWI protocol that allows for a much more efficient implementation, for both memory and computational requirements. This is particularly relevant in the context of embedded systems. We also prove that the simplifications do not compromise the original properties of the protocol. Second, we present an efficient deadlock detection mechanism based on BWI, that does not add any overhead to the protocol itself. Third, we present a real implementation of the protocol within the Linux operating system, based on the AQuoSA Framework and the pthread mutex API, that makes the protocol widely available for soft real-time applications.

Also, we present experimental results that highlight the effectiveness of our BWI implementation in reducing latencies.

Organization of the paper The remainder of the paper is organized as follows: Sec. 2 provides some prerequisite definitions and background concepts. Sec. 3 summarizes previous and alternative approaches to the problem. Sec. 4 describes our modification to the BWI protocol, focussing on the achieved improvements. Sec. 5 focusses on details about the actual implementation of the modified protocol on Linux, while Sec. 6 reports results gathered from experimental evaluation of the described implementation. Finally, Sec. 7 draws some conclusions, and quickly discusses possible future work on the topic.

2 Background

2.1 System Model

A real-time task τ_i is a sequence of real-time jobs $J_{i,j}$, each one modeled by an arrival time $a_{i,j}$, a computation time $c_{i,j}$ and an absolute deadline $d_{i,j}$. A periodic (sporadic) task is also associated a relative deadline D_i, such that $\forall j, d_{i,j} = a_{i,j} + D_i$, and a period (minimum inter-arrival time) T_i such that $a_{i,j+1} \geq a_{i,j} + T_i$.

Given the worst case execution time (WCET) is $C_i = \max_j\{c_{i,j}\}$, the processor utilization factor U_i of τ_i is defined as $U_i = \frac{C_i}{T_i}$.

In this paper, we consider *open systems* [9], where tasks can be dynamically activated and killed. In open systems, tasks belonging to different, independently developed, applications can coexist. Therefore, it is not possible to analyze the entire system off-line.

Also, hard real-time tasks must respect all their deadlines. Soft real-time tasks can tolerate occasional violations of their timing constraints, i.e., it could happen that some job terminates after its absolute deadline. The number of missed deadlines over a given interval of time is often used as a valid measure for the QoS experienced by the user.

An effective technique to keep the number of missed deadlines under control is to use Resource Reservation [18, 2] scheduling algorithms. According to these techniques, each task is assigned a virtual resource (vres[1]), with a maximum budget Q and a period P. Resource Reservations provide the *temporal isolation* property to independent tasks: a task is guaranteed to be scheduled for at least Q time units for every period of P time units, but at the same time, in order to provide guarantees to all tasks in the system, the mechanism may not allow the task to execute for more than that amount.

Many RR algorithms have been proposed in the literature, for both fixed priority and dynamic priority scheduling, and the work presented in this paper is applicable to all of them. However, our implementation is based on a variant of the Constant Bandwidth Server.

2.2 Critical Sections

Real-time systems are often designed as a set of concurrent real-time tasks interacting through shared memory data structures. Using classical mutex semaphores in a real-time system is prone to the well known problem of unbounded priority inversion [20]. Dealing correctly with such a phenomenon is very important, since it can jeopardize the real-time guarantees and cause significant QoS degradation and waste of resources. Effective solutions have been proposed in classical real-time scheduling algorithms, such as the Priority Inheritance Protocol (PIP), the Priority Ceiling Protocol (PCP) [20] and the Stack Resource Policy [4].

In particular, PIP is very simple to implement, it can work on any priority-based scheduling algorithm (both for fixed and dynamic priority) and does not require the user to specify additional scheduling parameters. According to PIP, when a task τ_h is blocked trying to access a critical section already locked by a lower priority task τ_l, τ_h lends its priority to τ_l for the duration of the critical section. When τ_l releases the critical section, it is given back its priority. In this paper we discuss an extension of the PIP for resource reservations.

2.3 Constant Bandwidth Server

The Constant Bandwidth Server (CBS [2]) is a well-known RR scheduling algorithm working on top of Earliest Deadline First (EDF [15]). As in any RR algorithm, each task is associated a *virtual resource* with parameters Q (the maximum budget) and P (the period). Each vres can be seen as a sporadic task with worst-case execution time equal to Q and minimum inter-arrival time equal to P. The

[1] We use the term virtual resource instead of the classical term server, to avoid confusion in readers that are not expert of aperiodic servers in real-time scheduling.

EDF system scheduler uses the vres parameters to schedule the associated tasks.

The fundamental idea behind CBS is that each request for a task execution is converted into a request of the corresponding vres with associated a dynamic deadline, calculated taking into account the bandwidth of the vres. When the task tries to execute more than its associated vres budget, the vres deadline is postponed, so that its EDF priority decreases and it is slowed down.

The CBS algorithm guarantees that overruns occurring on task τ_i only affect τ_i itself, so that it can *steal* no bandwidth assigned to any other task. This form of the *temporal protection* property is also called Bandwidth Isolation Property (BIP [2]).

2.4 Critical Sections and the CBS algorithm

Unfortunately, when two tasks belonging to different vres share mutually exclusive resources, the bandwidth isolation property is broken. In fact, one assumption of every RR algorithm is that the vres with the highest priority should be executed at each instant. However, when using mutex semaphores, a task (and its corresponding vres) could be blocked on the semaphore and not able to execute. Even if we are able to bound the blocking time of each vres by using an appropriate resource access protocol like the PIP, we still have to perform a careful off-line analysis of all the critical section code, in order to be able to compute the blocking time and use it in the admission control policy. Moreover, using the PIP with resource reservations is not straightforward, since it is not clear, for example, the budget of which vres should be depleted and the deadline of which should be postponed when an inheritance is in place.

Such a limitation is hard to tolerate in modern systems, since multi-threaded applications are quite common, especially within multimedia environments, and the various tasks often need to communicate by means of shared memory data structures needing mutual exclusive access. For this reason the BandWidth Inheritance protocol (BWI [14]) has been proposed as an extension of PIP suitable for reservation based systems.

2.5 The BandWidth Inheritance Protocol

The description of BWI in this section is not meant to be exhaustive, and the interested reader is remanded to [14] for any further detail. The BWI protocol works according to the following two rules:

BWI blocking rule when a task τ_i blocks trying to access a shared resource R already owned by another task τ_j, then τ_j is added to the list of the tasks served by the vres S_i of τ_i. If τ_j is also blocked, the chain of blocked tasks is followed until one that is not blocked is found, and all the encountered tasks are added to the list of S_i;

BWI unblocking rule when task τ_j releases the lock on R and wakes up τ_i, then τ_j is discarded from the list of vres S_i. If other vres added τ_j to their list, they have to replace τ_j with τ_i.

BWI is considered the natural extension of PIP to resource reservations. In fact, when a task is blocked on a mutex, the lock-owner task inherits its entire vres. In other words, the owner task τ_j can execute on its own vres *and* in the inherited vres (the one with the highest priority), so that the blocking time of τ_i is shortened. Most importantly, the BWI protocol preserves the bandwidth isolation properties between non-interacting tasks.

A blocking chain between two tasks τ_i and τ_j is a sequence $H_{i,j} = \{\tau_1, R_1, \tau_2, \ldots, R_{n-1}, \tau_n\}$ of alternating tasks and resources, such that, the first and the last tasks in the chain are $\tau_1 = \tau_i$ and $\tau_n = \tau_j$, and they access, respectively, resources R_1 and R_{n-1}; each task τ_k (with $1 < k < n$) accesses resource R_k in a critical section nested inside a critical section on R_{k-1}. For example, the following blocking chain $H_{1,3} = \{\tau_1, R_1, \tau_2, R_2, \tau_3\}$ consists of 3 tasks: τ_3 accesses resource R_2 with a mutex m_2; τ_2 accesses R_2 with a critical section nested inside a critical section on R_1; τ_1 accesses R_1. At run-time, τ_1 can be directly blocked by τ_2 and indirectly blocked by τ_3.

Two tasks τ_i and τ_j are *interacting* if and only if there exists a *blocking chain* between τ_i and τ_j. The BWI protocol guarantees bandwidth isolation between pairs of non interacting tasks: if τ_i and τ_j are not interacting, the behavior of τ_i cannot influence the behavior of τ_j and viceversa.

3 Related Work

Many practical implementations of the RR framework have been proposed since now. In the context of general-purpose OSes, the most widely known is probably Linux/RK [19], developed as a research project at CMU and later commercialized by TimeSys. More recently, an implementation of the Pfair [5] scheduling algorithm for reserving shares of the CPU in a multi-processor environment, in the Linux kernel, has been developed by H. Anderson et al. [1] in the $LITMUS^{RT}$ project. However, to the best of our knowledge, these approaches did not provide support for mutually exclusive resource sharing.

A quite common approach [12] when dealing with critical sections and capacity-based servers is to allow a task to continue executing when the vres exhausts its budget and the task is inside a critical section. The extra budget the task has been provided is then subtracted from the next replenishments. Coupling this strategy with the SRP [4] reduces the priority inversion phenomenon to the minimum. The technique has been applied to the CBS algorithm in [6]. In general, this approach is very effective for static systems where all information on the application structure and on the tasks is known a-priori. However, it is not adequate to

open and dynamic systems. In fact, in order to compute the *preemption level* of all the resources, the protocol requires that the programmer declares in advance all the tasks that will access a certain resource. Moreover, the vres parameters have to be fixed, and cannot be easily changed at run-time. Finally, this approach cannot protect the system from overruns of tasks while inside a critical section.

An approach similar to BWI protocol has been implemented in the L4 microkernel [21]. The Capacity-Reserve Donation (CREDO) is based on the idea of maintaining a *task state context* and a *scheduling context* as two, separated and independently switchable data structures. According to the authors, the technique can be applied to either PIP or Stack-Based PCP. Although being quite effective this mechanism is thoroughly biased toward message passing microkernel system architectures, and cannot be easily transposed to shared memory systems. Furthermore, to enable PIP or SRP on top of CREDO, it is necessary to carefully assign the priority of the various tasks in the system, otherwise the protocol can not work properly.

Also, in the context of micro-kernels, Mercer et al. [16] presented an implementation of a reservation mechanism on the real-time Mach OS. Interestingly, the implementation allowed for accounting the time spent within kernel services, when activated on behalf of user-level reserved tasks, to the reserves of the tasks themselves. Kernel services might be considered, in such context, as shared resources to which applications concurrently access. However, the problem of regulating application-level critical sections managed through explicit synchronization primitives, so to avoid priority inversion, is not addressed.

All this given, we say BWI is a suitable protocol for resource sharing in open, dynamic embedded systems, for the following reasons:

- BWI is completely transparent to applications. As with the PIP, the user must not specify additional scheduling parameters, as "ceiling" or "preemption-level";
- BWI provides bandwidth isolation between non-interacting tasks, even in the case of tasks that overrun inside a critical section. Therefore, it is not necessary to implement any additional budget protection mechanism for the critical section length;
- BWI is neutral to the underlying RR algorithm and does not require any special property of the scheduler. This allows us any modification of the scheduling algorithm without the need to reimplement BWI.

4 Improvements to the protocol

In this section, we focus on the limitations of the original formulation of the BWI protocol, and propose two modifications to its rules that, without compromising the guarantees, allow for a simplification of the implementation. In what follows, we assume that each task competing for access to shared resources is served by a dedicated vres.

4.1 Nested Critical Section Improvement

In presence of nested critical sections, the two BWI rules do not correctly account for all possible situations. Consider the case of a task τ_i that blocks on another task τ_j, after having been added to some vres S_h, different from its original one (S_i), due to previous inheritance (i.e., another task τ_h is blocked waiting for τ_i to release some lock). By following the blocking rule of BWI, task τ_j is added only to S_i, but, because also τ_h is blocked waiting for τ_i, the blocking delays induced on τ_h may be reduced if τ_i would have added to S_h as well.

In general, we are saying that τ_j should be attached to all the vres to which τ_i was bound (both directly and by inheritance due to BWI) before blocking itself.

As an example, consider the situation depicted in Figure 1, where we have four tasks, τ_A, τ_B, τ_C and τ_D, each bound to its own vres. S_A has $U_{S_A} = 6/25$ utilization, S_B has $U_{S_B} = 3/20$ utilization, $U_{S_C} = 3/15$ and $U_{S_D} = 4/10$. τ_A, τ_C and τ_D use mutex m_1 and τ_A and τ_B use mutex m_2. Also notice τ_A acquires the second mutex while holding the first one (nested critical section).

In this figure, and in all the figures of Sec. 6, symbols $L(_i)$ and $U(_i)$ denote wait and signal operations on mutex m_i. Light-gray filled rectangles denote tasks executing inside a critical section, with the number in the rectangle being the acquired mutex. Vertical dashed arrows denote the time instants a task is attached (and detached) to a vres different from its own one due to BWI inheritance. White filled ones denote a task being able to execute inside a vres different from its original one thanks to BWI, with the number in the rectangle being the mutex that caused the inheritance.

At time $t = 7$ τ_C blocks on mutex m_1, owned by τ_A, and τ_A is added to the vres of τ_C. Then, at time $t = 11$, τ_D blocks on mutex m_1 too. Again, τ_A is attached to the vres of τ_D, and it can now run inside any of the three vres exhausting their budget on time instants $t = 9$ (for τ_C) and $t = 13$ (for τ_D).

Suppose that at time $t = 15$ τ_A blocks on mutex m_2: honoring the original BWI rule, τ_B is added only to the vres of τ_A, whereas τ_A remains attached to the vres of τ_D and τ_C, although being blocked. In this way, τ_B can not take advantage of the bandwidth assigned to τ_C and τ_D, delaying their own unblocking. Notice this behavior is incidental: if τ_C and τ_D would start executing *after* τ_A blocks on m_2, then τ_B would have been added to all the vres of the three tasks.

4.2 Simplified BWI blocking rule

In order to face with the just shown issue, we propose a rewrite of the BWI blocking rule as follows:

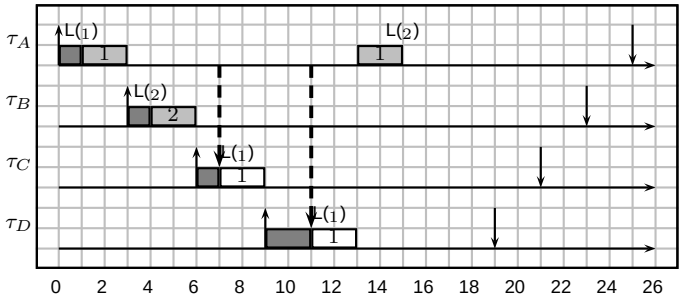


Figure 1: example of nested blocking situation not correctly handled by the original BWI protocol

new BWI blocking rule when a task τ_i blocks while trying to access a shared resource R already owned by another task, the chain of blocked tasks is followed until one that is not blocked is found, let it be τ_j. Then, τ_j is added to the vres τ_i belongs to. Furthermore, τ_i is replaced with τ_j in all vres that previously added τ_i to their task list due to this rule, if any.

Basically, the difference between the original and the new BWI formulation may be highlighted in terms of the invariant property of the algorithms. To this purpose, consider the wait-for-graph at an arbitrary point in time, where nodes represent tasks and an edge from node τ_A to node τ_B denotes that some resource is held by τ_B and requested by τ_A. Now, consider any ready-to-run task τ_j, and let G_j denote the set of tasks directly or indirectly blocked on mutexes held by τ_j. Well, the new BWI formulation has the invariant property that the running task is bound to all the vres of the tasks in G_j, so that each vres of any such task needs to bound dynamically (due to BWI) at most one task (the ready-to-run one, τ_j), in addition to the one (blocked) explicitly bound to it. Furthermore, the original BWI formulation also required each of these vres to be dynamically bound (due to BWI) to all (blocked) tasks found in the blocking chain: from the task explicitly bound to it up to τ_j.

Therefore, the following property holds for the new BWI formulation:

BWI one-blocked-task property Given a vres S_i, at each time instant it can have in its list of tasks only one other task in addition to its original task τ_i.

This is achieved because, according to the new blocking rule, every time a task τ_i blocks, if its mutex-owner is blocked too, we need to follow the blocking chain until a ready task τ_j is found, and add it to S_i. Obviously a mechanism which make it possible to traverse such a chain of blocked task has to be provided by the operating system (as the Linux kernel does). Furthermore, if τ_i was on its own already bound to other servers due to BWI, we need to replace τ_i with τ_j in those servers, keeping at 1 the number of additionally bound tasks in those servers. Since each task τ_i can, at time t, be blocked at most by only one other task τ_j, the just stated property always holds.

As an example if we have:

- task τ_A owning m_1 and running;
- task τ_B owning m_2 and blocked on m_1 (owned by τ_A);
- task τ_C owning m_3 and blocked on m_2 (owned by τ_B);

when a fourth task, task τ_D, tries to lock m_3, it blocks. According to original BWI protocol, we have to bind τ_C, τ_B and τ_A to the vres of τ_D. In the new formulation we only bind τ_A (the sole running task).

The main consequence of the new blocking rule on the implementation is lower memory occupation of the data structure needed by BWI, what is particularly relevant mainly in the context of embedded systems, especially when the task set is characterized by tight interactions and nested critical sections would cause the run-time creation of non-trivial blocking trees. In fact, even this is not necessarily of practical relevance (in a well-designed system interactions should be kept at the bare minimum), the memory overhead complexity of the new BWI formulation is *linear* in the number of interacting tasks, while in the original BWI formulation it was *quadratic*.

4.3 Correctness

The original description of the BWI protocol [14] was accompanied by a proof of correctness in the domain of hard real-time systems. The proof aimed at showing that, if the maximum interferences of the interacting tasks is properly accounted for in the admission control test, then the system scheduled with BWI allows all tasks to respect their deadlines. The following result constitutes the basis for the proof:

Lemma 1. *Each active vres always has exactly one ready task in its list.*

Proof. The lemma is clearly valid before any task blocks. Assume that the lemma holds true until time t, when task τ_i blocks on a resource, and, following the chain of blocked tasks, τ_j is selected. Our modified version of the BWI blocking rule only differs from the original one in stating that the running task τ_j replaces τ_i in *all* the lists of the vres where task τ_i has been bound (as opposed to only the last one). Notice that τ_i has been added to these vres because of one running task in each of them blocked on a resource directly or indirectly shared with τ_i. Thus, just before blocking, if the lemma is true, τ_i was the only ready task in all of them. Since τ_i is blocking, τ_j becomes on its turn the only runnable task in every vres, so the lemma continues to hold. □

Although this result may be used to prove that the new protocol is still correct, further details are omitted for the sake of brevity.

4.4 Lightweight Deadlock Detection

Finally, we improved the BWI protocol by adding the ability to detect deadlocks at run-time [2]:

Deadlock detection rule when applying the new BWI blocking rule to a task τ_i that blocks, for each task τ_k encountered while iterating the blocking chain of τ_i, if $\tau_k = \tau_i$, then a deadlock attempt is detected.

This is a lightweight yet effective approach for deadlock detection and below is a proof of correctness of it.

Theorem 1. *If there is a deadlock situation, the protocol detects it at run-time.*

Proof. As stated by Coffman et al. [7], necessary condition for deadlock is that there is a cycle in the wait-for-graph of the system. To detect a deadlock, every time a task τ_i blocks on another task τ_j, we have to add an edge from τ_i to τ_j in the graph and check if a cycle has been created. Suppose that just before time t there are no cycles, and so no deadlock is in place, and that at time t task τ_i blocks. Also consider the fact that, from any task, at most one edge can exit, directed toward the task's lock-owner. Therefore, if a cycle has been created by the blocking of task τ_i, then τ_i must be part of the cycle. Hence, following the blocking chain from τ_i, if a deadlock has been created, we will come back to τ_i itself, and so our algorithm can detect all deadlocks. □

It is noteworthy that the deadlock detection rule may be realized with practically zero overhead, adding a comparison in the search for a ready-to-run task, while we are following the chain of blocked tasks, according with the blocking rule.

5 BandWidth Inheritance Implementation

5.1 The Linux Kernel

Although not being a real-time system, the 2.6 Linux kernel includes a set of features making it particularly suitable for *soft* real-time applications. First, it is a fully-preemptable kernel, like most of the existing real-time operating systems, and a lot of effort has been spent on reducing the length of non-preemptable sections (the major source of kernel latencies). It is noteworthy that the 2.6 kernel series introduced a new scheduler with a bounded execution time, resulting in a highly decreased scheduling latency. Also, in the latest kernel series, a modular framework has been introduced that will possibly allow for an easier integration of other scheduling policies. Second, although being a general-purpose time-sharing kernel, it includes the POSIX priority-based scheduling policies `SCHED_FIFO` and `SCHED_RR`, that may result useful for real-time systems. Third, the recently introduced support in the kernel mainstream of the support for high-resolution timers is of paramount importance for the realization of high-precision customized scheduling mechanisms, and for the general performance of soft real-time applications.

Unfortunately, the Linux kernel has also some characteristics that make it impossible to realize hard real-time applications on it: the monolithic structure of the kernel and the wide variety of drivers that may be loaded within, the impossibility to keep under control all the non-preemptable sections possibly added by such drivers, the general structure of the interrupt management core framework that privileges portability with respect to latencies, and others. However, the wide availability of kernel drivers and user-space libraries for devices used in the multimedia field constitutes also a point in favor of the adoption of Linux in such application area. Furthermore, recent patches proposed by the group of Ingo Molnar to the interrupt-management core framework, aimed at encapsulating device drivers within kernel threads, are particularly relevant as such approaches would highly increase predictability of the kernel behaviour.

At the kernel level, mutual exclusive access to critical code sections is managed in Linux through classical spinlocks, RCU primitives, mutexes and rt-mutexes, a variant of mutexes with support for priority inheritance. The latter ones are particularly worth to cite, because they allow for the availability of the PIP in user-level synchronization primitives. This is not only beneficial for time-sensitive applications, since thanks to the rt-mutex run-time support, we have been able to implement the BandWidth Inheritance protocol without any modification to the kernel mutex-related logics.

At the user/application level, locking and synchronization may be achieved by means of futexes (Fast Uerspace muTEX [11]) or of standard POSIX mutexes, provided by the GNU C Library [10] and, on their turn, implemented through futexes. One remarkable peculiarity of futexes and POSIX mutexes, is that their implementation on Linux involves the kernel (thus a relatively expensive system call) only when there is a contention that requires arbitration.

5.2 The AQuoSA Framework

The CPU scheduling strategies available in the standard Linux kernel are not designed to provide temporal protection among applications, therefore they are not suitable for time-sensitive workloads. The AQuoSA framework [17] (available at http://aquosa.sourceforge.net) aims at filling this gap, enhancing a standard GNU/Linux system with scheduling strategies based on the RR techniques described in Sec. 2.3.

[2] The method proposed here differs from the method proposed in [14], as it is much more efficient.

AQuoSA is designed with a layered architecture. At the lowest level[3] there is a small patch (Generic Scheduler Patch, GSP) to the Linux kernel that allows dynamically loaded modules to customize the CPU scheduler behaviour, by intercepting and reacting to scheduling-related events such as: creation and destruction of tasks, blocking and unblocking of tasks on synchronization primitives, receive by tasks of the special `SIGSTOP` and `SIGCONT` signals). A Kernel Abstraction Layer (KAL) aims at abstracting the higher layers from the very low-level details of the Linux kernel, by providing a set of C functions and macros that abstract the needed kernel functionalities. The Resource Reservation layer (RRES) implements a variant of the CBS scheduling policy on top of an internal EDF scheduler. The QoS Manager layer allows applications to take advantage of Adaptive Reservations, and includes a set of bandwidth controllers that can be used to continuously adapt the budget of a vres according to what an application needs. An user-space library layer allows to extend standard Linux applications to use the AQuoSA functionality without any further restriction imposed on them by the architecture.

Thanks to an appropriately designed access control model [8], AQuoSA is available not only to the *root* user (as it happens for other real-time extensions to Linux), but also to non-privileged users, under a security policy that may be configured by the system administrator.

An interesting feature of the AQuoSA architecture is that it does not replace the default Linux scheduler, but coexists with it, giving to soft real-time tasks a higher priority than any non-real-time Linux task. Furthermore, the AQuoSA architecture follows a non-intrusive approach [3] by keeping at the bare minimum (the GSP patch) the modifications needed to the Linux kernel.

5.3 Bandwidth Inheritance Implementation

Design goals and choices The implementation of the BWI protocol for AQuoSA has been carried out with the following design objectives:

- to provide a full implementation of BWI;
- to allow for compile-time disabling of BWI;
- to allow the use of BWI on a per-mutex basis;
- to impact as low as possible on the AQuoSA code;
- to have as little as possible run-time overheads.

In order to achieve such goals, our implementation:

- uses the C pre-processor in order to allow compile-time inclusion or exclusion of the BWI support within AQuoSA;
- does not modify the Linux kernel patch (i.e., BWI is entirely implemented inside the kernel modules);
- does not modify the libraries and the APIs;
- does not remove or alter the core algorithms inside the framework, especially with respect to:
  - scheduling: it is not necessary to modify the implementation of various scheduling algorithms available inside AQuoSA;
  - vres queues: we do not modify the task queues handling, so that the old routines continue to work seamlessly;
  - blocking/unblocking: when BWI is not required/enabled the standard behaviour of AQuoSA is not modified by any means.

Using BWI Since BWI is the natural extension of PIP for RR-based systems, in our implementation the protocol is enforced every time two or more tasks, with scheduling guarantees provided through AQuoSA RR vres, also share a POSIX mutex that has been initialized with `PTHREAD_PRIO_INHERIT` as its protocol. This way the application is able to choose to use BWI or not on a per-mutex basis. Furthermore, all the code already using the Priority Inheritance Protocol automatically benefits of BandWidth Inheritance, if the tasks are attached to some vres.

Deadlock detection Once a deadlock situation is detected, the current implementation may be configured for realizing one of the following behaviors: 1) the system forbids the blocking task from blocking on the mutex, and returns an error (`EDEADLK`); 2) the system logs a warning message notifying that a deadlock took place.

5.4 Implementation Details

The BWI code is completely integrated inside the AQuoSA architecture and only entails very little modification to the following software components:

- the KAL layer, where the KAL API has been extended with macros and functions exploiting the in-kernel rt-mutexes functionality, for the purpose of providing, for each task blocked on an rt-mutex [4], the rt-mutex on which it is blocked on, and the task that owns such an rt-mutex;
- the RRES: where the two BWI rules are implemented.

The core of the BWI implementation is made up of a few changes to the AQuoSA data structures, and of only four main functions (plus a couple of utility ones):

1. `rres_bwi_attach()`, called when a task is attached to a vres;

[3] For a more detailed description, the interested reader may refer to [17].

[4] Note that such information is available inside the kernel only for rt-mutexes, for the purpose of implementing PIP.

2. `rres_bwi_block()`, called when a task blocks on an rt-mutex, that enforces the new BWI blocking rule (Sec. 4.1);
3. `rres_bwi_unblock()`, called when a task unblocks from an rt-mutex, that enforces the BWI unblocking rule;
4. `rres_bwi_detach()`, called when a task is detached from a vres.

The produced code can be found on the AQuoSA CVS repository, temporarily residing in a separate development branch. It will be merged soon in the very next releases of AQuoSA. It has been realized on top of the 2.6.21 kernel release and tested up to the 2.6.22 release.

In Tab. 1 the impact of the implementation on the source code of AQuoSA is briefly summarized.

	added	modified	removed
source files	2	0	0
lines of code	260	6	0

Table 1: Impact of our modification on AQuoSA sources.

6 Experimental evaluation

In this section we present some results of the experiments we ran on a real Linux system, with our modified version of AQuoSA installed and running, and with a synthetic workload provided by ad-hoc designed programs. These experiments are mainly aimed at highlighting features of the BWI protocol under particular blocking patterns, and gathering the corresponding overhead measurements.

6.1 Examples of execution

In the first example we have two tasks, τ_A and τ_B, sharing a mutex. τ_A has a computation time of $2\,msec$ and a period of $5\,msec$, and is given a reservation of $2\,msec$ every $5\,msec$. τ_B has a computation time of $6\,msec$ and a period of $15\,msec$, and is given a reservation of $2\,msec$ every $5\,msec$. In Fig. 2 we show the two schedules obtained with (bottom schedule) and without (top schedule) BWI.

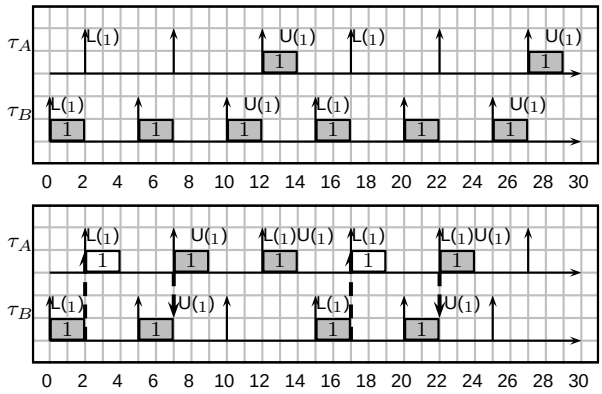


Figure 2: BWI effectiveness in reducing the tardiness

Notice that the use of BWI notably reduces the tardiness of both tasks, improving the system performance. This comes from the fact that τ_B always arrives first and grabs the lock on the shared mutex. If BWI is not used, then τ_A is able to lock the mutex and run only after τ_B ran for $2\,msec$ each $5\,msec$ time interval and completed its $6\,msec$ execution (at which instant it releases the lock). Thus, τ_A skips repeatedly the opportunity to run every two vres instances out of three: quite an outstanding waste of bandwidth. If BWI is in place, as soon as τ_A arrives and blocks on the mutex, τ_B is attached to its vres and completes execution much earlier, so that τ_A is now able to exploit two vres instances out of three, and the system does not waste any reserved bandwidth at all.

In Fig. 3 we show how the protocol is able to effectively enforce bandwidth isolation, by means of an example consisting of 5 tasks: τ_A and τ_C sharing m_1, τ_D and τ_E sharing m_2, and τ_B. Notice that τ_A does not share any mutex with τ_E. Also, τ_B has an earlier deadline than τ_C and τ_E, but a later one than τ_A, and this is a possible cause of priority inversion. When BWI is not used (top schedule), after τ_C and τ_E having locked m_1 and m_2 (respectively), they are both preempted by τ_B, and the inversion occurs. Furthermore, as a consequence of τ_E succeeding in locking m_2, since it has earlier deadline than τ_C, τ_A misses its deadline, which means bandwidth isolation is not preserved.

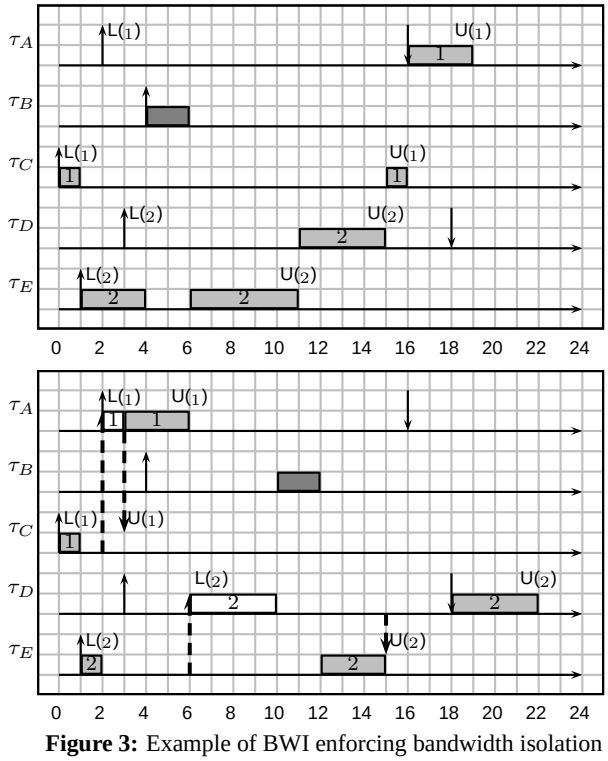


Figure 3: Example of BWI enforcing bandwidth isolation

On the other hand, when using BWI (bottom schedule), priority inversion is bounded, since τ_B is no longer able to

preempt τ_C nor τ_E. Moreover, the behaviour of τ_E and τ_D can only affect themselves, between each others, and can no longer cause τ_A to miss its deadline, and this is the correct behaviour. Indeed, τ_D and τ_E are interacting tasks, and it is not possible to provide reciprocal isolation between them.

In the last example, in Fig. 5, we demonstrate the effect of BWI on bandwidth and throughput. We see (with no reclamation policy enabled) the protocol removes the waste of bandwidth due to blocking. In fact, while a task is blocked the bandwidth reserved for its vres can not be exploited by anyone else, if BWI is not in place. This is not the case if we use BWI, since when a task blocks its lock-owner is bound to such a vres and can consume the reserved bandwidth. Furthermore, thanks to our modification to the blocking rule (Sec. 4.1), this is also true in case of nested critical sections. For this example we used eight tasks, τ_A, τ_B, ..., τ_H. The mutexes are five with τ_A using m_0, m_1 and m_2; τ_B using m_2, m_3 and m_4; τ_C using m_1 and m_4; τ_D using m_1; τ_E using m_4; τ_F using m_2; τ_G using m_0; τ_H using m_0 too. Each task τ_i is bound to a vres with $U_i = 10/100$ (10% of CPU). The locking scheme is choose to be quite complex, in order to allow blocking on nested critical sections to occur. As an example of this in Fig. 4 the wait for graph at time $t = 40\,sec$, when all the tasks but τ_A are blocked, is depicted.

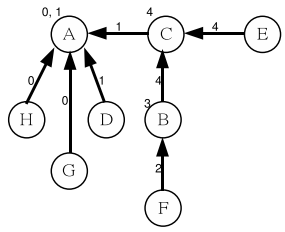


Figure 4: Wait-for graph for the example in Fig. 5. The numbers beside each task are the mutex(es) it owns. The number next to each edge is the mutex the task is waiting for.

Coming back to Fig. 5, the thick black line is the total bandwidth reserved, for each time instant t, for all the active vres. The thin black horizontal line represents the average value of the bandwidth. The thick gray line, instead, is the CPU the various running tasks are actually using and the thin gray line is its mean value. The thick black curve stepping down means a task terminated and its vres being destroyed, and so time values on the graphs are finishing times.

Comparing the two graphs it is evident that, when BWI is used (left part), 100% of the *reserved* CPU bandwidth is exploited by the running tasks, both instantaneously and on average. On the contrary, without BWI (right part) there exist many time instants during which the bandwidth that the running tasks are able to exploit is much less than what it has been reserved at that time, and the mean value is notably lower than the reserved one too. This means some reserved bandwidth is wasted. Finally, notice finishing times are are much smaller with BWI enabled.

Event	Max. exec. (μsec)		Avg. exec. (μsec)	
`blocking`	BWI unused	0	BWI unused	0.01
	used	1	used	0.169
`unblocking`	unused	0	unused	0.052
	used	3	used	0.116

Table 2: Max and mean execution times, with and without BWI

6.2 Overhead evaluation

We also evaluated the computational overhead introduced by our implementation. We ran the experiments described in the previous section on a desktop PC with 800 MHz *Intel(R) Centrino(TM)* CPU and 1GB RAM and measured mean and maximum times spent by AQuoSA in correspondence of task block and unblock event handlers, either when `PTHREAD_PRIO_INHERIT` was used and not.

BWI	context switches #	
unused	task τ_A	26
	task τ_B	34
used	task τ_A	25
	task τ_B	34

Table 3: context switch number with and without BWI using periodic sleeping tasks

Tab. 2 shows the difference between the measured values with respect to the ones obtained when running the original, unmodified version of AQuoSA (average values of all the different runs) [5].

BWI	context switches #	
unused	task τ_A	607
	task τ_B	414
	task τ_C	405
used	task τ_A	343
	task τ_B	316
	task τ_C	405

Table 4: Context switch number with and without BWI using greedy tasks.

As we can easily see, the introduced overhead is negligible for tasks not using the protocol. Anyway, also when BWI is used, the overhead is in the order of one tenth of microsecond, and this is definitely an acceptable result.

With respect to context switches, we see in Tab. 3 that the protocol has practically no effect if typical real-time tasks, with periodic behaviour, are considered.

On the contrary, if "greedy" tasks (i.e., tasks always running without periodic blocks) are used, Tab. 4 shows that the number of context switches they experience is dramatically smaller when using BWI. This is due to the bonus bandwidth each task gets thanks to the protocol.

7 Conclusions and Future Work

In this paper, we presented an improved version of the BWI protocol, an effective solution for handling critical sec-

[5] Note that, in the latter case, the use of `PTHREAD_PRIO_INHERIT` by tasks implies the use of the original PIP protocol, not the BWI one.

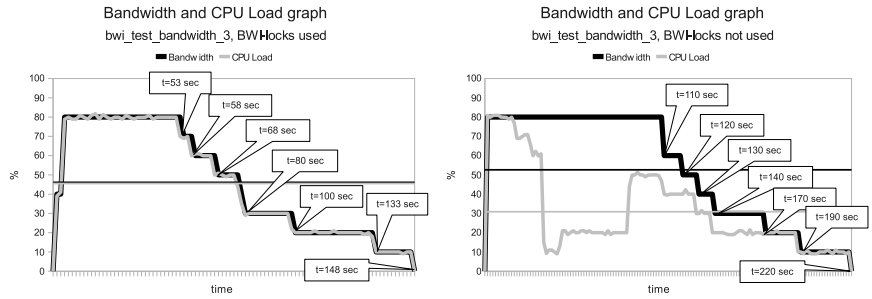


Figure 5: Resource usage with BWI

tions in resource reservation based systems. We also proposed an implementation of the protocol inside AQuoSA, a reservation framework working on Linux. Finally, we ran some experiments in order to evaluate the overhead the protocol introduces when used on such a concrete system.

Our modifications improve correctness and predictability of BWI, enable deadlock detection capabilities and enforce better handling of nested critical sections. The implementation is lean, simple and compact, with practically no need of modifying the framework core algorithms and structures. The experimental results show this implementation of BWI is effective in allowing resource sharing and task synchronization in a real reservation based system, and also has negligible overhead.

Regarding future works, we are investigating how to integrate ceiling like mechanisms inside the protocol and the implementation, in order to better deal with the problem of deadlock, so that we can prevent instead of only check for it. Work is also in progress to modify a real multimedia application so that it will use the AQuoSA framework and the BWI protocol. This way we will be able to show if our implementation is useful also inside real world applications with their own blocking schemes.

Other possible future works include the investigation of more general theoretical formulation to extend the RR methodologies and the BWI protocol to multiprocessor systems. Also, it would be interesting to adapt the AQuoSA framework to the `PREEMPT_RT` kernel source tree, so to benefit from its interesting real-time features, especially the general replacement, within the kernel, of classical mutexes with rt-enabled ones.

References

[1] Linux Testbed for Multiprocessor Scheduling in Real-Time Systems ($LITMUS^{RT}$). `http://www.cs.unc.edu/ anderson/litmus-rt/`.

[2] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proc. IEEE Real-Time Systems Symposium*, pages 4–13, Madrid, Spain, Dec. 1998.

[3] L. Abeni and G. Lipari. Implementing resource reservations in linux. In *Real-Time Linux Workshop*, Boston (MA), Dec. 2002.

[4] T. P. Baker. Stack-based scheduling of real-time processes. *Real-Time Systems*, (3), 1991.

[5] S. Baruah, N. Cohen, C. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 6, 1996.

[6] M. Caccamo and L. Sha. Aperiodic servers with resource constraints. In *IEEE Real Time System Symposium*, London, UK, December 2001.

[7] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, 1971. ISSN 0360-0300.

[8] T. Cucinotta. Access control for adaptive reservations on multi-user systems. In *Proc. 14^{th} IEEE Real-Time and Embedded Technology and Applications Symposium (to appear)*, April 2008.

[9] Z. Deng and J. W. S. Liu. Scheduling real-time applications in open environment. In *Proc. IEEE Real-Time Systems Symposium*, Dec. 1997.

[10] U. Drepper and I. Molnar. The native posix thread library for linux. Technical report, Red Hat Inc., February 2001.

[11] H. Franke, R. Russel, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *Ottawa Linux Symposium*, 2002.

[12] T. M. Ghazalie and T. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems*, 9, 1995.

[13] G. Lipari and C. Scordino. Current approaches and future opportinities. In *International Congress ANIPLA 2006*. ANIPLA, November 2006.

[14] G. Lipari, G. Lamastra, and L. Abeni. Task synchronization in reservation-based real-time systems. *IEEE Trans. Computers*, 53 (12):1591–1601, 2004.

[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, Jan. 1973.

[16] C. W. Mercer, S. Savage, and H. Tokuda. Processor Capacity Reserves: An Abstraction for Managing Processor Usage. In *Proc. 4th Workshop on Workstation Operating Systems*, 1993.

[17] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari. AQuoSA – adaptive quality of service architecture. *Software: Practice and Experience*, on-line early view, 2008.

[18] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource Kernels: A Resource-Centric Approach to Real-Time and Multimedia Systems. In *Proc. Conf. on Multimedia Computing and Networking*, January 1998.

[19] R. R. Rajkumar, L. Abeni, D. de Niz, S. Ghosh, A. Miyoshi, and S. Saewong. Recent Developments with Linux/RK. In *Proc. 2nd Real-Time Linux Workshop*, Orlando, Florida, november 2000.

[20] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9), September 1990.

[21] U. Steinberg, J. Wolter, and H. Hartig. Fast component interaction for real-time systems. In *Proc. 17^{th} Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 89–97, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2400-1.

Energy Models of Real Time Operating Systems on FPGA

Saadia Dhouib, Jean-Philippe Diguet, Eric Senn and Johann Laurent
surname.name@univ-ubs.fr
European University of Brittany - UBS
Lab-STICC, CNRS/UMR 3192
F-56321 Lorient Cedex, France

ABSTRACT

This paper introduces a methodology for modeling power and energy consumption of embedded systems running operating systems. We notice that internal services of the operating system such as interprocess communications, scheduling, context switches are not the major cause of power and energy consumption in an embedded system.

Based on this observation, we have applied our methodology to embedded peripheral devices managed by the operating system. The proposed model is general. In this paper, it is illustrated by the Ethernet standard peripheral device. We analyze the key parameters affecting its power and energy consumption and focus on the relationship between energy consumption and software and hardware parameters like the transmission protocol and the frequency of the processor. Then, we propose a power and energy consumption model, the parameters of which are set after real measurements.

Experimental results are presented for Montavista Linux, an RTOS ported and executed on the XUP Virtex-II pro development board embedding a powerpc processor.

General Terms

Keywords

1. INTRODUCTION

Energy and power consumption are significant constraints in the design of embedded systems. To reduce the power and energy consumption of these systems, it is necessary to estimate the energy consumption of the system components at the first design phases, when implementation decisions have not been made yet. This requires high level power and energy models.

The increasing complexity of embedded system applications has led to the use of operating systems which have hard and/or soft real time constraints. They are considered as a software layer between the system resources (processor, memory and peripheral devices) and the applicative tasks. Limited power budgets of embedded systems have made it necessary to consider the OS influence on power and energy dissipation. The OS sources of consumption could be either the internal services (scheduling, context switch, IPC,...) or device management services.

In a previous study [6], we introduced high level power models for embedded processors. OS routines consumption has been estimated this way. Now, we propose a methodology to model consumption of peripheral devices managed by embedded operating systems.

The methodology is based on physical measurements realized on a standard reconfigurable board, namely, Xilinx Virtex II board that contains a RTOS, Montavista Linux, running on a PowerPC processor. The trend for variable-frequency designed processors has led us to consider the frequency of the processor as one of the model parameters. We have selected a standard peripheral device as a case study: the Ethernet interface.

The remainder of the paper is organized as follows. Section 2 gives an overview of related work. Section 3 describes our methodology. Section 4 describes the hardware and software frameworks for experiments. Section 5 presents the energy and power consumption model of the Ethernet interface managed by the RTOS routines and drivers. Finally, we summarize our solution and conclude in section 6.

2. RELATED WORK

Much of the work on energy and power consumption modeling and estimation of real time operating systems [3, 4, 7] only considers the internal services of the OS.

Analysis and/or Modeling of power consumption due to peripheral devices management was treated by few research works [1], even though it corresponds to the most important part.

Tan et al. [7] were the first authors who dealt with *modeling* the energy consumption of operating systems at the kernel level. They derived energy consumption macro models for two embedded OS's, μC/OS and Linux OS. However, they did not develop models for I/O driver energy consumption.

Dick et al. [4] analyzed the energy consumption of the μC/OS RTOS when running several embedded applications. They targeted a Fujitsi SPARClite processor based embedded system. This work represents only an analysis of RTOS policies on embedded system power consumption. The authors did not develop an energy consumption model.

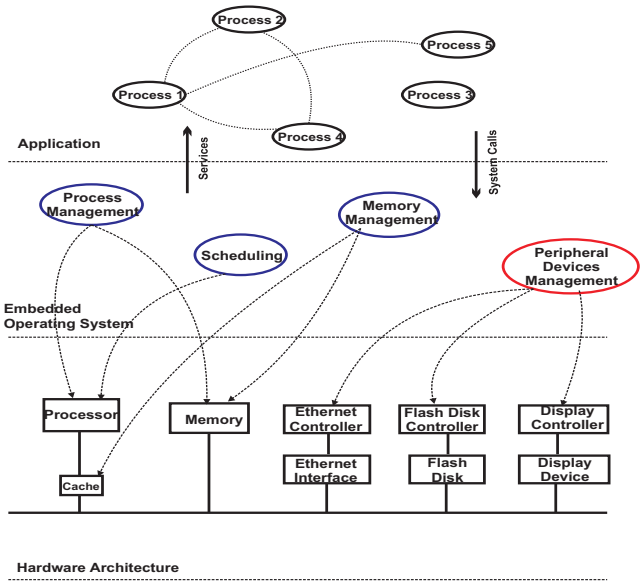


Figure 1: Interactions between Applicative tasks, Embedded Operating System and Hardware Architecture

Acquaviva et al. [1] characterized the energy consumption of the eCos Real Time Operating System running on a prototype wearable computer, HP's SmartBadgeIII. They analyzed the energy impact of the RTOS both at the kernel and the Audio Driver Level. This work focused on the relationship between energy consumption and processor frequency. The authors analyzed but did not model the energy consumption of internal services and I/O drivers of the operating system.

Vahdat et al. [8] conducted a general study on aspects of Operating System design and implementation to improve energy efficiency. They investigated low power modes of embedded devices and proposed energy efficient techniques to use operating system functionalities.
A comparative study of different approaches is exposed in table 1. From this table, we can see that almost no energy consumption models address memory and peripheral device management. Since energy consumption of OS internal services is insignificant in comparison with the energy consumption of I/O devices access [9], we are interested in analyzing and modeling Embedded Operating System management of I/O devices access by applicative tasks.

3. RTOS ENERGY AND POWER CONSUMPTION MODELING AND ESTIMATION

The purpose of our approach is to model the energy and power consumption of an embedded operating system managing several embedded devices such as Ethernet interface, flash memory. Interactions between applicative tasks, embedded OS and Hardware Architecture are showed in Fig.1.

3.1 Microblaze and μC/OSII case study

We conducted a study on the energy consumption of RTOS internal services using a Microblaze core processor and the μC/OSII RTOS. The target FPGA is embedded in the Xilinx Virtex II pro development board. We measured the average power and energy dissipation of the RTOS primitives by repeatedly calling them in test programs. Furthermore, we characterized the energy dissipation of the scheduler. The experimental results, presented in Table 2, showed that power consumption does not vary so much with RTOS primitives. Actually, power consumption is usually observed constant with any application executing on a scalar processor [2]. Energy dissipation varies with T_{exec} of basic services. The most important source of dissipation is task creation primitive which only occurs at start time (no dynamic task creation with μC/OSII). In the case of scheduler, we notice that energy dissipation is increasing due to idle times introduced by longer scheduling period.

Table 2: Energy and power dissipation of μC/OSII services

RTOS service	Average Power consumption	Average Energy consumption
Task creation	454mW	$100\mu J$
Mailbox creation	439mW	$1{,}2\mu J$
Message queue creation	435mW	$1{,}74\mu J$
Mutex creation	438mW	$1{,}35\mu J$
Semaphore creation	441mW	$1{,}25\mu J$
Memory partition creation	430mW	$1{,}88\mu J$
Scheduler (tick period=10ms)	461mW	$5{,}34\mu J$
Scheduler (tick period=20ms)	459mW	$11{,}17\mu J$
Scheduler (tick period=30ms)	458mW	$17{,}72\mu J$

3.2 Rational for energy consumption modeling of RTOS management of peripheral devices

We make power consumption and performance studies on a typical image processing application using the μC/OSII and the Microblaze soft core. The application contains 7 tasks, calls semaphores, mailboxes and generates 14 context switches per application period. We measured the RTOS temporal overhead, it represents 0,2% of the total execution time. Consequently the energy overhead is the same.
Furthermore, in a study performed by A. Weissel [9], power measures realized on an iPAQ3970 handheld have shown that the processor and memory consumption part is 23% of the whole system power dissipation, LCD part is 18%, wireless interface is 31% and hard disk is 26%. From those results, we can see that peripheral devices contribute significantly to total power consumption.
We can conclude that the energy consumption of the internal services of the RTOS is not significant, especially in the case of data flow applications. On the other hand, peripheral devices managed by the OS are a significant source of power and energy dissipation.

Table 1: Comparison between embedded OS's Energy Consumption Models

Model	OS Services						Energy Consumption Analysis	
	System Calls	Inter Process Communications	Context Switch	Scheduling	Memory Management	Peripheral Device Management	Analysis	OS/Processor
A. Acquaviva et al. [1]	characterized	not characterized	characterized	not characterized	not characterized	characterized (audio driver)	physical measures	eCos / StrongARM 1100
K. Baynes et al. [3]	characterized	characterized	characterized	characterized	not characterized	not characterized	Simulation	μC/OS Echidna NOS / Motorola MCORE
R.P. Dick et al. [4]	characterized	characterized	characterized	characterized	not characterized	not characterized	physical measures	μC/OS / Sparclite
Tan et al. [7]	modeled	modeled	modeled	modeled	not modeled	not modeled	Simulation	μC/OS / Sparclite
A. Weissel [9]	modeled with Processor/Memory events					modeled	physical measures	Linux / Intel XScale

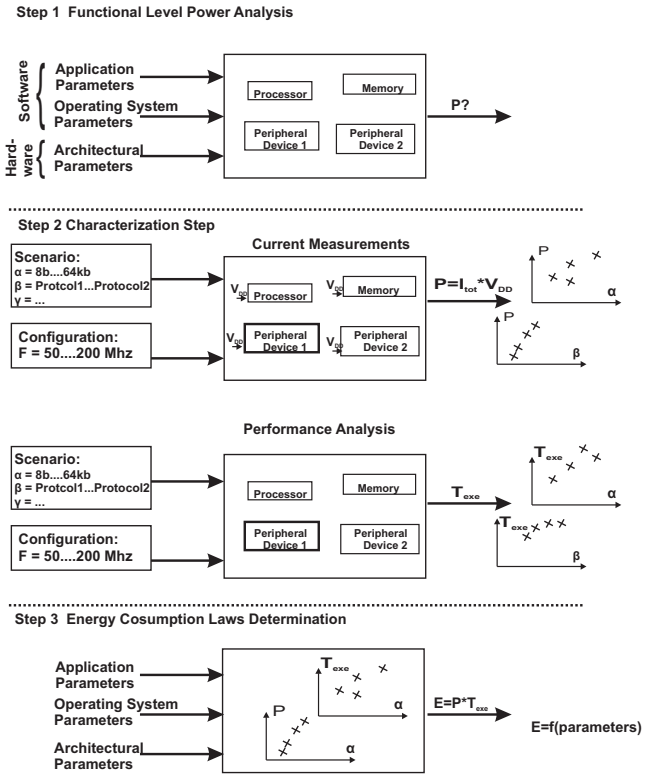


Figure 2: Model Definition Steps

3.3 Methodology

Power and energy estimation methodology is divided into two parts, model definition and the estimation process.

3.3.1 *Model Definition*

We perform model building for each hardware component related to the execution of applications using an operating system on an embedded architecture. Model Definition is composed of three steps (cf. Fig.2).

The first step is a functional-level power analysis of the target architecture components. This analysis provides us with a set of parameters that could impact the power and energy consumption. At the characterization step, we determine, by using physical measures, consumption laws describing the average supply current's evolution relative to these parameters. We also carry out performance analysis of the OS based application execution and discern relevant parameters that influence the performance execution. Then, we determine performance laws describing the execution time evolution relative to these parameters. At the final step, we merge power consumption and performance models to compute energy consumption laws.

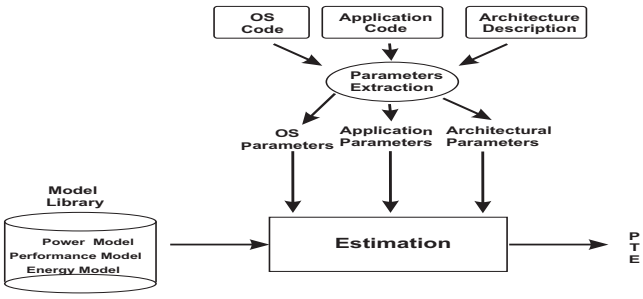


Figure 3: Estimation Process

3.3.2 *Estimation Process*

The model definition results in a model of the embedded target component, such as processor, memory and peripheral devices. Software parameters are extracted from the application code. Hardware parameters are extracted from hardware description files. The model uses the selected parameter values as inputs and accounts for all power and energy consumption sources. Software parameters are extracted from the application code. The estimation process is explained in Fig.3.

4. SYSTEM OVERVIEW

In this section we describe the target system for our experimental measurements. The Hardware is a XUP Virtex-II pro development board, while the OS is Montavista Linux, a real time embedded operating system from Montavista Software that we ported to the target platform.

4.1 The Hardware Platform

Our power and energy consumption modeling methodology has been conducted on an experimental framework composed of an XUP Virtex-II pro development board, a 256MB SDRAM memory, a 512 MB compact Flash and an Ethernet interface. The Virtex-II pro implements PowerPC 405, which is a simplified version of IBM PowerPC processor. Two PowerPC cores, memory management unit and data and instruction cache are integrated in the same chip.
Many I/O controllers are integrated in the FPGA, such as audio, Ethernet, UART, compact flash and SDRAM.

4.2 The Software Platform

The operating system that we analyze in this study is the Montavista embedded Linux 3.1. It is based on the 2.4.6 linux kernel, is preemptive and integrates a fixed priority low overhead real time scheduler. Montavista is considered a real time operating system because it introduces High resolution Timers, offering developers increased control over real-time applications.
To analyze the dynamic behavior and to retrieve performance information, we have extended Montavista kernel with the Linux Trace Toolkit. LTT provides a modular way of recording and analyzing all significant OS events related to any subset of running processes. The LTT time and memory overhead is minimal ($< 2.5\%$ when observing core kernel events) [10].

5. ETHERNET COMMUNICATIONS ENERGY AND POWER CONSUMPTION MODEL

We select a standard peripheral device, the Ethernet interface, as a representative example implemented in most of embedded systems. As a first step, we identified the key parameters that can influence the power and energy consumption of Ethernet communications. Then we conducted physical power measures on the XUP pro development board, and took execution time values from the traces obtained by LTT. Measures were realized when running different testbenches that contain RTOS routines stimulating the Ethernet interface. Once we obtained all measures, we built the power and energy consumption model of the Ethernet interface.

5.1 Analysis of Relevant Parameters

Our study is focused on the effect of the operating system on power and energy consumption of embedded system components. In the case of the Ethernet interface component, we identified hardware and software parameters influencing energy consumption.

5.1.1 Hardware Parameters

When sending data from the applicative tasks to the Ethernet controller, the OS encapsulates data to form packets conforming to the TCP or the UDP protocol. As shown in Fig.4, the encapsulation process is performed by the kernel services and the device driver. The OS sends all computing tasks to the processor. These tasks require clock cycles, so processor frequency should be a parameter of models. The Ethernet Controller is connected with the processor and the main memory through the system bus, so the frequency of the bus is also a parameter of the model.
The effects of processor cache misses on the power and energy consumption also should be considered. In our case, two configurations can be used. In the first configuration, when there are cache misses, the processor fetches data from the FPGA block RAM, and for the second configuration, from the external SDRAM. In conclusion, hardware parameters for our models are processor frequency, bus frequency and primary memory type (BRAM or.

5.1.2 Software Parameters

Software parameters are related to the applicative tasks and the operating system services. At the application level, the significant parameter is the IP packets data size. The maximum data size corresponds to the maximum IP packet size (64 kb) minus the packet header size.
At the operating system level, we tested two different transmission protocols: the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP) [5]. TCP is a reliable stream delivery service. This means that the information is guaranteed to arrive, or an error will be returned. UDP is connectionless and unreliable which means that it does not establish a virtual circuit like TCP, nor does it demand acknowledgement. UDP prioritizes speed over reliability.

5.2 Power and Energy Characterization

We performed power consumption characterization for three components of the XUP board as shown in Fig.5.
The first component is the core processor which is powered by a 1.5 V power supply. The second corresponds to the FPGA I/O which are powered by a 2.5 V power supply such as the MAC Ethernet controller. The third component is the physical Ethernet controller which is powered by a 3.3 V power supply.
We used test programs that only stimulate the OS networking services. Therefore, only the processor, RAM and Ethernet Interface are solicited. We measured the average supply current of the processor core, the MAC Ethernet Controller and the physical Ethernet controller in relation to the variation of each software and hardware parameter. Curves fitting this data yields the consumption laws.
Using the program execution time T_{exe} given by LTT traces, we also compute average energy ($E = I_{total} * V_{DD} * T_{exe}$).

5.3 Models

5.3.1 Power consumption laws

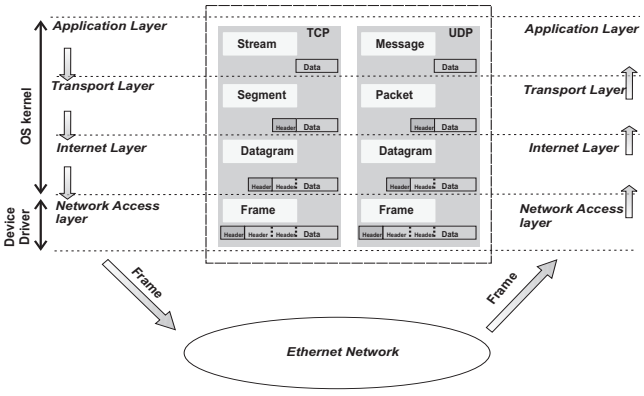


Figure 4: Network communications

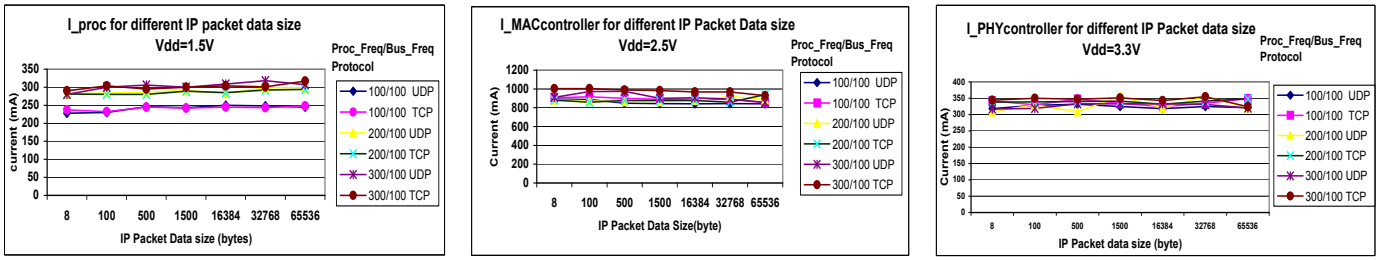


Figure 6: Current variation according to IP packets data size

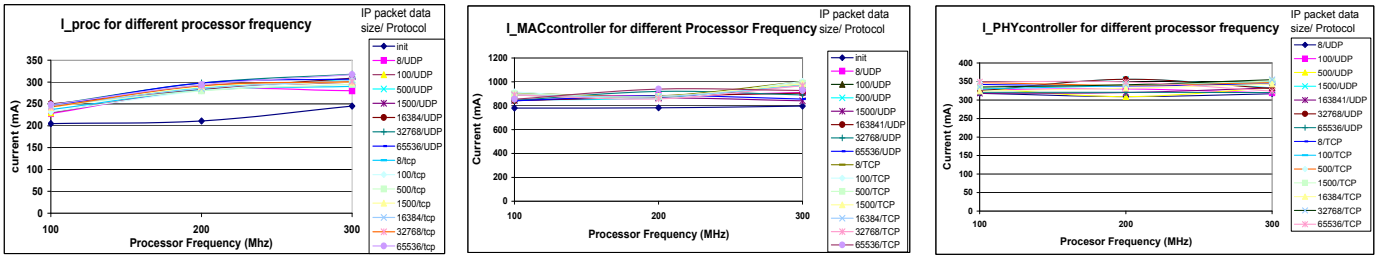


Figure 7: Current variation according to processor frequency

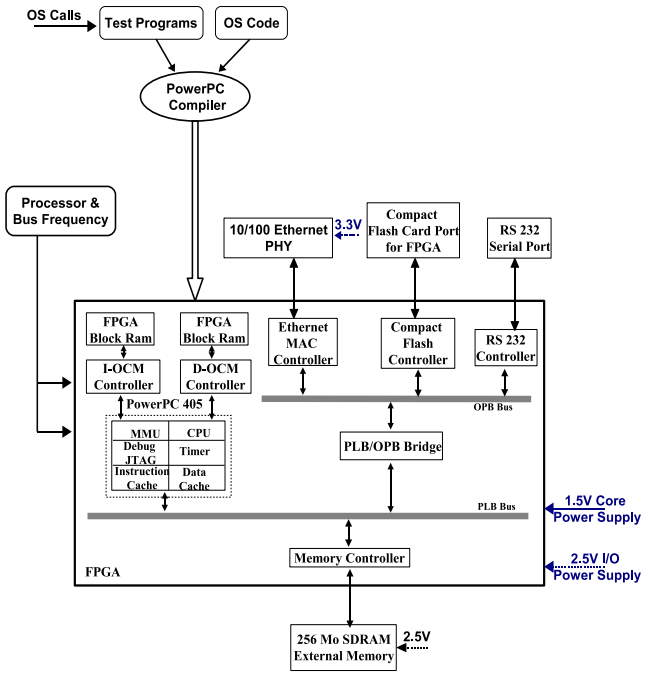


Figure 5: Embedded System Components Characterized

Table 3: Power laws

Processor	$P_{proc}(mW) = 0.45 * F_{proc}(MHz) + 315$
MAC controller	$P_{MAC}(mW) = 0.65 * F_{proc}(MHz) + 2100$
PHY controller	$P_{PHY}(mW) = 1096.22$

To determine the power consumption variation, we realized current measures when sending data from the XUP board to a machine on the local network. We tuned the IP packet data size from 8b to 64kb and used TCP and UDP protocols. We fixed the bus frequency to the maximum (100MHz), and we took measures for three processor frequencies (100, 200, 300 MHz). Fig.6 shows the evolution of the current when varying the IP packet data size. From the curves, we can notice that for each processor frequency and each transmission protocol, the current is approximately constant even if the IP packet data size is variable. In this case, the most significant parameter of the processor power consumption is its frequency. In fact, Fig.7 shows, for each power supply, the evolution of the current for different processor frequencies. The curve *init* corresponds to the initial state when the RTOS is running and no data is sent on the Ethernet. Power dissipation can be considered linear with the processor frequency for the processor and MAC controller component. For the PHY controller, power dissipation can be considered constant. The power consumption mathematic laws of the processor, MAC controller and PHY controller are represented by the equations in Table 3. For the three laws, the average error between power consumption measured and values estimated by the models is 3,5%, the maximum error is 9%.

5.3.2 *Energy consumption laws*

Following the methodology defined in section 3.3, we made performance analysis of the whole system. Then, we calcu-

Table 4: Mac Controller Energy Consumption (μJ) per byte transmitted

		Data Cache to SDRAM						Data Cache to BRAM					
Processor Freq(Mhz)		100 SDRAM		200 SDRAM		300 SDRAM		100 BRAM		200 BRAM		300 BRAM	
Protocol		UDP	TCP	UDP	TCP	UDP	TCP	UDP	TCP	UDP	TCP	UDP	TCP
IP Packet data size (byte)	8	57,915	40,532	37,252	20,672	41,280	16,612	57,938	35,615	40,781	19,641	27,713	18,000
	100	4,906	4,443	2,968	2,298	2,749	2,811	5,229	4,066	3,137	2,520	2,496	2,167
	500	1,146	2,349	0,772	1,096	0,789	1,047	1,270	1,360	0,864	0,900	0,641	0,806
	1500	0,651	0,932	0,441	0,648	0,404	0,618	0,707	0,891	0,481	0,643	0,415	0,598
	16384	0,386	0,594	0,280	0,423	0,264	0,432	0,400	0,576	0,288	0,423	0,234	0,410
	32768	0,372	0,583	0,295	0,404	0,250	0,409	0,379	0,577	0,274	0,410	0,233	0,402
	65536	0,365	0,543	0,278	0,434	0,237	0,388	0,438	0,531	0,280	0,383	0,237	0,394

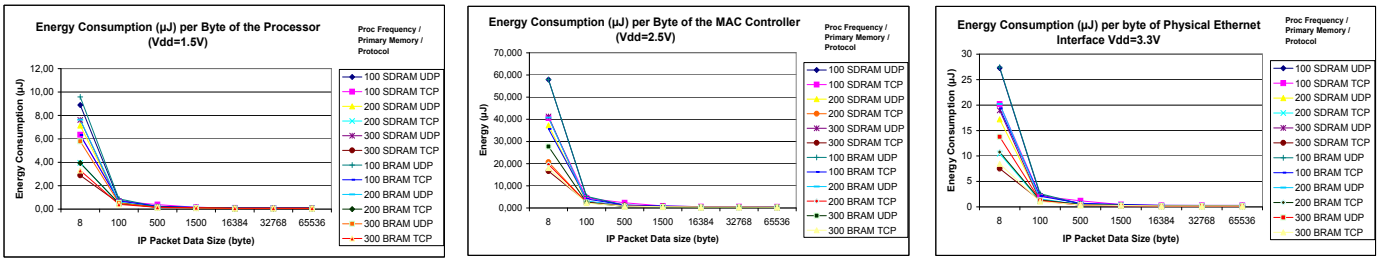


Figure 8: Energy variation per byte transmitted according to IP packet data size

late energy dissipation values in relation to the variation of all the model parameters, i.e., bus frequency, processor frequency, primary memory, IP packet data size and transmission protocol. We obtained energy values for the processor, MAC controller and PHY controller. Energy consumption per byte transmitted for the MAC controller are exposed in Table 4, which indicates a complex relation among the parameters.

The evolution of energy consumption per byte according to IP packet data size is shown in Fig.8.

We infer from the analysis of Fig.8 that using BRAM or SDRAM as a principal memory does not influence energy consumption. In the other hand, processor frequency, transmission protocol and IP packet data size are significant parameters. We studied the influence of the two software parameters, and then developed the energy model laws. For each power supply, processor frequency and transmission protocol, we obtained a law functions of IP packet data size parameter. Then the energy model is composed of several laws that depend on hardware and software parameters.

Fig.9 shows the evolution of the MAC controller energy consumption for 200MHz processor frequency and functions of IP Packet data size. We have divided the graphic in two parts: the first part corresponds to IP packet data size varying from 8b to 1500b, the second corresponds to IP packet data size from 1500b to 64kb. We can see that UDP protocol consumes less energy than TCP except for small packets (<=100byte). From this experimental observation, which is applicable to all the curves of Fig.8, we remark that contrary to received ideas, TCP generates less energy overhead than UDP for small packets data size .

In Table 5, we give energy laws related to the MAC and physical controllers. For each transmission protocol and each processor frequency, there are two laws. The first is for IP packet data size less than 1500 byte, the second is for IP packet data size greater than 1500 bytes. Since the maximum transmission unit (MTU) of the Ethernet network is 1500 byte, the Internet layer fragments IP packets larger than MTU. On the other hand, there is more encapsulation and no fragmentation for IP packets smaller than MTU. We can see from the table 5, that encapsulation yields more energy dissipation than fragmentation.

The model we propose has some fitting error with respect to the measured energy values it is based on. We use the following average error metric: $\frac{1}{n}\sum_{i=1}^{n}\frac{|\tilde{E}_i - E_i|}{E_i}$ where $\tilde{E}_i$'s are energy values given by the model and E_i's are energy values based on power and performance measures. We observe that average error for energy is larger than power estimation error, this is due to performance estimates. Actually, performance variations are caused by the OS background activities. For example, for a 500 byte IP packet data size, TCP protocol and 200 Mhz processor frequency, time necessary to send the packet through Ethernet varies from 23μs to 24632μs.

6. CONCLUSIONS

The first part of this work demonstrates why is it important to model energy dissipation of RTOSs, especially peripheral devices management. Then, we presented a methodology for modeling power and energy consumption of embedded systems running operating systems. The methodology is composed of two parts: model definition and estimation process. We focused our study on operating system peripheral device management and we took, as a case study, a standard peripheral, namely the Ethernet Interface. Such models are necessary to take power/energy efficient decisions at first design phases. The approach is generic and can be applied to any peripheral (e.g. flash memory, audio, video, etc.)

This work is a part of a global project aiming a system level design framework in which our models will be implemented.

7. REFERENCES

[1] A. Acquaviva, L. Benini, and B. Riccó. Energy characterization of embedded real-time operating systems. In *Proceedings of the Workshop on Compilers*

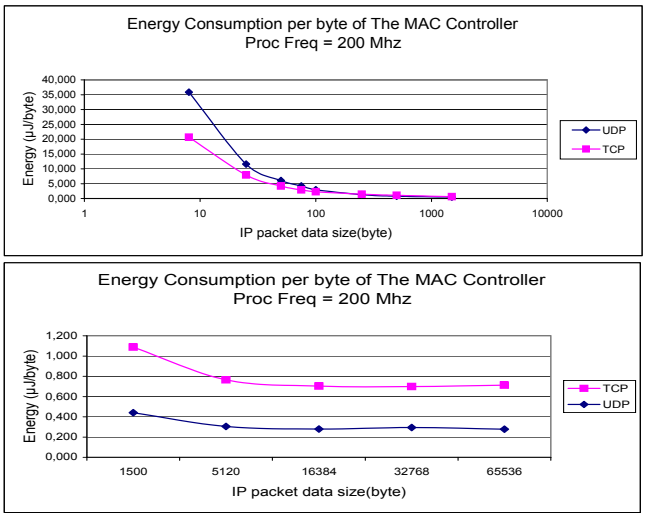


Figure 9: Energy variation per byte transmitted according to IP packet data size, $F_{proc} = 200Mhz$

Table 5: Energy model

Proc Freq	Model $E(\mu J/byte) = a * Packet_{size}^{b}$	Error
MAC Controller(2.5V)		
100MHz	$E_{UDP} = 304,87 * P_{size}{}^{-0,88}, if P_{size} < 1500b$ $E_{UDP} = 1,47 * P_{size}{}^{-0,13}, if P_{size} \geq 1500b$	6,28%
	$E_{TCP} = 126,29 * P_{size}{}^{-0,68}, if P_{size} < 1500b$ $E_{TCP} = 1,80 * P_{size}{}^{-0,107}, if P_{size} \geq 1500b$	10,16%
200MHz	$E_{UDP} = 181,26 * P_{size}{}^{-0,86}, if P_{size} < 1500b$ $E_{UDP} = 0,85 * P_{size}{}^{-0,11}, if P_{size} \geq 1500b$	7,14%
	$E_{TCP} = 60,73 * P_{size}{}^{-0,65}, if P_{size} < 1500b$ $E_{TCP} = 1,17 * P_{size}{}^{-0,1}, if P_{size} \geq 1500b$	8,26%
300MHz	$E_{UDP} = 144,11 * P_{size}{}^{-0,84}, if P_{size} < 1500b$ $E_{UDP} = 0,79 * P_{size}{}^{-0,11}, if P_{size} \geq 1500b$	8,59%
	$E_{TCP} = 63,44 * P_{size}{}^{-0,65}, if P_{size} < 1500b$ $E_{TCP} = 1,09 * P_{size}{}^{-0,09}, if P_{size} \geq 1500b$	8,68%
PHY Controller(3.3V)		
100MHz	$E_{UDP} = 140,45 * P_{size}{}^{-0,86}, if P_{size} \leq 1500b$ $E_{UDP} = 0,72 * P_{size}{}^{-0,13}, if P_{size} \geq 1500b$	7,4%
	$E_{TCP} = 61,14 * P_{size}{}^{-0,68}, if P_{size} < 1500b$ $E_{TCP} = 0,89 * P_{size}{}^{-0,106}, if P_{size} \geq 1500b$	9,92%
200MHz	$E_{UDP} = 88,005 * P_{size}{}^{-0,85}, if P_{size} \leq 1500b$ $E_{UDP} = 0,41 * P_{size}{}^{-0,108}, if P_{size} \geq 1500b$	7,53%
	$E_{TCP} = 31,9 * P_{size}{}^{-0,657}, if P_{size} < 1500b$ $E_{TCP} = 0,71 * P_{size}{}^{-0,117}, if P_{size} \geq 1500b$	8,91%
300MHz	$E_{UDP} = 88 * P_{size}{}^{-0,856}, if P_{size} \leq 1500b$ $E_{UDP} = 0,454 * P_{size}{}^{-0,126}, if P_{size} \geq 1500b$	14,5%
	$E_{TCP} = 31,9 * P_{size}{}^{-0,657}, if P_{size} < 1500b$ $E_{TCP} = 0,58 * P_{size}{}^{-0,105}, if P_{size} \geq 1500b$	10,15%

and Operating Systems for Low Power (COLP'01), sep 2001.

[2] A. C. Amit Sinha. Jouletrack - a web based tool for software energy profiling. In *Design Automation Conference*, pages 220–225, 2001.

[3] K. Baynes, C. Collins, E. Fiterman, B. Ganesh, P. Kohout, C. Smit, T. Zhang, and B. Jacob. The performance and energy consumption of embedded real-time operating systems. *IEEE Transactions on Computers*, 52(11):1454–1469, Nov. 2003.

[4] R. P. Dick, G. Lakshminarayana, A. Raghunathan, and N. K. Jha. Power analysis of embedded operating systems. In *Proceedings of the 37th Conference on Design Automation (DAC-00)*, pages 312–315, NY, jun 2000. ACM/IEEE.

[5] C. Hunt. *TCP/ IP Network Administration (2nd ed)*. O'Reilly, 1998.

[6] E. Senn, J. Laurent, E. Juin, and J. Diguet. Refining power consumption estimations in the component based aadl design flow. In *FDL'08, ECSI Forum on specification and Design Languages*, 2008.

[7] T. K. Tan, A. Raghunathan, and N. K. Jha. Embedded operating system energy analysis and macro-modeling. In *Proceedings of the 2002 IEEE International Conference on Computer Design (ICCD'02)*, 2002.

[8] A. Vahdat, A. Lebeck, and C. Ellis. Every joule is precious: A case for revisiting operating system design for energy efficiency. In *Proceedings of the Ninth ACM SIGOPS European Workshop 2000*, Sept. 2000.

[9] A. Weissel. *OS Services for Task Specific Power Management*. PhD thesis, Erlangen University, 2006.

[10] K. Yaghmour and M. R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In USENIX, editor, *2000 USENIX Annual Technical Conference: San Diego, CA, USA, June 18–23, 2000*. USENIX.

Evaluation of a Minimal POSIX Tracing Service Profile for Real Time Embedded Systems *

P. Parra M. Knoblauch C. Rodríguez
O. Rodríguez S. Sánchez
Department of Computer Engineering
University of Alcala
Alcala de Henares, Spain

A. Viana
ESA/ESTEC - TEC/EDD
Noordwijk, The Netherlands

Abstract

In this paper we propose the definition of a minimal tracing service profile for the POSIX trace standard, suitable for embedded platforms, that comprises the most important primitives of the whole POSIX standard and covers two more key points. The first point is introduced to overcome the limitation of the current POSIX trace system that does not address the need for eventually peeking at the trace in LIFO mode, which could facilitate the remote maintenance of embedded systems that lack, among other resources, of a file system. The second point is the definition of a mechanism for managing different levels of trace information that are not included in the standard. The proposal is completed with the specification of a set of implementation requirements to ensure that the tracing system, with a minimal and deterministic intrusion, is capable of covering both the continuous tracing required during testing and validation phases and also the remote diagnosis during system lifetime. Finally, we present the results obtained with the implementation of this proposal over ERCOS-RT, a minimal real-time operating system developed for space systems and applications.

1 Introduction

In the field of real-time systems, tracing support is generally considered to be a useful tool during testing processes but it is usually removed when the system is deployed. It is worth considering, however, that this support should be an integral part of the whole system, maintained during the operational phase. The benefits of this approach are twofold. First, it avoids the fact, paradoxically accepted by many developers, that the deployed system is different from the one that was validated.

Second, the evolution of the system during all its life time can be retrieved from the trace information, aiding in the diagnosis of any possible fault and opening the possibility of taking action remotely. Moreover, it is striking that, although the trace information of a generic system can be generated by different service levels (e.g. operating system level, middleware, applications, etc.), none of the tracing mechanisms allow that kind of classification, which would ease, during recovery and analysis of the system traces, routing every event to a suitable tool according to its level of abstraction.

The development of these kinds of applications needs some tracing mechanism in order to ensure correct system behaviour. System monitoring provides a lot of information that is very handy when one is validating a system's functionality and performance. It is also useful for certifying real-time system constraints. On embedded systems capable of communicating with an external outpost or control centre, like those integrated in a satellite, trace information can be viewed as an integral part of the system. This information is a key source of knowledge, not only during validation and testing, but also across the life cycle of the system.

This approach allows the diagnosis of any possible fault and opens the possibility of taking action remotely to ensure the product maintenance. In recent research works tracing has been proposed as a mechanism to develop dynamic systems where adaptation is the key for success. This adaptation is performed by a reflexion level that requires full knowledge of the system state [12]. It is important to emphasize that it is unrealistic to assume that all contingencies can be foreseen during validation phases. Doing it only leads to a false sense of security [13, 14]. As Edsger Dijkstra noted, "program testing can be quite effective for showing the presence of bugs, but is hopelessly inadequate for showing their absence" [7].

*This work has been supported by *Comisión Interministerial de Ciencia y Tecnología (CICYT)* of Spain, grant ESP2005-07290-C02-02

It is feasible to perform validation process analyzing trace information obtained by non-intrusive techniques (e.g. using on-chip emulators or logical analyzers). Though, in very few cases is it acceptable in terms of cost to integrate these kinds of solutions as part of the final deployed system. Therefore, in these cases it is necessary to have a tracing support provided by the real-time operating system which allows two tracing mechanisms: one capable of tracing in continuous mode, used in system validation; and other, used sporadically, which allows remote monitoring at certain instants of the lifetime of the system.

Moreover, provided that during validation process a software trace support is used, it is essential to maintain it on the final system to ensure that the validated system is identical to the one which is deployed.

Paradoxically, this fact is not always accepted by developers, which is a misconception. Even though sometimes it only slightly alters the response time of the system, it can lead to a more serious error when the platform incorporates a cache mechanism. In this case, a minor code change could modify the memory layout and thus, the sets of code fragments that share every cache line. These kinds of problems are referred in [6].

Another relevant aspect in the validation process is the ability of the tracing support to aid in the analysis of the traced information. In embedded systems, this information is often generated in different service levels. During the recovery and analysis phases, it is desirable to redirect the traced events to the corresponding tool depending on the level of abstraction of the service.

The current POSIX trace support, defined by the standard 1003.1q, is designed mainly to provide support for the system validation processes, but it does not address the need for eventually peeking at the trace in LIFO mode which would facilitate the remote maintenance of embedded systems. An example of this use could be to recover the latest events ocurred in the system at some crucial instants, such as, after an unexpected system reset. Furthermore, the standard does not define any mechanism for managing different levels of trace information.

Here we propose the definition of a minimal tracing service profile for the POSIX trace standard, suitable for embedded platforms. It comprises the most important primitives of the whole POSIX standard. Apart from these original primitives, we propose an extension to the standard to cover these two key points: recovery of the trace in LIFO mode and the definition of different levels of trace information. The definition of such a tracing support is an open topic [11]. The work we present here is intended to complement the current proposals in order to solve the problems raised by the tracing support in the development of these kinds of systems.

This work is completed with the specification of a set of requirements that any implementation of the profile must meet to ensure that, maintaining a minimal and deterministic intrusion, it is possible to cover both the continuous mode tracing (required during validation phases) and the remote diagnosis after the deployment and during the lifetime of the system. Finally, we present the results obtained from the implementation of this proposal over ERCOS-RT, a minimal real-time operating system developed for space systems and applications.

2 POSIX trace standard overview

The POSIX trace standard has been developed to provide tracing facilities to the systems. It defines two main data types, called *events* and *trace streams*. The former includes events, occurred on the system, that must be traced. The latter is the buffer stream where the information of each event will be stored in order to be eventually analysed.

2.1 Trace events

When a specific application or program needs to be traced, all the required traceable events are defined. In POSIX trace terminology, the points where the information must be generated are called *trace points*, and the information itself is called *trace events*. Each event belongs to a certain *event type* and is also associated with an *event name*. When an instrumented application wants to register a new event, it must invoke the `posix_trace_eventid_open` routine, which returns the event identifier. If the event had been previously defined, the function call returns its former event identifier. The event trace mechanism is performed by calling the `posix_trace_event` routine. The standard specifies the information that must be saved with every trace, namely:

- The trace event type identifier
- A time-stamp
- The process identifier of the traced process
- The thread identifier of the traced process, if the operating system supports threads
- The program address where the trace is being performed
- Any extra data associated with the event and previously defined by the user
- The extra data size

2.2 Stream buffers

When any system application traces an event, its information is stored in the stream buffer. The POSIX trace standard specifies that streams must be created by processes. The relationship between streams and processes is many-to-many, i.e. all events associated with a process are traced in all stream buffers belonging to that process. Thus, it is possible to trace events from a single process into many streams. The POSIX standard supports also event filtering. This means that it is possible to filter some specified events in order not to store them in the stream buffer. By doing this, events corresponding to one thread can be associated with a single stream. It also allows tracing the events of various processes into one single stream. This situation occurs when a process creates some buffer streams before creating the rest of the processes by using the corresponding `fork` system calls. In the case of one single process compounded by many threads, all events of the different threads are traced into all the streams belonging to the process.

The standard defines two types of streams: active streams and pre-recorded streams. An active stream is created to trace events during system execution. It can also be associated with a log file in order to store the information on a persistent object when a flush operation is performed. A pre-recorded stream is designed to retrieve events that have been previously recorded in a log file. They are frequently used to carry out off-line analysis of the tracing activity.

2.3 Tracing roles

The POSIX trace standard defines three types of roles called *trace processes*: the trace controller process, the traced process and the analyser process.

Controller process. The controller process is in charge of the stream buffer creation and, in most cases, of the trace system start up. It must carry out the following operations: (1) creating the trace stream with its attributes; (2) starting and stopping the trace system; (3) filtering the events that are being traced in the corresponding streams; and (4) shutting down the stream.

Traced process. The traced process is the one that is being traced. The standard defines only two operations that must be carried out by this kind of process: (1) registering a new user event by calling the `posix_trace_eventid_open` routine; and (2) tracing the appropriate event by calling the `posix_trace_event` routine.

Analyzer process. The analyser process is in charge of retrieving the traced events from the stream buffer in order to analyse the system behaviour. This process can perform on-line or off-line analysis, depending on the type of the stream.

It is not mandatory that every role is performed by a different process. In the next sections we will see that, in our implementation, controller and the traced processes are the same process. The analyser, on the contrary, is to be implemented separately.

2.4 Implementation options

The standard defines different layers in the implementation which can be fulfilled or not, depending on the particular system trace functionality. These layers are:

Trace layer. This layer is mandatory. It includes the tracing mechanism in charge of tracing the different events and their storage in the streams.

Trace log layer. This layer comprises facilities to perform a system trace by using logs, allowing off-line analysis.

Trace inheritance layer. This layer allows storing trace information of several processes into a unique stream. This option is activated in the moment when a process creates a stream and, after that, forks several processes which will inherit the stream. By this way it is, thus, possible to associate one stream to several processes.

Trace filtering layer. This layer allows filtering events to prevent their tracing into a specific stream.

3 Minimal POSIX tracing services

Real-time embedded systems have certain characteristics that complicate the development of software applications for these environments. As a general basis, these systems have strong limitations on the available development resources: they normally lack of storage devices, memory protection or virtual memory mechanisms, etc. Focusing on these systems, the POSIX minimal real-time system profile was proposed. This profile is a set of POSIX API primitives intended for small real-time embedded systems that eliminates most of the services that are only meaningful on the general purpose systems.

Currently, none of the POSIX tracing services [5] is included on the Minimal Real-Time Systems Profile. Though recently there have been proposals in this regard [11], an identification of the services to be provided is still under discussion.

Here we propose a minimal POSIX tracing service profile that covers the basic tracing services of the original POSIX standard and also adds, as an extension, a pair of new services intended to allow the proposal to meet the needs of real-time embedded systems.

The first of these new services is designed to facilitate the remote tracking and control at certain instants of time during the lifetime of the system. We call this kind of service sporadic tracing, to differentiate it from the continuous tracing, used solely during testing processes.

The main difference between them is that continuous tracing is aimed at analyzing all the tracing information generated by the system on every test case and thus, it is necessary to recover on FIFO mode, all the events sent to the trace stream buffer. In sporadic tracing, instead, the goal is reading on LIFO mode, at a certain given time, the contents of the stream buffer. This buffer snapshot can then be send to a remote control centre, which will establish a diagnosis of the current situation.

The second proposal of amendment to the standard is related to the analyzing process of the tracing information. On real-time systems, this tracing information is generated in different service levels. A simplified approach to this reality would force us to distinguish at least two levels: operating system and application level. The POSIX trace standard imposes a single trace stream per process so that it becomes too expensive to classify at recovery time the events belonging to the different levels and thus, it is not feasible to redirect every event towards the right tool depending on the level of abstraction. The new proposed primitive solves this drawback introducing the possibility of specifying the service level of every traced event.

The following subsections explain the different parts of the proposal: the set of POSIX primitives belonging the standard that are added to the minimal profile, the new set of primitives for sporadic and continuous tracing and the ones that allow multilevel tracing. Furthermore, a last subsection shows the requirements that an implementation of this proposal must follow to guarantee all services, minimizing the overhead.

3.1 POSIX trace standard primitives

The POSIX trace system suffers from the same excess of weight as the whole standard. The vast majority of the interfaces it provides are neither not suitable nor necessary for embedded platforms.

From the original set of primitives, we propose the following subset to be a part of the minimal POSIX tracing services:

- `posix_trace_eventid_open`. Used by the traced thread to obtain an identifier for a certain event that is to be traced.
- `posix_trace_event`. Traced thread calls this primitive to trace a certain event.
- `posix_trace_create`. Instantiates and initializes the trace stream. Used by the controller thread.
- `posix_trace_shutdown`. Disables the trace stream. Used by the controller thread.

This subset does not include any primitive for retrieving events from the stream buffer. This is due to the fact that the analyser process will run on the setup host, not in the target.

However, a low priority thread (the idle thread) will retrieve the events directly from the stream buffer. The events will then be sended to the analyzer process[1] running in the control centre.

3.2 Sporadic tracing

We propose a modification to the original standard to allow sporadic tracing. With this method, the goal is reading on LIFO mode, at a certain given time, the contents of the stream buffer. This buffer snapshot can then be send to a remote control centre, which will establish a diagnosis of the current situation.

The primitive that supports the sporadic tracing is the following:

- `posix_trace_get_stream_buffer`. It can be used to retrieve a certain number of events commencing from the last one stored in the stream buffer.

To be able to implement this mechanism, some considerations have to be addressed.

3.3 Multilevel tracing

We propose a modification to the original standard to allow the possibility of assigning a level identifier to the different traced events. The level is used to build the event identifier and it can be used to select the events belonging to a certain level of abstraction. When recovering the tracing information, the analyzer thread can demultiplex the information and route it to different tools or applications that will perform the analysis of the data depending on the level to which they belong.

To support multilevel tracing, a new primitive is defined:

- `posix_trace_eventid_open_with_level`. It has a similar semantic to the original `posix_trace_eventid_open`, with the difference that it can associate to the requested event, an additional identifier to determine the level to which it belongs.

This original primitive is maintained only for compatibility but it does not make sense to use in a system. If any event is requested with the original primitive, it will be assigned to level zero.

[1] By means of any kind of communication link, usually a serial line.

3.4 Implementation considerations

To make the controller thread as less intrusive as possible it must have the lowest priority. This implies that it limits its execution to the amount of time the system remains idle. The stream buffer implementation must allow the storage of events produced by the traced threads and their recovery by the controller thread in mutual exclusion. It must be implemented as a circular buffer as shown in Figure 1. Keeping low the overhead introduced by the controller thread access to the stream buffer will lead to a low overhead in the trace generating primitives. This is a key point in the implementation of the controller thread.

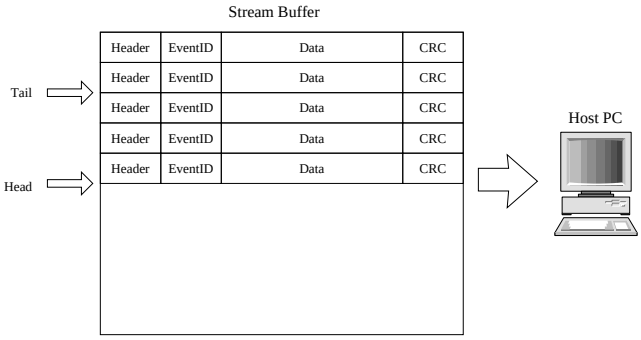


Figure 1. Stream buffer implementation.

In order to minimize the amount of overhead introduced by the mutual exclusion access, the stream buffer works with fixed-size fragments of event information.

The `posix_trace_event` primitive thus divides the traced event information in fixed-size fragments and adds some header to let the analyzer reconstruct it. The primitive, then, invokes an internal system call to store the fragments, one by one, in the stream buffer.

Fragmentation of the trace events allows the trace system call latency to be keept under a given threshold (directly proportional to the fragment size). In most Real-Time Operating Systems (RTOS), the context switch latency is related to the system call latency. The fragmentation scheme that we propouse avoids, in most cases, the context switch latency to have an indeterminated latency.

The size of the tracing buffer is defined statically. When the buffer is full and a new fragment is inserted, the oldest fragment is overwritten, and a flag on the header of the new fragment is set in order to signal the loss of fragments for the analyzer. The size of the buffer must be chosen according to two fundamental parameters: (1) the remaining physical memory available for the tracing mechanism and (2) the ratio between the arrival frequency of the different traced events and the output package transmission frequency. Since the controller thread is only executed when the system is idle, the amount of time available to retrieve a package from the stream buffer (and thus the output frequency) depends on the system load and can be calculated. The only way to avoid data loss during continuous tracing mode is keeping the output frequency higher than the tracing frequency and appropriately choosing the size of the buffer in order to handle the tracing bursts arisen between the executions of the controller thread.

The amount of information that the `posix_trace_get_stream_buffer` primitive can retreive is obviously limited by the size of the buffer. In addition, if new events occur during the execution of the primitive, these will not be retrieved in the current execution or, in case of overflow, some might appear out of order. Sequence numbers allow the events reordering and/or filtering in the external outpost.

4 Profile evaluation

The proposed Minimal Trace POSIX Service Profile has been evaluated over ERCOS-RT. ERCOS-RT is a real-time operating system developed over the standard platform of the European Space Agency (ESA) in space missions: the ERC32 processor [4, 10]. ERCOS-RT has been also ported to the next ERC32 evolution, called LEON, in its different versions, LEON, LEON2 and LEON3 [1]. ERCOS-RT was specifically designed to be compliant with the Minimal Real-Time System Profile (POSIX.13) for embedded systems. Therefore, modifying it in order to incorporate the proposed Trace Service Extension was perfectly affordable.

The ERCOS-RT design has included, also, the capability of adding the following RTOS events to the Trace Stream Buffer: (1) schedule entry and schedule exit; (2) semaphore wait and semaphore signal; (3) thread creation and thread termination; (4) thread block; and (5) interrupt trigger. This capability allows, amongst other things, to collect during analysis all the information required for measuring task execution times, compute the operating system overhead, detect deadlocks, etc. A tool, called Kiwi [2], can be also used to analyse in a graphical way the most relevant RTOS events retrieved in a session. Figure 2 shows an example of this analysis tool.

The implementation has passed a twofold evaluation. On one side we verified, on a real system, the correctness of the support for continuous, sporadic and multilevel tracing. On the other side, we measured the overload introduced by the new service in terms of execution time and code size.

The real-time system selected as test bench is an onboard satellite software. This system has been developed using the EDROOM tool [8]. This tool is inspired on ROOM [9] and UML2 [3] methodologies and it provides graphical design and automatically code generation under the component based paradigm. EDROOM incorporates a service library (`edroomsl`) to support the communication,

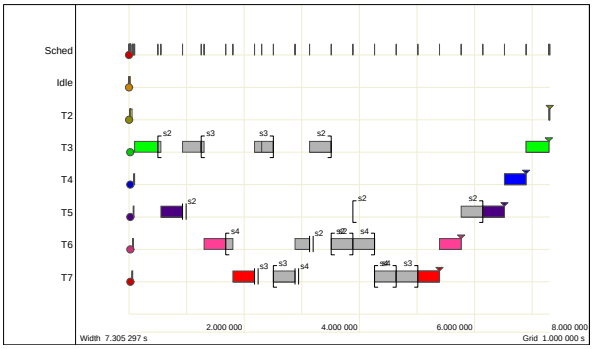


Figure 2. Kernel tracing using Kiwi.

scheduling and timing services that real-time requires. The `edroomsl` library provides also the capability to incorporate trace information in the generated system code. The trace information can be analysed using the same graphical notation that EDROOM uses during design. The trace analysis of this level is shown in Figure 3.

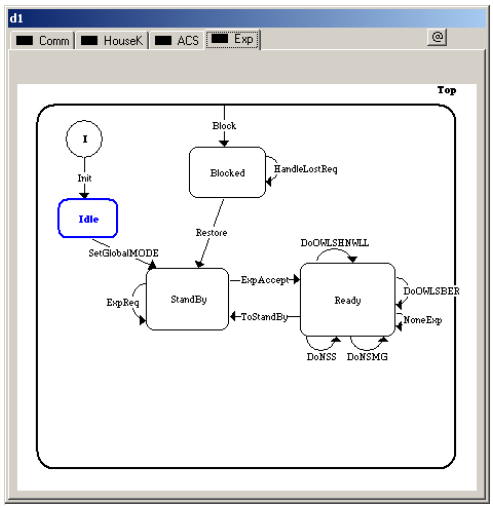


Figure 3. EDROOM level behaviour tracing.

For the on-board satellite application level we have also the facility of tracing the events associated with the on-board software behaviour. These events are sent to the Electrical Ground Support Equipment (EGSE) to monitor system aspects such as subsystem configuration, power consumptions, communication and experiment programs, etc. Figure 4 shows a snapshot of the EGSE application.

Figure 5 depicts how the continuous ERCOS-RT trace service works. System threads invoke the `posix_trace_event` primitive to trace the different events. This trace mechanism is depicted as "T" in the figure. The idle thread sends the stream buffer information, via serial line, through the interface depicted as "A" in the

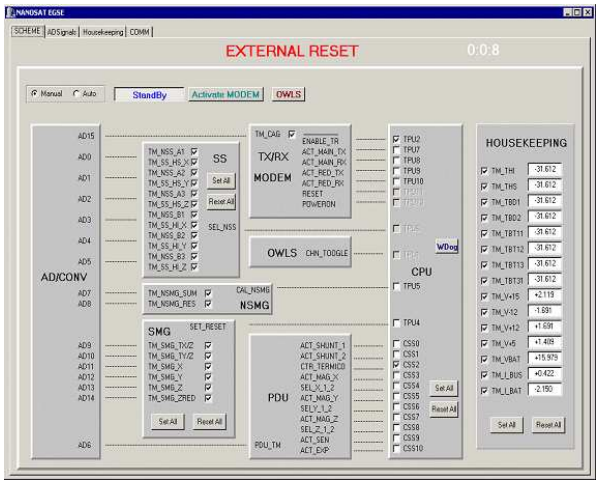


Figure 4. Application level tracing.

figure. The destiny of this information is a remote PC where the information can be analysed off-line or on-line.

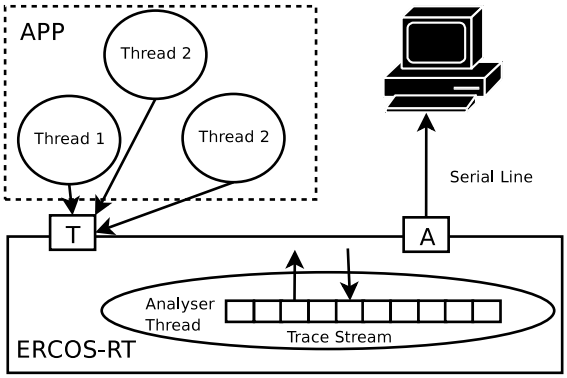


Figure 5. Tracing system.

The sporadic ERCOS-RT trace service, although, gets the trace information from the stream buffer information, in a LIFO way, only when the `posix_trace_get_stream_buffer` primitive is invoked. The information retrieved can be used to build a telemetry that will be sent to the mission centre. The telemetry can then be analysed in an off-line mode using the same tools as in continuous mode.

The multilevel tracing support allows, with both continuous and sporadic tracing, to treat the trace events of the unique stream buffer, trivially classifying them according to their corresponding level. Once classified, events are routed to the adequate tools for their analysis. In the on-board satellite software system employed, the three distinguished levels are (1) the level corresponding to the RTOS; (2) the level of EDROOM services; and (3) the application level. As it has been mentioned, a separate analysis tool ex-

ists for every one of these three levels. This capability has shown to be of great help as much for system validation, as for possible diagnosis, from the control centre, in case of eventualities of the flight system.

The code size overhead due to the new services is 3% of the 7500 lines of code of ERCOS-RT. The measurements have been taken over ERC32 working at 16MHz clock speed. The main result is that our RTOS event tracing support shows about 30 microseconds execution time overhead in every system call. Taking into account the improvements arisen in the validation and support processes, we believe that these outcomes can be considered acceptable.

5 Conclusions

The tracing support can be considered to be useful not only during testing processes but also during the lifetime of the system. In this paper, we have presented a minimal tracing service profile for the POSIX trace standard that includes the most important primitives. We have also added an extension to cover the need for eventually peeking at the trace in LIFO mode. This support could facilitate the remote maintenance of embedded systems. Along with this extension, we have also defined a mechanism for managing different levels of trace information in order to analyze the events with a suitable tool according to its level of abstraction. The proposal is completed with the specification of a set of implementation requirements to ensure that the tracing system is capable of covering both the continuous tracing and the remote diagnosis. These requirements assure that a minimal and deterministic intrusion is performed during the testing and validation phases, and also during the life time of the system.

The proposal has been implemented over an RTOS called ERCOS-RT and evaluated over an on-board satellite software system. Some results of performance tests have been also presented.

References

[1] Gaisler Research. http://www.gaisler.com.

[2] KIWI. http://rtportal.upv.es/apps/kiwi/.

[3] UML 2.0. www.u2-partners.org/uml2wg.htm.

[4] SAAB Ericcson Space. 32-bit microprocessor and computer development programme - Final. ESA Contractor Report, 1997.

[5] Standard for Information technology-Portable Operating Systems Interface (POSIX) - Part 1: System Application Program Interface (API) - Admendment 7: Tracing [C Language], 2000.

[6] G. Bernat, A. Colin, J. Esteves, G. Garcia, C. Moreno, N. Holsti, T. Vardanega, and M. Hernek. Considerations on the LEON cache effects on the timing analysis of on-board applications. In *Proceedings of DASIA 2008*, 2008.

[7] E. W. Dijkstra. *A discipline of programming*. Prentice-Hall, 1976.

[8] O. R. Polo, D. la Cruz J. M., G.-S. J.M., and E. S. ED-ROOM. automatic C++ code generator for real-time systems modelled with ROOM. In *NTCC2001 IFAC Conference*, November 2001.

[9] Selic, B., Gulleckson, G., and W. P.T. *Real-Time Object Oriented Modelling*. John Wiley and Sons, 1994.

[10] V. Stachetti, J. Gaisler, G. Goller, and C. L. Gargasson. 32-bit processing unit for embedded space flight applications. *IEEE Transactions*, 43:873–878, June 1996.

[11] A. Terrasa, A. Espinosa, and A. García-Fornes. Lightweight posix tracing. *Softw. Pract. Exper.*, 38(5):447–469, 2008.

[12] F. Valpereiro and L. M. Pinho. POSIX trace based behavioural reflection. *Lecture Notes in Computer Science*, 4006, 2006.

[13] M. M. Waldrop. A bad week for soviet space flight. *Science*, 241, 1989.

[14] M. M. Waldrop. Phobos at mars: A dramatic view and then failure. *Science*, 245, 1989.

An Integrated Model for Performance Management in a Distributed System

Scott A. Brandt, Carlos Maltzahn, Anna Povzner, Roberto Pineiro, Andrew Shewmaker, Tim Kaldewey
Computer Science Department
University of California, Santa Cruz
{scott,carlosm,apovzner,rpineiro,shewa,kalt}@cs.ucsc.edu

Abstract

Real-time systems are growing in size and complexity and must often manage multiple competing tasks in environments where CPU is not the only limited shared resource. Memory, network, and other devices may also be shared and system-wide performance guarantees may require the allocation and scheduling of many diverse resources. We present our on-going work on performance management in a representative distributed real-time system—a distributed storage system with performance requirements—and discuss our integrated model for managing diverse resources to provide end-to-end performance guarantees.

1 Introduction

Many computer systems ranging from small, embedded computers to large distributed systems have Quality of Service (QoS) requirements. Examples include flight control systems, defense systems, automotive systems, multimedia systems, transaction processing systems, virtual machines on shared hardware, and many others. Even traditional best-effort systems have hidden QoS requirements that are frequently expressed in terms of responsiveness.

Addressing the QoS requirements in all but the most trivial of systems may require the management of many resources: CPU, memory, network, cache, storage, power, and others. While a large amount of research has been conducted on how to provide QoS for individual resources, relatively few approaches—notably those of Lee [6] and Hawkins [3]—address overall system QoS or end-to-end QoS in distributed systems. We focus on end-to-end QoS in a distributed system using commodity hardware.

Interaction and dependencies between resources in complex/distributed systems require integrated solutions to provide overall performance guarantees. For example, compression algorithms may save network bandwidth and/or storage space, but at the cost of higher CPU utilization. Overall, the guarantees provided by a chain of resources can be no stronger than in the weakest link of that chain and

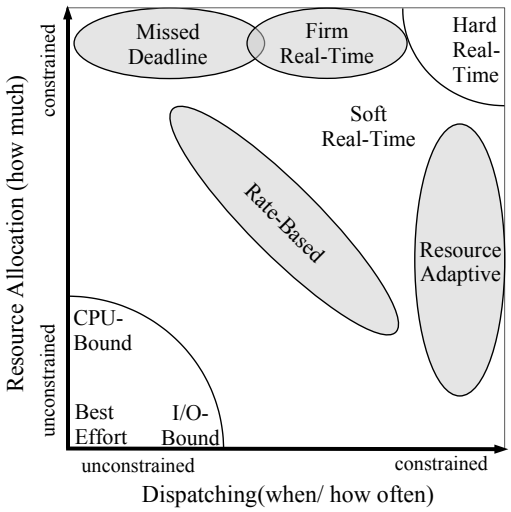


Figure 1: Classification of performance requirements in terms of Resource Allocation and Dispatching (RAD).

incompatible strategies for enforcing guarantees in different components may violate the overall QoS requirements, even if both components meet all of their individual requirements. For example, a network may provide a specified QoS by transferring a desired amount of data to a networked storage device, but in smoothing the network traffic to meet its QoS requirements, it may destroy the burstiness in the original workload that enables sequential accesses required for the disk to meet its I/O performance requirements.

Our goal is to develop a unified model for end-to-end QoS in complex and distributed systems that enables overall performance guarantees via the integrated management of all of the resources in the system. Our solution should support all types of processing guarantees ranging from best-effort to hard real-time. It should also allow the composition of guarantees on the individual resources for system-level performance guarantees independent of workloads, whether known or unknown *a priori*.

Our solution is based on the RAD scheduling model [1], originally developed in the context of CPU scheduling and subsequently extended to include other resources (*e.g.*, disk

I/O [11]). In the RAD model, resources are allocated in terms of *Resource Allocation* and *Dispatching* or, alternatively, *Rate* and *Period*. Resource allocation determines the amount of resources provided to a process over time, *e.g.*, percentage of CPU usage, network utilization, or disk head time. Dispatching determines the times at which the (reserved) resources must be delivered, effectively determining the granularity of the reservation. We have shown these two parameters to be sufficient to describe and support a wide range of scheduling policies ranging from best-effort to hard real-time [7], depicted conceptually in Figure 1.

In the RAD model, rate and period specify the desired performance, which must be enforced by the scheduler for the particular resource. The details of the scheduler depend upon the characteristics of the resource. We have developed schedulers for several resources, including CPU [1] and disk [11]. Our current work extends our disk scheduling research and adapts the RAD model to include network and I/O buffer cache management and begins to examine the interdependencies among those guarantees.

Our current focus is on managing the performance of distributed storage systems. Distributed storage shares many of the important properties of other distributed systems of interest to the embedded real-time community, such as sensor networks. In a distributed storage system, there are many independent I/O initiators operating on results in local memories and transferring data over a shared network to common targets. Where real-time data capture is important, sensor networks must also deal with local and distributed storage performance management (as well as power management).

Distributed storage performance management is challenging for a variety of reasons:

- End-to-end performance guarantees require the integrated management of at least four resources: the client buffer cache, the network, the storage server buffer cache, and the disk.
- Disk I/O is workload-dependent and individual requests are stateful and non-preemptible with response times that are only partially deterministic, varying by 3–4 orders of magnitude between best and worst-case performance.
- Independently-acting storage clients transfer data via a shared network. Rate enforcement ensures that the overall traffic is feasible, but traffic shaping must be used to avoid network congestion leading to packet loss [5, 10].
- Client and server I/O buffer caches must manage variance in the application I/O patterns and present the requests to each device so as to maximize its predictability and optimize its performance.

We discuss the RAD resource management model and explain its application to each of the system resources, providing results from our proof-of-concept implementations where available.

2 Architecture

Our target system is a distributed storage system consisting of clients accessing common storage devices over a shared network. The system is closed—we control all of the relevant resources in the system, including the clients' CPUs, buffer cache, and network access, and the servers' network access, buffer cache, and storage devices. No non-compliant traffic exists on the network and no non-compliant clients may access the storage. Although we control the resources, we do not control the applications, which may issue requests at any time.

Aside from the scale of our system, which may include up to many thousands of nodes and petabytes of storage, it is also representative of distributed embedded systems such as sensor networks or distributed satellite communications systems[1].

Our goal is to provide I/O performance guarantees to applications running on the client nodes. Application requirements have many forms: guaranteed throughput for a multimedia application; a guaranteed share of the raw disk performance for a virtual machine; and guaranteed latency for a transaction processing system. Regardless of the form of the requirements, our goal is a unified resource management system that ensures the performance of each workload through all of the resources, independent of other workloads.

Making and keeping I/O guarantees in a distributed storage system requires the integrated management of a number of resources, as shown in Figure 2, including the disk, the storage server buffer cache memory, the network, and the client buffer cache memory. The overall guarantees can be no stronger than can be provided in any individual component and the guarantees must be composable in order to provide an end-to-end guarantee.

We base our work on the RAD integrated scheduling model [1]. Originally developed for CPU scheduling, RAD separates scheduling into two distinct questions: *Resource Allocation*, or how much resources to allocate to each task, and *Dispatching*, or when to allocate the resources a task has been allocated. These two questions are independent and separately managing them allows a scheduler to simultaneously handle tasks with diverse real-time processing requirements ranging from best-effort to hard real-time [7].

[1] Although most satellite communication systems are monolithic custom (single) satellites, we are working with researchers at IBM Almaden on a DARPA-funded distributed communication satellite architecture that has many properties in common with our (ground-based) distributed storage system and which will use similar RAD-based resource management.

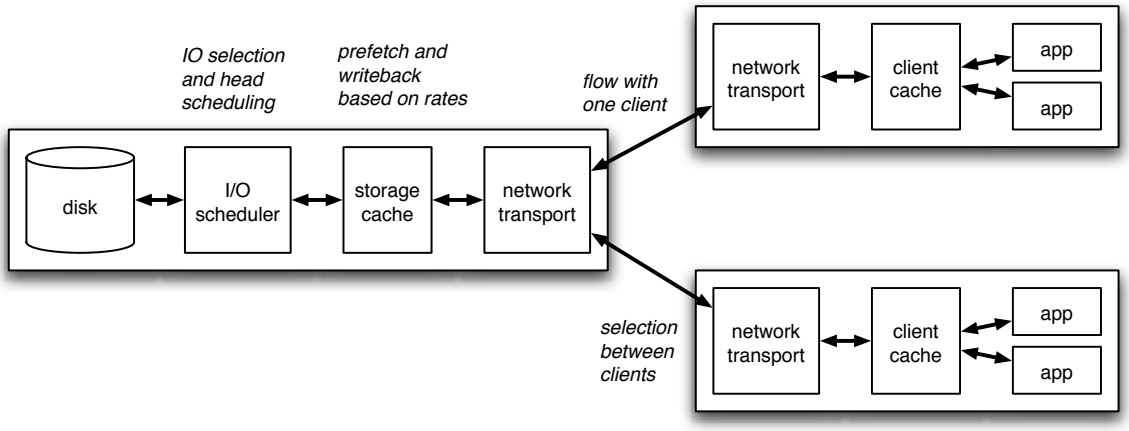


Figure 2: Components in the I/O path.

In the RAD model, feasibility of a task set is trivially verifiable by summing up the resource usage of each of the tasks sharing a resource; if the sum is less than 100% of the available resource(s), the task set is feasible. Scheduling is done at run-time and may be accomplished with any rate-enforcing optimal real-time scheduler[2]. For CPU scheduling we use a version of EDF with timers to interrupt jobs that have used up their allocated budget for the current period.

A *resource broker* is responsible for translating varied application requirements into a uniform representation of application needs and for performing the feasibility verification required for robust admission control. This depends upon the existence of a uniform resource allocation and scheduling model for all managed resources.

In order to manage the diverse resources in our system, we have had to extend the RAD model in a number of different ways. Achieving good disk performance requires both a guaranteeable metric of performance as well as careful management of the workload to ensure and maintain the physical and temporal contiguity of related requests. We manage disk performance in terms of disk head time, which is reservable and guaranteeable up to 100% of the available time [4]. We also add a third layer to the model allowing the reordering of disk requests [11]. Disk requests are dispatched according to both deadline requirements and performance heuristics.

Our simple storage network behaves somewhat like a single CPU in that each transmit port may only serve one client's data at a time. Unlike a CPU, the control of the network is decentralized; each client must independently decide when it will start and stop transmitting data. The RAD model remains relatively intact for network scheduling, but the scheduler is quite different. We introduce a novel network scheduler called Less Laxity More that is intended to approximate the behavior of Least Laxity First without centralized control.

Our work on buffer cache management currently focuses primarily on the storage server. Although cache memory can be relatively trivially partitioned according to the memory needs of each process, the RAD model determines the partition by indicating exactly how much cache is needed for each process. Each task must be able to store a multiple of the amount of data that may be transferred per period. Interestingly, this means that the best case for the disk is also the worst case for the cache, as described in Section 5. The cache may also be used for rate and period transformation between the client and the disk, allowing the client to temporarily transfer data at a higher rate than the disk allows, and to transfer data with a smaller period than is feasible for our disk scheduler.

Because each of the resources is managed via the RAD model, the guarantees are easily composable. Although the utilization of different resources vary for a given task, the deadlines will be the same, allowing for simple synchronization of the use of the different resources. Overall, if the reservation for a given I/O stream is satisfiable on each of the resources, the stream can be admitted and its I/O performance can be guaranteed.

The following sections discuss our management of each of the resources in more detail.

3 Guaranteed disk request scheduling

Our real-time disk scheduler is designed to meet three goals. First, the scheduler must provide guaranteed, integrated real-time scheduling of application request streams with a wide range of different timeliness requirements. The mechanical nature of disks adds an additional set of require-

[2] A sub-optimal scheduler may also be used, with a suitably modified feasibility test

ments. Sequential accesses experience orders of magnitude lower latencies than random accesses, and good disk scheduler can significantly improve performance by reordering requests to increase sequentiality. Thus, as a second goal, our disk scheduler must provide not just guaranteed performance but good performance. Finally, in a shared storage system, performance of an I/O stream may be affected by seeks introduced by competing I/O streams. Therefore, the scheduler must also isolate I/O streams from the behavior of others so that none of the streams cause another to violate its requirements.

Traditional real-time disk schedulers guarantee reservations on throughput [2, 13, 12]. However, due to the orders of magnitude difference between best-, average-, and worst-case response times, hard throughput guarantees on general workloads require worst-case assumptions about request times allowing reservations of less than 0.01% of the achievable bandwidth. Our Fahrrad real-time disk I/O scheduler [11] uses a different approach based on *disk time utilization* reservations [4]. A reservation consists of the *disk time utilization u* and the *period p*. Disk time utilization specifies an amount of time a disk will make available for a given request stream to service its I/O requests. The period specifies the granularity with which the request stream must receive its reserved utilization. Reservations are associated with I/O request streams, which represent related sets of requests that may come from a single user, process, application, or a set of these.

Fahrrad implements the RAD model and adapts it to disk scheduling. Since the basic goal of our scheduler is to provide a full range of timeliness guarantees, Fahrrad implements the two layers of the RAD model: resource allocation and dispatching. Resource allocation is done via the broker, which ensures feasible resource allocation and maps application requirements into disk time utilization and period. I/O request dispatching, which chooses which I/O stream requests to process, is based loosely on EDF. Because disk I/O is stateful, adapting the RAD model to disk scheduling requires the addition of a third layer concerned with I/O request *ordering*. Fahrrad allows request ordering by logically gathering as many requests as possible into a set with a property that the requests in the set can be executed in any order without violating any guarantees. We now describe each layer in greater detail.

Resource allocation is made via the broker and consists of two parts: translation of application requirements into a common representation—disk time utilization and period—and admission control on the basis of this representation. Most applications express their I/O performance requirements in terms of throughput and latency[3]. In order to make utilization reservations, applications specify their desired throughput and/or latency and their expected I/O behavior to the broker. Given knowledge about disk performance characteristics, the broker translates throughput and I/O behavior into utilization. When nothing is known about I/O behavior, the broker assumes worst-case request response time, resulting in no worse performance than with throughput-based schedulers.

Applications with no real-time requirements are associated with a best-effort I/O request stream that receives a minimum or remaining unreserved utilization. Latency requirements translate directly to the period reservation. If an application sends I/O requests according to its reservation, its requests will be queued no longer than one period. Since the reservation is guaranteed by the end of each period, the latency is bounded by that period.

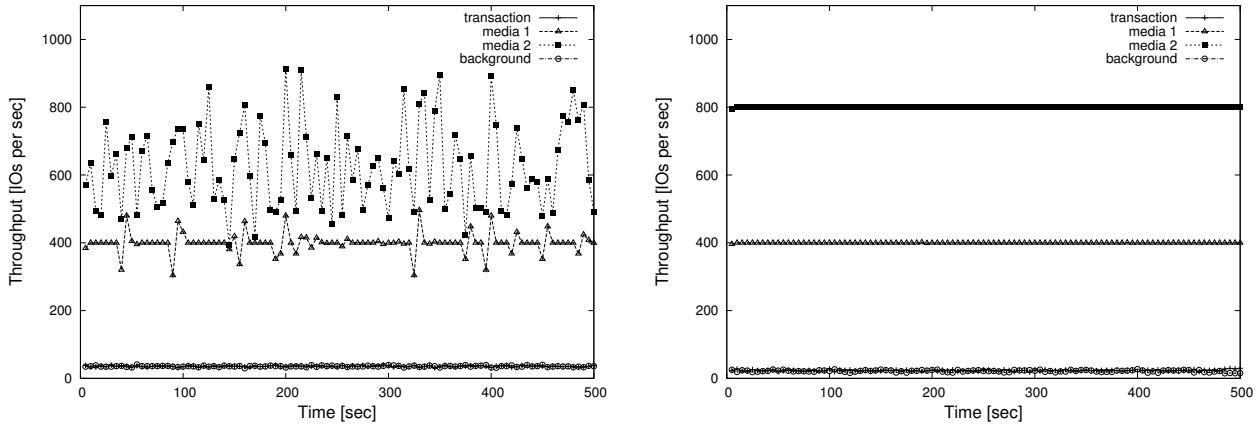
Once translated into the utilization and period, the broker decides that the reservation is feasible as long as the total sum of the utilizations on a given disk (plus a little extra) are less than or equal to 100%. The extra reservation is needed to account for blocking due to the non-preemptibility of I/O requests. In our task model, preemptible jobs are divided into non-preemptible I/O requests analogous to non-preemptible portions of CPU jobs. We have shown previously that a task set is feasible as long as we reserve enough extra time for one worst-case request in the task with the shortest period [11].

In order to guarantee the reserved budget $b = u * p$ for a given stream, the broker has to make an additional reservation. Since service times of I/O requests are not known *a priori* and I/O requests are non-preemptible with a large potential worst-case request time (WCRT), the scheduler cannot issue a request unless there is a worst-case request time left in the current period. Thus, in order to guarantee the desired budget b, the broker must actually budget $b + WCRT$ [11].

Fahrrad guarantees the reserved utilization for each request stream by correctly measuring and accounting for all I/O requests issued and seeks occurred. Fahrrad temporarily assumes that each request takes worst-case time, and allows $\lfloor b_i/WCRT \rfloor$ requests from stream i in the current period into the reordering set. Each time a request completes, the scheduler measures its execution time and updates the budget based on actual execution times. If there is enough budget left to issue one or more worst-case requests, the scheduler continues to dispatch additional requests until the reservation is met. I/O streams whose reservation has been met must wait until their next period to receive more service.

The architecture of Fahrrad is shown in Figure 3, which implements the dispatching and ordering layers of the RAD model. The architecture consists of *request stream queues*, the *Disk Scheduling Set* (DSS), the request *dispatching policy*, and the request *ordering policy*. Each request queue contains the requests from a single I/O stream and requests are ordered by their arrival times. The request dispatching

[3] An exception to this is virtual machines, which want a share of the disk performance with latency bounds.



(a) Linux (b) Fahrrad

Figure 4: Behavior of mixed workload during 500 seconds, with and without Fahrrad. Points are the average for 5-second intervals.

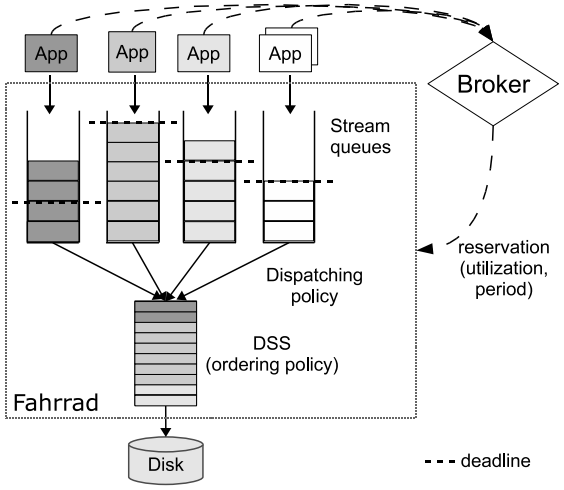


Figure 3: Fahrrad architecture.

policy takes requests from request queues and sends them to the DSS such that DSS always contains the largest set of requests that can be executed in any order without violating any utilization reservations. All requests in the DSS are assumed to take the worst-case time and the number of requests in the DSS is dictated by the earliest deadline in the system. The request dispatching policy moves all requests that have to be executed in the current period from the stream with the earliest deadline. Any remaining time is filled with requests from other stream queues. In order to minimize inter-stream seeking, the dispatching policy tries to maximize the number of requests from the same stream in the DSS (from streams with later deadlines and thus looser scheduling constraints). Since requests are assumed to take worst-case time, the scheduler always guarantees that the stream with the earliest deadline will meet its reservation regardless of the order in which the requests are sent from the DSS to the disk. If requests take less than worst-case time, the dispatcher allows more worst-case requests to the DSS if there is enough space left. The ordering policy takes requests from the DSS and sends them to the disk in an order that optimizes head movement.

While Fahrrad tries to minimize the interference between I/O streams by minimizing inter-stream seeking, some seeks between streams are unavoidable. In order to guarantee isolation between streams, we account for extra seeks caused by inter-stream seeking by reserving "overhead" utilization. We account for these seeks in the reservations of streams responsible for inter-stream seeking and bill these streams for the additional seeking. In this way, the I/O performance achieved from the reserved utilization depends only upon the workload behavior.

Figure 4 shows the performance obtained with Fahrrad. It compares a mixed-application workload running on a standard Linux system (a) and one with Fahrrad (b). The workload combines two “media” streams, a transaction processing workload with highly bursty request arrivals, and a random background stream simulating backup or rebuild. Fahrrad meets both the utilization guarantees and throughput requirements of the I/O streams and its throughput exceeds that of Linux by about 200 I/Os per second.

4 Guaranteeing storage network performance

Most general-purpose networks provide a best-effort service, striving for good overall performance while offering no guarantees. Network hardware with built-in QoS features exists, but is relatively expensive and is usually limited to static configurations that distinguish between classes of

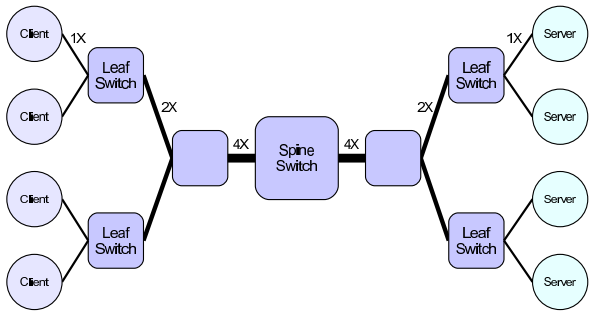


Figure 5: Fat tree network

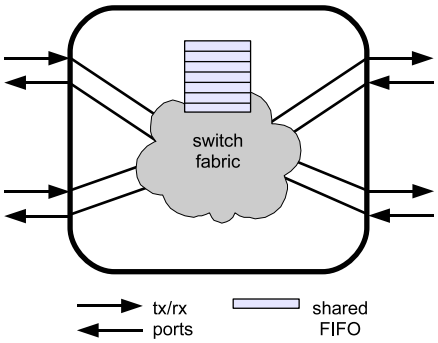


Figure 6: Simple model of a standard Ethernet switch

traffic rather than individual streams. We are interested in a more general cooperative end-to-end protocol that does not rely on specialized network hardware. Adapting the RAD model to the Network (RAD on the Network, or Radon) allows for flexible, general, and fine-grained performance guarantees using commodity network hardware.

We distinguish three major classes of networked storage: Network Attached Storage (NAS), Storage Area Networks (SAN), and distributed file systems. NAS is the most common and least expensive storage network, where one or more servers individually provide a file system interface over a commodity networks. More expensive SANs are composed of storage arrays connected with a high performance network, e.g. Fibre Channel, addressed as a local device. Distributed file systems come in two flavors, for Wide Area and Local Area Networks. Wide area systems generally serve large numbers of users, operate over a large variety of technologies, and are generally grown rather than designed. On the other hand, local area systems are usually designed to provide a high performance parallel file system for a defined clientele. We focus on local area distributed file system.

Figure 5 is our canonical storage network–a closed, full bisection bandwidth, fat tree network of standard Gigabit Ethernet switches. Each of the switches have a set of ports connected via a switch fabric and shared memory for queuing requests, as shown in Figure 6. Packets contending for the same destination port are queued. Continuous contention (congestion) may cause once isolated streams to interfere with each other. In the worst case, the queue will overflow and packets will be lost. Distributed file systems experience a particular case of congestion called incast [5, 10] where a file spread among many servers is sent in simultaneous bursts to a client, which can overflow a switch buffer with little or no warning signs.

Given the theoretical capabilities and limitations of commodity storage networks, the question, “How much of the resource is actually reservable?" has to be answered. We performed a simple characterization with a commodity Gigabit Ethernet switch supporting jumbo frames. Figure 7 shows that one to seven nuttcp UDP clients communicating with the same host achieve linear scaling for aggregate load up to 900 Mbps while experiencing packet loss of under 3% averaged over 10 seconds. Achieved load leveled off with an offered load greater than 900 Mbps while packet loss increased dramatically. The performance of a single connection appears to be limited by a host’s NIC, as a single client reaches a maximum throughput of approximately 600 Mbps over the network and above 1000 Mbps using the host’s loopback device.

These results show that well paced, short burst, fixed rate streams are able to achieve good individual performance with low packet loss while achieving 80% utilization of the network resource. With accurate congestion detection and bounded responses, we expect to be able to further increase overall utilization and decrease packet loss.

Before introducing our model for network resource management, we will briefly describe the most widely deployed end-to-end network protocol, TCP/IP. Network performance is determined by the flow control mechanism, which manages the rate at which data is injected into the network in the absence of congestion. Congestion control mechanisms, adapt the rate and timing of transmissions when congestion is detected. TCP/IP is one of most successful protocols ever developed, but its congestion control algorithms do not allow for any performance guarantees. It continuously tries to increase throughput at the sender by increasing the window (burst) size and uses packet loss as a congestion signal to throttle the sender drastically. Even for a single connection, this results in a sawtooth pattern for throughput over time and a large variance in packet delays, as a the queue continually overflows and drains.

4.1 RAD on Networks (Radon)

The RAD model was originally developed to manage a single resource with a centralized dispatcher. In the case of networks, the RAD model has to accommodate multiple dispatchers for a single resource, where the resource is a transmit port on a switch. The admission process ensures that the aggregate utilization of each switch port is not greater than one. Ideally, dispatchers should be able to cooperatively manage flow control and congestion con-

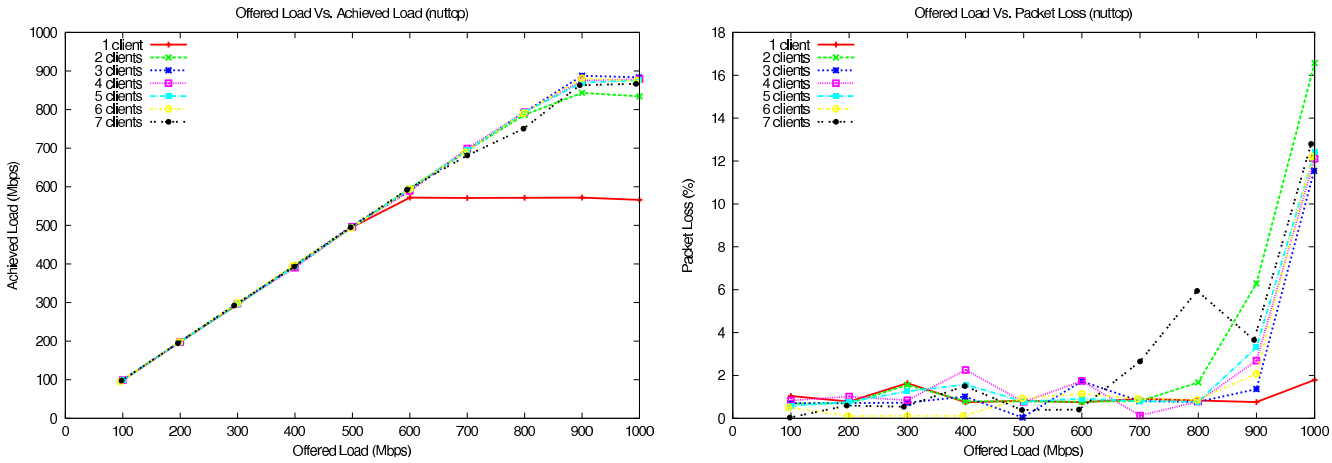


Figure 7: One to seven nuttcp UDP clients offering aggregate loads ranging from 100 to 1000 Mbps

trol based on individual resource allocations, minimizing the use of the queue on a switch. The definitions for the RAD model on networks are as follows:

Resource Allocation A task T_i's reservation (u_i, p_i), where u_i is network time utilization and p_i is the length of the period for which u_i is guaranteed.

Dispatching A task T_i has a budget $e_i = u_i \cdot p_i$, and consists of a sequence of jobs $J_{i,j}$, each having a release time $r_{i,j}$ and a deadline $d_{i,j} = r_{i,j} + p_i$.

The major challenge in guaranteeing network resources is to avoid dispatching synchronized bursts of packets while minimizing communication and synchronization overhead. Ideally this means that a host does not require external information to determine when to dispatch its requests. Scheduling algorithms like Earliest Deadline First (EDF) require all dispatchers contending for the same resource to know the release times of all jobs so that they can agree on the earliest deadline. Furthermore, the clocks of the dispatchers must be synchronized at a granularity corresponding to the smallest possible difference between deadlines. Thus, when a resource is scheduled by multiple dispatchers, a different algorithm is required.

The Least Laxity First (LLF) [8] scheduling algorithm defines the laxity of a job $l_{i,j}$ as the time remaining before the job must be scheduled in order to meet its deadline, $l_{i,j} = d_{i,j} - t - e'_i$, where t is the current time and e'_i is the budget remaining in the period. EDF schedules based on the deadline by which a job must be finished, while LLF schedules based on the deadline by which a job must be started. LLF is optimal for scheduling a single resource in the same sense that EDF is, if a feasible schedule exists, then both will find one [8]. Implementing LLF across multiple dispatchers would require just as much communication and synchronization as EDF, but it lends itself to an approximation suitable for distributed dispatchers because the measure of laxity is relative while deadlines are absolute.

Thus, we propose an approximation to LLF is called Less Laxity More (LLM). As long as no congestion is detected, streams of packets are transmitted as fast as possible up to the allocated budget. When congestion is detected, each sender will use a normalized notion of a job's laxity–percent laxity–the ratio of laxity to the total remaining time until the deadline. More formally:

$$\%laxity = \frac{l_{i,j}}{d_{i,j} - t}$$

This definition of urgency can equivalently be expressed in terms of budget since

$$\%budget = (1 - \%laxity)$$

4.2 Flow Control and Congestion Control

Before developing LLM, we simulated different flow control mechanisms based on the queuing model shown in Figure 8. One mechanism was to implicitly drive flow control by the disk performance reservation. This simulation uses a token bucket model where clients are allowed to submit a storage request to the system when tokens are available. Tokens are managed by the server, which is constantly monitoring the current performance of each client. However, in the most promising simulation, clients replenish tokens required to achieve the reserved performance themselves based on server-assigned rates and periods, while the server directly manages tokens for unused resources. This shows that flow control fits well with the RAD model.

Congestion can be detected by observing packet loss or by measuring changes in transmission delay. The response to congestion is traditionally a multiplicative decrease in window size. We suggest bounding the change in window

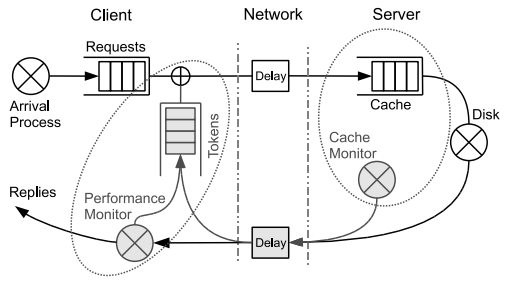


Figure 8: Queuing-theoretic model of Radon

size, making it proportional to percent laxity. Furthermore, we propose explicitly dealing with incast by postponing the dispatch time of the next window based on a model of the current queue depth of the bottleneck switch. Congestion can be detected in its early stages using the measure of relative forward delay as proposed in TCP Santa Cruz [9]. Relative forward delay allows hosts to model the queue depth of a bottleneck switch and allows congestion on the forward and reverse paths to be differentiated. This capability can become crucial on storage networks, where read and write patterns create asymmetric flows.

Flow Control Budget (in packets) $m_i = e_i/pktS$, where $pktS$ is the worst case packet service time

Congestion Control Windows adjusted in size and dispatch time

Window Target $w_{op} = (1 - \%laxity) \cdot w_{max}$

Size Change $w_{change} = \dfrac{-|w_i - w_{op}|}{2}$

Dispatch Offset $w_{offset} = \dfrac{N_{obs}}{pktS} \cdot \text{rand}$

Where w_i is the current window size and N_{obs} is the observed depth of the bottleneck switch's queue. The resulting window size is also obviously bound by the minimum window size and the remaining budget.

Even if individual hosts do not know who among them has the least laxity, they can cooperatively control congestion using the relative measure of their own laxity.

5 Buffer management for I/O guarantees

The goals for our buffer-cache in the context of performance management are two-fold. First, the buffer-cache must provide a single solution that addresses a continuous spectrum of performance guarantees, ranging from best-effort to hard real-time. The buffer-cache must guarantee capture and retention of data as long as needed, but not any longer, before forwarding it to a device. The second goal of the buffer-cache is to enhance the performance of devices, allowing performance reservations for rates and periods that the devices may not be able to provide by themselves.

Buffering serves three main functions. First, buffers are used to stage and de-stage data, decoupling components and introducing asynchronous access to the devices. The second function of the buffers are speed matching between different devices allowing fast transfers to/from slow devices. Finally, they are used to shape traffic between devices, increasing or decreasing burstiness in the workload. The ability to decouple components is driven by the amount of buffers available to the system. Speed matching is dependent upon the transformation of one component's rate to another and vice versa. Finally, the shape of the workload is influenced by the length of the period, among other factors.

Buffering can also be used for caching by placing a small, fast storage device in front of larger, slower device. Distributed storage systems use buffering on a number of system components such as storage clients, storage servers, network switches, and disks. Storage clients, for example, use caching to capture working sets in order to consolidate reads and writes. Storage servers employ caches to stage and de-stage data, capture request bursts, and prefetch data based on sequential access patterns.

In this section we will focus on buffering in storage servers but we believe that many of the principles apply to buffering in general. For the rest of the section we refer to buffering applied to storage servers as buffer caching. For now we also assume that the buffer cache is also non-volatile, as is standard in storage servers.

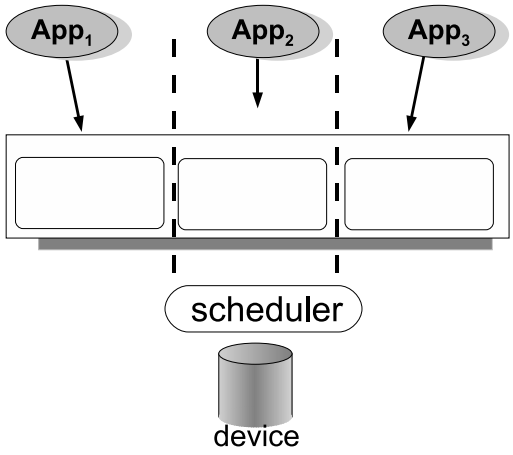


Figure 9: Cache Architecture.

Decoupling components, such as the disk and the network, requires enough buffer space to be allocated. Each application receives a dedicated partition as shown in Figure 9. The amount of buffer space assigned to each stream is a product of the guaranteed rate and period, and also influenced by workload characteristics and performance goals. Streams with performance requirements receive a minimum amount of dedicated buffer space based on worst case

request times on the device, whereas streams with soft-performance requirements might receive buffer space according to average case request times. Streams with no performance requirements are aggregated into a single reservation and may receive any uncommitted resources from streams with performance needs.

Maximizing the utilization of a resource requires worst-case buffer space allocation. We allocate space for as many requests as can be served by a device during a period based on the best case request time for the device. Thus the best-case on the device represents the worst case buffer allocation. This amount represents an upper bound on the buffer space needed within a period for read-only streams. In the case of streams involving writes, allocating extra buffers enables delayed write-back to the disk.

Embedding an application's behavior into the performance reservation (*e.g.*, sequential/random ratio, read/write ratio) allows efficient allocation of resources, for example by allocating less buffers for random streams. Efficient resource allocation can be achieved to the extent an application's workload can be characterized. If such characterization is missing, default worst-case assumptions are made.

An application's reservation is transformed into a resource reservation by means of rate and period transformations. *Rate transformation* and *period transformation* are mechanisms which allow the buffer-cache to decouple an application's reservation from the underlying device's capabilities while maintaining performance guarantees. It is possible to shape bursty workloads, using period transformations, into uniform accesses over long periods of time when that results in making better use of the device (*e.g.*, network). Similarly, by transforming short periods into long periods it is possible to introduce burstiness into the workload, reducing device utilization and overhead (for example, extra seeks on disks).

The period length of write-only streams can be elongated by means of *period transformation*, provided enough buffer space is available to hold the additional updates. It is possible to remove extra seeks in a predictable manner by transforming shorter periods into longer periods. The overhead previously imposed by short periods is then transformed into reservable utilization that could be used for admitting more request streams. Finally, since buffers have no context switch cost between stream requests, it enables reservations with very short periods. This turns some unmanageable scenarios into feasible situations that can be supported by the whole system.

Rate transformation is achieved by means of speculative reads and delayed writes. Rate transformation decouples an application's rate from a device's rate, allowing faster rates than the device can actually support. Finally, exposing faster rates to the application results in faster access times per request, allowing applications to release requests closer to the end of the period without missing deadlines.

6 Conclusion

End-to-end performance management in a complex, distributed system requires the integrated management of many different resources. The RAD integrated scheduling model provides a basis for that management and is adaptable to a variety of different resources. Based on a separation of the two basic resource management questions—how much resources to allocate to each process and when to provide them—RAD supports a wide variety of different types of processes. Our ongoing work demonstrates the applicability of the model to CPU scheduling, disk scheduling, network scheduling, and buffer cache management.

Our future work focuses on fully generalizing the RAD model. The addition of constraints—required processing characteristics such as deadlines—and heuristics—desired processing characteristics such as minimizing jitter or task migrations—give the model sufficient flexibility to describe a wide variety of existing and hypothetical schedulers with different properties. We are also extending the model to apply to additional resources and dimensions, including multi-processor and multi-disk scheduling.

Acknowledgements

This work was supported in part by National Science Foundation Award No. CCS-0621534. Additional support was provided by Los Alamos National Laboratory. The other members of our RADIO research team–Richard Golding and Theodore Wong—contributed significantly to the development of the ideas in this paper.

References

[1] S. A. Brandt, S. Banachowski, C. Lin, and T. Bisson. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003)*, pages 396–407, Dec. 2003.

[2] J. L. Bruno, J. C. Brustoloni, E. Gabber, B. Özden, and A. Silberschatz. Disk scheduling with quality of service guarantees. In *Proceedings of the 1999 IEEE International Conference on Multimedia Computing and Systems (ICMCS '99)*, pages 400–405, 1999.

[3] W. Hawkins and T. Abdelzaher. Towards feasible region calculus: An end-to-end schedulability analysis of real-time multistage execution,.

[4] T. Kaldewey, T. Wong, R. Golding, A. Povzner, C. Maltzahn, and S. Brandt. Virtualizing disk performance. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2008)*, Apr. 2008.

[5] E. Krevat, V. Vasudevan, A. Phanishayee, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan. On application-level approaches to avoiding TCP throughput collapse in cluster-based storage systems. In *Proc. Petascale Data Storage Workshop at Supercomputing'07*, Nov. 2007.

[6] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, and J. Hanson. A scalable solution to the multi-resource QoS problem. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS 1999)*, Dec. 1999.

[7] C. Lin, T. Kaldewey, A. Povzner, and S. A. Brandt. Diverse soft real-time processing in an integrated system. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS 2006)*, pages 369–378, Rio de Janeiro, Brazil, Dec. 2006.

[8] A. K. Mok. *Fundamental Design Problems of Distributed Systems for Hard Real-time Environment*. PhD thesis, Massachusetts Institute of Technology, May 1986.

[9] C. Parsa and J. J. Garcia-Luna-Aceves. Improving TCP congestion control over internets with heterogeneous transmission media. In *Proceedings of the 7th IEEE International Conference on Network Protocols (ICNP)*. IEEE, 1999.

[10] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan. Measurement and analysis of tcp throughput collapse in cluster-based storage systems. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[11] A. Povzner, T. Kaldewey, S. Brandt, R. Golding, T. M. Wong, and C. Maltzahn. Efficient guaranteed disk request scheduling with fahrrad. In *Eurosys 2008*, April 2008.

[12] L. Reuther and M. Pohlack. Rotational-position-aware real-time disk scheduling using a dynamic active subset (DAS). In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003)*, Dec. 2003.

[13] T. M. Wong, R. Golding, C. Lin, and R. Becker-Szendy. Zygaria: Storage performance as a managed resource. In *12th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS06)*, Apr. 2006.

Integrating Real Time and Power Management in a Real System

Martin P. Lawitzky[‡§¶] David C. Snowdon[‡§] Stefan M. Petters[‡§]

[‡] NICTA* Sydney, Australia

[§] University of New South Wales Sydney, Australia

[¶] TU München Munich, Germany

firstname.lastname@nicta.com.au

Abstract

Deploying dynamic voltage and frequency scaling (DVFS) techniques in a real-time context has generated some interest in recent years. However, most of this work is based on highly simplifying assumptions regarding the cost and benefit of frequency scaling. We have integrated a measurement-based DVFS technique with an EDF based scheduling framework. This enables the use of the dynamic slack caused by the variability of execution time, to reduce energy consumption and thus extend battery life or reduce thermal load. We have tested the approach using hardware instrumentation on a real system. This paper describes not only the theoretical basis for the work, but also our experiences with DVFS when confronted with physical reality.

1 Introduction

Power management in embedded devices may be motivated by several factors: extended battery lifetime, reduced need for heat sinks and other thermal dissipation devices, a limited power supply (e.g. a solar powered system), or simply improved environmental sustainability.

Various policies for energy savings have been widely deployed in portable devices like laptop computers without real-time requirements. However, due to their interactive nature, heuristics, with ill defined impact on the temporal behaviour of the system are acceptable. This is obviously not the case for real-time systems where temporal behaviour is considered a prime system property similar in importance to the functional behaviour.

In the last ten years a large body of work has been devoted to the integration of power management policies combined with real-time scheduling. However, most approaches assume an inversely proportional relationship between the CPU core frequency and execution time and ignore issues like a substantial frequency switching cost, static power consumption or the effect of a changing memory frequency on performance and power. All of these issues are evident in real hardware platforms.

Within this work we attempt to take our experience with the physical reality of DVFS and develop an integrated real time and power-management scheduling framework. In order to perform DVFS in a real-time environment, tracking of dynamic slack is essential. Dynamic slack is encountered thanks to the difference between the worst case execution time (WCET) and the actual execution time. For most software the actual execution time is subject to substantial variability as the code is subject to different input parameters at run time.

The RBED work by Brandt et al. [1, 2] is an earliest deadline first (EDF) based scheduling framework. Tracking of system slack is an integral part of this scheme. Major advantages of the approach are high utilisation (thanks to EDF), and integrated non-RT and RT scheduling. It does this by temporal isolation and thus ensures a graceful degradation in the event of an overload situation.

We have set out to integrate power management in an RBED-based scheduling framework. To address the simplifications of previous work, a good model for the temporal- and energy-consumption impact at different frequency settings is required. In previous work we have developed such time [3] and energy [4] models. Our proposed approach allows for arbitrary and frequency-dependent cost of a frequency switch in the time and energy domain. Examples for such requirements are XScale-based processors or the Crusoe [5].

Based on these models we have developed a scheme which makes use of the implicit slack tracking of the RBED framework. We have implemented this work on

*NICTA is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs

a XScale-based platform within the OKL4 microkernel. In this initial paper we have kept a number of assumptions made by the original RBED approach about real-time tasks: we assume periodic, non-communicating tasks where deadlines are runnability requirements (i.e. a job must complete before the next job from that task is released) on a single processor. The relaxation of these assumptions is subject to ongoing work within our group.

The next section will cover the most relevant related work before we revisit our previous modeling work and an overview of RBED. Section 5 details our extensions to the RBED model to integrate our power management work. The implementation and our evaluation of it are the subject of Section 6 before summing up with the conclusions.

2 Related work

Strategies for dynamically scaling voltage and frequency (DVFS) to save energy is a wide research area. A number of groups have developed approaches towards both power management in real-time systems and performance prediction under frequency scaling. Within this paper we concentrate on the small representative set of these contributions which we consider most relevant to our work.

Using DVFS when considering timing constraints raises the need for precise execution-time prediction. Weissel and Bellosa [6] explored event-counter-based prediction of performance degradation as part of their development of a best-effort DVFS scheme. Monitoring certain system events allows predictions which are within a small margin of error compared with the widespread, but unrealistic, assumption of a linear relationship between core clock frequency and execution time. Refining this approach, Choi at al. [7,8] described the effects of on-chip and off-chip cycles towards performance degradation under frequency scaling. Performance monitoring counters (PMCs) and an on-line regression technique were used to calculate the balance between these two. One disadvantage of this technique was the $100\mu s$ processing time required at each frequency switch. While these papers do explicitly evaluate timing constraints, they do not target real-time systems. Ultimately they had a substantial impact on the development of our previous work [3, 4].

A number of groups [9–12] have integrated DVFS and real-time scheduling approaches. Pillai and Shin [9] simulated their *RT-DVS* algorithm for different task sets and different hardware configurations. They implemented a kernel extension for the Linux scheduler and have shown that deadline-based approaches can perform better than fixed-priority-based scheduling (since they have knowledge of the future workload).

The scheduling of sporadic tasksets was addressed by Qadi et al. [10]. They achieved significant power savings for their particular problem. This work is closely related to [9] and uses off-line techniques to determine a global level of slowdown. Both fall in the category of inter-task DVS, not exploiting the possibilities of DVS inside running jobs.

Dudani et al. [11] used system slack time to allow certain jobs to run at lower CPU core frequencies based on the two common assumptions of many DVFS papers: that execution times scales linearly with the CPU clock frequency and that running jobs slower implies saving energy. Unfortunately, simulating an ideal real-time system without taking comparing with the behaviour of a real system did not expose the issues with their proposed solution.

The real-time DVFS approaches above are based on the assumption that stretching workload maximise utilisation saves energy in general. Depending on the hardware, this assumption may be misleading (as shown by Aydin et al. [12], our previous work [4]).

Exploring the possibilities of DVFS for periodic task sets, Aydin et al. [12] have shown that premature frequency scaling can even lead to *increased* power consumption. Careful consideration of on-chip and off-chip workload has to be made to achieve considerable system wide power savings. However, similarly to the other approaches, Aydin et al. [12] evaluate their approach in idealised simulations rather than in a physical system environment.

The effects of frequency switching overhead which represent significant, un-interruptible sections, were largely ignored in recent work. However, our paper shows how these side effects can be managed in real-world real-time systems.

3 Prediction of Time and Energy

In our previous work [3, 4] we have developed an accurate method to model time and energy consumption under DVFS. This is now deployed in the implementation of the DVFS-RBED policy. Here, we briefly introduce the relevant concepts. Further detail is provided in the original publications [3, 4].

3.1 Time

The execution time for a given piece of software can be described as the sum of the times spent waiting on different functional units of the system. This can be time in the CPU core actually executing instructions, time spent waiting for main memory, time spent waiting for I/O operations to complete and so on.

All of these operations can be considered as scaling inverse proportionally with their respective clock frequencies. This allows the execution time C to be expressed as follows:

$$C = \frac{c_{cpu}}{f_{cpu}} + \frac{c_{bus}}{f_{bus}} + \frac{c_{mem}}{f_{mem}} + \frac{c_{io}}{f_{io}} + \dots \quad (1)$$

The coefficients c_x can be interpreted as being caused by a number of events combined with an number of wait-states associated with each of these events. Ideally there would be a way to observe the events directly. However, many CPU cores provide us with a means to observe some events which are correlated with the events in question. Depending on the architecture chosen, these have different names; e.g. *event ocurrence counter* for the PowerPC or *performance event counter* in AMD chips. In Intel chips these are usually called *performance monitoring counters* (PMCs) and we are going to use this term for the remainder of the paper. The available events which can be observed varies widely between architectures, but usually include beside many others *good* predictors like cache misses, TLB misses, or write backs.

Within this paper we focus on events concerning memory accesses, which in this case involve the bus and the memory frequency. In our sample platform the I/O is not subjected to a separate frequency, hence we will concentrate will only use c_{cpu}, c_{bus}, and c_{mem} respectively. Depending on the number of appropriate PMCs available (in this case, 2), the equation can be linearly extended.

$$\begin{aligned} c_{bus} &= \alpha_1 PMC_1 + \alpha_2 PMC_2 + \dots \\ c_{mem} &= \beta_1 PMC_1 + \beta_2 PMC_2 + \dots \end{aligned} \quad (2)$$

The PMC readings are application specific, while the coefficients α_i and β_i are architecture specific. Therefore, a given hardware platform simply needs to be calibrated.

Most architectures provide a cycle counter. This expresses the execution time C of an application in terms of the cycles c_{tot} of the CPU core frequency f_{cpu}. As such the final parameter c_{cpu} can be computed based on a single measurement:

$$c_{cpu} = c_{tot} - \frac{f_{cpu}}{f_{bus}} c_{bus} - \frac{f_{cpu}}{f_{mem}} c_{mem} \quad (3)$$

Now that all parameters of the model are instantiated, we can, from a measured part of the application, reason about the progress the application would make at the fastest frequency setpoint and the time required to execute the remainder of the task in any given frequency. This will be further explained in Section 5.2.

3.2 Energy

The energy model follows a similar logic to the time model: the energy consumed by the system depends on the properties of the workload. It depends largely on the number and type of operations performed, as well as the static power over a given time interval Δt.

$$\begin{aligned} E = {} & V_{cpu}^2(\gamma_1 f_{cpu} + \gamma_2 f_{bus} + \gamma_3 f_{mem})\Delta t + \\ & V_{cpu}^2(\alpha_0 PMC_0 + \dots + \alpha_m PMC_m) + \\ & \beta_0 PMC_0 + \dots + \beta_m PMC_m + \\ & \gamma_4 f_{mem}\Delta t + P_{static}\Delta t, \end{aligned} \quad (4)$$

Equation 4 consists of five groups of terms.

1. The first term accounts for the constant event rate owing to the CPU, memory and bus clocks within the CPU (in our test system, the memory controller and processor bus are on-die). The number of cycles executed is proportional to the time. These are proportional to the the square of the CPU core voltage.

2. Similarly, the next term associates certain events described by performance monitoring counters PMC_i with energy consumed within the chip (i.e. proportional to V_{cpu}^2). For these, only the event count, rather than the frequency are relevant.

3. The PMCs are also used to describe off chip events, like memory accesses, for which the CPU core voltage scaling has no impact.

4. The memory bus frequency is seen external to the CPU, and is therefore accounted for by a term which is independent from the CPU core voltage.

5. Finally the static power constitutes all of the power not effected by frequency or workload changes (and the energy is therefore proportional to the time spent executing).

The performance monitoring counter values PMC_i have to be a measure of the events during the time interval Δt. In turn the time interval at some target frequency can be expressed using Equation 1. For clarity of the presentation, we have not combined the equations as the result would become unwieldily.

While Equation 4 is presented with a single scalable voltage domain (the CPU core voltage) and a constant voltage domain, it is trivial to extend to multiple scalable voltage domains. Currently, the model does not explicitly take the effects of I/O events into account. The task model assumes that blocking on I/O will invoke the scheduler which initiates a switch to a different task. In this case, I/O is not accounted to the job which actually initiates the I/O operation, but the one executing during the I/O operation. However, as the energy model is determined offline this effect is not taken into account. The effects of I/O on the model are subject of ongoing research in our group. Further details on performance counter selection, etc, can be found in our prior work.

4 RBED Summary

Since our work is based on the RBED scheduling framework developed by Brandt et al. [1] we will briefly introduce the the background and concept of the RBED approach. Devices with mixed timing requirements represent the majority of today's embedded systems. Many of them allow the installation of arbitrary user software which may have unknown timing behaviour. Demanding precise worst-case execution-times at installation time is not feasible. Therefore, other precautions must be taken to ensure that the behaviour of any given application cannot harm the provision of timing requirements of other programs. The misbehaviour under overload conditions is one of the major shortcomings of classic *earliest deadline first* (EDF) scheduling. Brandt et al. developed a multi-class real-time scheduler for the seamless support of mixed hard, soft and non real-time applications. It ensures a timely separation of tasks by preemptability and a resource allocating governor. Resource allocation takes place at run-time where tasks dynamically request a share of the CPU time. The resource allocator can be implemented as a user space application that performs an on-line schedulability analysis. The preemptive scheduler implements EDF but ensures that only resources granted by the resource allocator are used.

One advantage of this approach is that allocation of resources for soft real-time tasks can be independent of the actual WCET of its jobs. Thus, for a soft real-time task, where an application may miss (a small number of) deadlines, instead of using the worst-case execution time, a smaller timeslice may be allocated. This allows over-allocation of the system while keeping single applications in temporal isolation; i.e. one soft real-time job exceeding its allocated resources has no impact on the schedule of other tasks. This allows seamless integration of hard real-time, soft real-time and best-effort tasks in one system with a unified scheduling policy.

U	utilisation of the entire taskset, $U = \sum_{\forall i} u_i$
u_i	utilisation of a given task τ_i, $u_i = E_i/T_i$ at the top frequency setpoint
$r_{i,n}$	release time of a given job $J_{i,n}$
$d_{i,n}$	absolute deadline of a given job $J_{i,n}$
$x_{i,n}$	current service time $u_i(t-d_{i,n-1})$ of a given job $J_{i,n}$
C_i	WCET of a given task τ_i at the top frequency setpoint
E_i	budget allocated to task τ_i
D_i	relative deadline of task τ_i
T_i	period/minimal inter arrival time of task τ_i
C_i^*	part of current job of task τ_i completed (equivalent at top speed)

Figure 1: Nomenclature Used

For a complete description of the algorithms including the proof of correctness, we refer to Brandt's original work [1]. For the relevant nomenclature see Figure 1.

Given the feasibility of dynamic resource dispatching, the introduction of per-job budgets allows for a trivial measurement of resource usage and on-line reallocation of slack time. The task model is illustrated in Figure 2 and can be described as follows:

A task τ_i consists of multiple subsequent jobs $J_{i,r}$. These jobs cannot overlap ($r_{i,r+1} \geq r_{i,r} + T_i$).

Every job is preemptible at any time and the jobs of a task have a minimum inter-arrival time (IAT) which is called the period in the case of periodic tasks. No more than one job can be released within the period/IAT. Each job has a deadline relative to its release time. For the scope of this paper, the relative deadline equal to the period of the job. ($D_i = T_i$)

Each job has a worst case execution time (WCET) C_i which can be obtained using well-known techniques and is very likely to be larger than the actual execution time $x_{i,r}$. As part of the RBED scheduling, a budget E_i is reserved to each job $J_{i,r}$ representing the timeslice that must be granted to the job by the scheduler. In case

of hard real-time requirements, the budget equals the WCET ($E_i = C_i$) to guarantee timely completion of all jobs. For soft real-time requirements, the assigned budget may be smaller than the WCET $E_i \leq C_i$ to guarantee timely completion of all jobs $\{J_{i,r} | x_{i,r} \leq E_i\}$.

In most cases, the job will finish in less time than reserved ($x_{i,r} \leq E_i$). The remaining budget $E_i - x_i$ is reserved, but not used and is therefore called slack $S_{i,r}$.

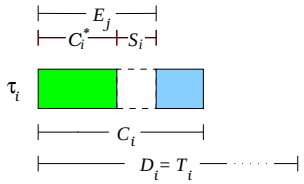


Figure 2: Parameters of Job J_j

If a per-job budget E_i assigned to a task τ_i is smaller than the WCET of a single job $J_{i,j}$ of τ_i, the job's runtime can potentially exceed the budget. Since this is an overload situation and would likely have drastic consequences in classic EDF scheduling, precautions are taken in RBED. The scheduler preempts every job when it has used up its budget E_j and extends the deadline D_j of the job by one period T_j. At the same time, the job's budget E_j is refilled to the assigned value. Thus, the remainder of the execution is postponed until it is scheduled again.

This guarantees a temporal isolation among all tasks in the system as long as $U = \sum_{\forall i} u_i \leq 1$ with $u_i = E_i/T_i$. To ensure this condition is met, the resource allocator (RA) which may be an ordinary user space task must acknowledge all requests for changing job's periods, budgets or deadlines. The idea of dynamic reallocation of processing time is based on five theorems.

Theorem 1 The earliest deadline first (EDF) algorithm will determine a feasible schedule if $U \leq 1$ under the assumption $D_i = T_i$.

Theorem 2 Given a feasible EDF schedule, at any time a task τ_i may increase its utilisation u_i by an amount up to $1 - U$ without causing any task to miss deadlines in the resulting EDF schedule.

Theorem 3 Given a feasible EDF schedule, at any time a task τ_i may increase its period without causing any task to miss deadlines in the resulting EDF schedule.

Theorem 4 Given a feasible EDF schedule, if at time t task τ_i decreases its utilisation to $u_i' = u_i - \Delta$ such that $\Delta \leq x_{i,n}/(t - r_{i,n})$, the freed utilisation Δ is available to other tasks and the schedule remains feasible.

Theorem 5 Given a feasible EDF schedule, if a currently released job $J_{i,n}$ has negative lag at time t (the task is over-allocated), it may shorten its current deadline to at most x_i/u_i and the resulting EDF schedule remains feasible.

The resource allocator algorithm can be described as follows: U_{Kernel} describes the worst case utilisation required by the operating system. $U_{BE,min}$ describes the minimum reserved utilisation reserved for all best-effort tasks.

1. Assign desired utilisation $U_{HRT,i}$ to all hard real-time (HRT) tasks as long as $U_{HRT} \leq 1 - U_{Kernel} - U_{BE,min}$ where $U_{HRT} = \sum_{\forall i_{HRT}} u_{i_{HRT}}$ Reject all other requests for HRT resources.

2. Distribute utilisation not reserved for hard real-time tasks, best-effort tasks or the operating system among the soft real-time tasks according to their requested resources. In case $U_{SRT} = 1 - U_{Kernel} - U_{BE,min} - U_{HRT} < \sum_{\forall i_{SRT}} u_{i,desired}$ each SRT task is assigned $u_i = u_{i,desired} / \sum_{\forall j_{SRT}} u_{j,desired}$

3. The total utilisation reserved for best-effort tasks is the remaining utilisation which can be described as $U_{BE} = 1 - U_{Kernel} - U_{HRT} - U_{SRT} \geq U_{BE,min}$. U_{BE} is equally distributed among all best-effort tasks.

5 Model Extension

In order to integrate our DVFS work with the RBED approach we need to extend the RBED model. Our task model is based on the idea that each job can be slowed down, if this has beneficial effects for the total energy consumed by the system. Jobs are preemptible and after each preemption the frequency is re-evaluated.

One major issue when performing frequency scaling on real-world architectures is the lengthy time sometimes required to switch voltage and frequency. A frequency switch can be modelled as a substantial atomic section. At design time, this must be accounted for appropriately.

The following section describes how this model can be integrated with the RBED scheduling algorithm.

5.1 Budget for Switching

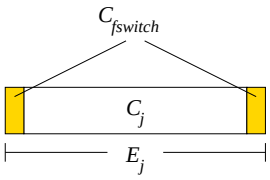


Figure 3: Budget Components

One of the fundamental advantages of RBED scheduling is the absence of a complex schedulability test which requires a priori knowledge of all parameters of the taskset. Allocating time budgets isolates tasks from one another. To guarantee this isolation, it is necessary to ensure that no job can use more resources than allocated *a priori*.

Figure 3 shows, how a time budget must be allocated in order to guarantee isolation within our DVFS framework. The first frequency switch that needs to be considered is the initial one, which may be required if the previous job was scaled to a frequency lower than the minimum frequency necessary to guarantee timeliness.

Each job $J_{j,r}$ can potentially preempt another job $J_{i,s}$ when it becomes un-blocked due to an interrupt if $D_j < D_i$. At preemption time, the scheduler determines the energy optimal frequency setting for job $J_{j,r}$ which may differ from the optimal set point for job $J_{i,s}$. In the case where a frequency switch is performed, sufficient time must be allowed such that the preempted job $J_{i,s}$ is able to restore its frequency to guarantee timely completion. Our approach is the automatic donation the time required for one frequency switch to the preempted job. A second frequency switch must be accounted for in each task's budget. This policy is illustrated in Figure 4 a) and b). These depict the point in time of the preemption and the subsequent donation of the switching cost for the second frequency switch of job $J_{j,r}$ to job $J_{i,s}$. The dashed boxes indicate as-yet unused time budget.

When a job becomes ready and the system is in its idle state, no automatic donation need to be performed since the idle task does not need to restore its frequency set point. In this case, the job can use up the additional budget for further slowdown if beneficial for the system's total energy use.

The key point of this section is this: within a job, an arbitrary number of frequency changes can be performed as long as the budget accounts for two frequency changes.

Technically, it would be sufficient to perform the automatic donation only if a frequency switch actually takes place. However, this requires an unreasonable amount of house keeping without tangible benefit.

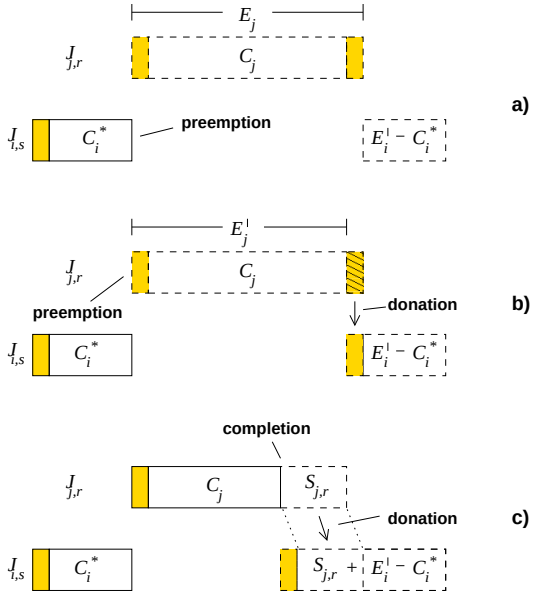


Figure 4: DVFS-RBED Preemption Behaviour

5.2 Dynamic Slack Management

On top of the RBED scheduling algorithm, Lin and Brandt [2] have developed a set of slack reclaiming mechanisms which enhance the temporal behaviour of soft real-time applications in RBED: SRAND and SLAD are consistent with our approach as they allocate slack to the highest priority task as soon as possible. SLASH which attempts to borrow slack from future jobs of the same task and BACKSLASH, which allocates slack to past jobs which have not completed within their budget, could be integrated with our work without conflict.

The task model of RBED allows implicit knowledge of slack time in the schedule. Slack $S_{i,r}$ generated by job $J_{i,r}$ represents the amount of scheduled but unused processing time. This slack time can be pushed forward to the next runnable task's budget in the schedule without harming the timely behaviour of any task in the task set. Such a donation of slack is depicted in Figure 4 c). A formal proof can be found in [2]. It should be noted, however, that this implies that slack is only donated to jobs with a later deadline. When extending the work to communicating and blocking tasks, this condition needs

to be revisited and ascertained.

Since the amount of budget allocated for each job should be sufficient for the job to complete within the given budget of the job, additional budget allows longer processing than required.

Reducing the clock frequencies (f_{cpu}, f_{mem}, f_{bus}) can reduce the total energy consumption for the job's execution. To determine feasible frequency setpoints, the time model in Section 3.1 is used. The set of feasible setpoints $\{\sigma = \{f_{cpu}, f_{mem}, f_{bus}\} | C^{\sigma}_{i,r} \leq E_{i,r}\}$ for job J_i, r is then investigated for their potential effects on the job's energy consumption using the model described in Section 3.2. If a slower execution saves energy, a frequency switch is performed.

The algorithm illustrated in Figure 5 shows the actions to be taken as part of each scheduler invocation. First, all frequency set points are tested for feasibility. Therefore, the time for potential frequency switches is taken into account. Note, the time needed for a frequency switch is not necessarily constant. It is zero in case the frequency is unchanged and may be substantial for other transitions. We assume, changing from set point σ_A to σ_B is a constant, but the transition from σ_A to σ_C may take a different amount of time. A real world example for this behaviour is the Crusoe processor [5]. Xscale processors like the PXA270 or PXA555 have certain frequency combinations which where the transition is almost instantaneous (turbo mode changes) and others which require a substantial amount of time.

Second, the energy consumption for the job at all feasible set points is investigated. The set point which leads to the lowest energy consumption is then chosen.

After returning from a preemption, the job may have received a donation of slack time if the preempting job has not used up its entire reserved budget. Therefore, the energy optimal frequency is recalculated on each scheduler invocation.

One related issue is the absence of a precise measure for the progress of a job. Thus, the progress is estimated using the same time model (Section 3.1). Knowing the frequency set point and the number of events which have occurred during the job's execution so far, the time model can be used to determine the remaining processing time. Figure 6 illustrates this idea. The events depicted represent PMC events. The number of these events is in reality large, but has been limited for illustrative purposes.

A job of task τ_i at maximum frequency σ_{max} is depicted in Figure 6 a) the height of the boxes indicates the power consumption and thus relates to the frequency setpoint. In the example, five events $\epsilon_{1..5}$ happen during execution of this job. Figure 6 b) shows the execution

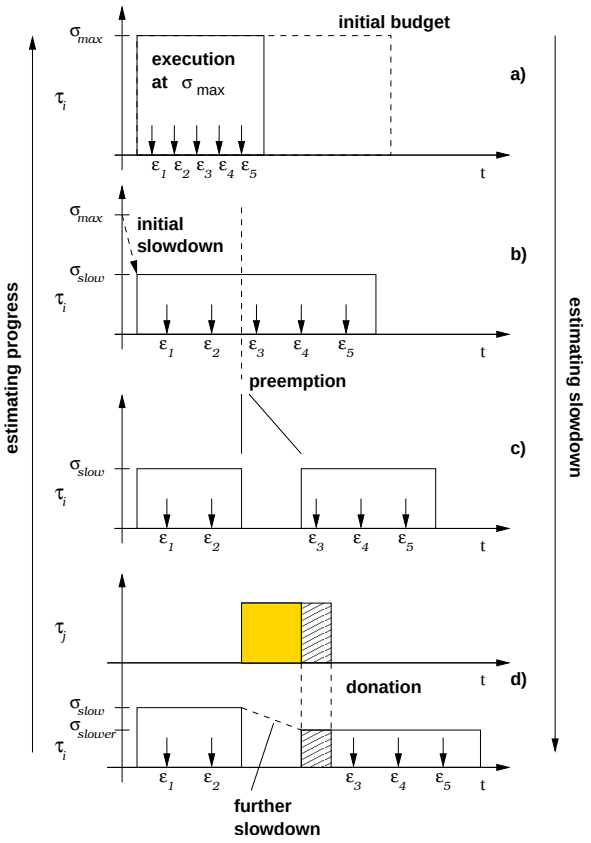


Figure 6: DVFS-RBED Progress estimation and dynamic slowdown

of the same job at a lower frequency set point σ_x. The number of events remains five. The preemption of the job is depicted in Figure 6 c). This preemption changes neither the total execution time of the job, nor the number of events during execution. Figure 6 d) shows the actual execution of task τ_j preempting task τ_i. While the number of events remains five a further slowdown is possible due to a donation of slack from task τ_j.

To perform a further slowdown after a preemption and donation, the remaining runtime of the job $J_{i,r}$ must be determined. A measure for progress is needed. Since there is no absolute measure for an application's progress the scheduler could determine, the estimation technique in 3.1 is performed in the reverse order. The job's execution time so far and number of events can be mapped to an equivalent progress at σ_{max}. We call this the absolute progress $C^*_{i,r}(t)$ of job $J_{i,r}$ at time t.

The remaining execution time at any frequency can now be determined in the reverse order, using the difference between progress and WCET ($C_i - C^*_{i,r}(t)$) and the time model 3.1.

```
newEnergy = energyAtCurrentFrequency
newFrequency = currentFrequency
for frequency in frequencySetPoints
    if executionTimeAtSwitchedFrequency + switchingCost.Time < remainingBudget
            && switchingCost.Energy + energyAtSwitchedFrequency < newEnergy
        newEnergy = switchingCost.Energy + energyAtSwitchedFrequency;
        newFrequency = frequency
if newFrequency != currentFrequency
    switchFrequency(newFrequency)
```

Figure 5: Algorithm

In cases for which the performance degradation can not be estimated safely, the algorithm may use a look-up table to determine the WCET C_i of hard real-time tasks. In such a scenario progress estimation can not be performed safely either, thus a further slowdown is not possible for hard real-time jobs.

Since our mechanism is based on the fact that slack time is released at the end of a job, slack time can only be re-allocated if a job finishes execution. In the case of sporadic tasks, no jobs may be executed for a long period of time. This slack time is currently not taken into account in our algorithm and would be assigned to best-effort tasks (if available) or the idle task.

5.3 Static Slack Management

To gain maximum possible energy savings, static slack time must be distributed entirely. Static slack time describes the amount of time in the schedule which is not allocated. In other words, it describes the sum of all idle times in the schedule.

Static slack which is not allocated to any task cannot be reclaimed or used for power management purposes using our proposed algorithm. We found two different options, to ensure full utilisation of the system. The reader may recall $u_i = E_i/T_i$ where E_i does not necessarily describe the true execution time but the reserved time slice.

One solution is a distribution of static slack among all existing tasks in the system which is later forwarded as dynamic slack since jobs will likely complete earlier than their budget expires. Dynamic slack is then used for optimal frequency scaling. The advantage of this approach is that all tasks benefit from the slack evenly likely leading to a decreased energy consumption.

The other solution is the introduction of a *ghost* task. This task's sole purpose is freeing up its own budget. This approach has the benefit of keeping track of static slack explicitly. This enables easier exploitation of this slack in a dynamic situation where tasks are added at runtime.

6 Implementation Issues and Lessons Learned

6.1 Experimental platform

We implemented the proposed algorithm on an off-the-shelf *Gumstix Connex* platform which runs the L4 microkernel *OKL4 v1.5.2* which was the latest release at the time of implementation and the *Iguana* operating system [13].

In particular, we wrote three software modules implementing the proposed algorithms. First is the preemptive EDF scheduler which replaces the fixed priority scheduling algorithm of the L4 microkernel. Priority was replaced in the task control block by deadlines, budget and period. The scheduler preempts running jobs when the job's budget is used up. If a job completes before its budget is entirely used, the remaining budget which represents the generated slack is enqueued in a deadline sorted budget queue.

Second, the user space resource allocation was integrated in the iguana root task which has special privileges. One privilege added is the exclusive right to perform system calls to the scheduler. This ensures that no other task can make its way around the mechanism. Deadlines, budgets and periods can be allocated at build time or dynamically changed on runtime if the resource allocator permits the change.

Third, a module for arithmetic evaluation of the time model 3.1 and the energy model 3.2 was added to the kernel space scheduler. It evaluates the execution-time estimation for all possible frequency settings and then calculates the energy consumption at all feasible set points. Finally it chooses the set point related to the lowest energy consumption.

Figure 7 shows the software architecture for our implementation which is partially similar to the RBED implementation we received by courtesy of Brandt et al.

The hardware platform consists of a *Gumstix Connex* motherboard, an *Etherstix* network interface as well as an *Audiostix 2* sound card and a *Tweener* serial console

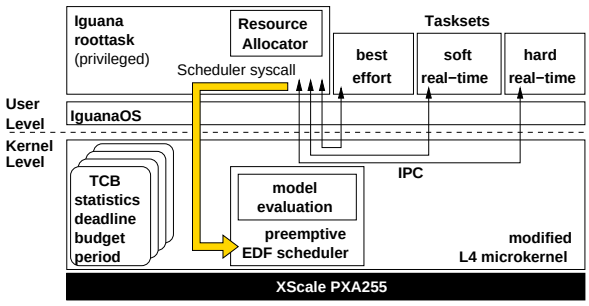


Figure 7: DVFS-RBED Software Architecture

driver. We have chosen this configuration to generate a typical static and dynamic power consumption of an embedded device. The *XScale PXA255* CPU frequency as well as the bus frequency and the memory frequency can enter 22 different setpoints in this configuration. The *Gumstix* platform does not support voltage scaling.

All power consumption was measured with a power- and energy-measurement device developed within the group. This device obtained an accuracy of better than a Milliwatt. The static power consumption of this hardware configuration in idle mode was measured to be $1.13W$. The network interface was measured to consume the majority of this ($0.8W$). While active, at the lowest frequency set point, the power consumption of the device is $1.45W$ and $1.57W$ at the fastest.

The CPU deployed in our experiments implements two types of frequency switches. The standard switch changes frequencies of CPU, bus and memory to one of the 22 different modes. These changes take a substantial time between $500\mu s$ and $600\mu s$. This time is caused by a combination of required operations, like putting the memory in self refresh mode, and the settling time of the phase-locked loop circuitry. The other switching operation implemented in the *PXA255* is called "turbo" mode switching. It allows fast switching between two different CPU frequencies inside frequency pairs, maintaining the memory and bus frequencies. Those turbo frequency changes are intended for peak processing requirements and happen synchronously without disrupting the memory controller or any peripherals. These switches require only a small number of nanoseconds.

Furthermore, the *PXA255* CPU implements a number of low power states which may be useful depending on the maximum acceptable interrupt latency. These low power states also range from single cycle clock gating to extremely deep sleep states which requiring a delay on the order of one millisecond to exit.

6.2 Best effort threads

One of the key points in the RBED scheduling algorithm is the seamless integration of hard real-time, soft real-time as well as best effort tasks. Best effort tasks can be described as tasks which do not necessarily have a periodic or frequent blocking point and do not raise any real-time requirements. Nevertheless, the goal is a guaranteed continuous progress regarding their execution even under high system load to maintain the system in a responsive state.

To keep the scheduling algorithm simple, periods, budgets and deadlines are assigned to all best effort tasks. Thus, these tasks are treated as real-time tasks with artificial deadlines. Depending on the system parameters and the priority of power savings over best-effort performance restricting the execution of best-effort tasks to a small share can save energy. If the execution of best-effort tasks is not restricted, the system may never go to low-power idle mode.

6.3 Lessons learned

Calculating the energy purely needed by one particular job shows, that faster execution saves energy because the execution time decreases faster than the power consumption increases for our particular platform. However, we believe there is a trend in modern processors, with static power taking an increasing share of the overall system power consumption. Since our experimental application is a device which is constantly turned on, the energy required by the device to stay turned on must be taken into account. Due to the high static power consumption of the device and compared to the low dynamic power consumption, the impact of the idle time is substantial. For our particular experimental platform it turned out, that the lowest possible frequency setpoint is always the most energy saving.

The algorithm in Figure 5 which evaluates the time model and the energy model must be performed in fixed point arithmetic because floating point processing would cause too much overhead in general and on the *XScale* architecture in particular, since it does not implement a floating point unit.

Another lesson learned is that a microkernel requires extremely careful implementation of this approach. The first microkernels developed gave the kernel design a bad name because of the large number of context switches are necessary and which expose deficiencies in context switching costs as poor system performance. Years of research and development have lead to the L4 microkernel which was designed for very high *interprocess communication* (IPC) performance. This ap-

proach was successful and L4-based kernels are widely deployed in modern embedded systems. While the success of current L4 microkernels is based on fast context switches and IPC. This advantage is threatened by the overhead of Figure 5 on each context switch.

Our choice, the *PXA255* processor was based on its ability to change between run mode frequency and turbo mode frequency without a substantial switching time. However, the *PXA270* processor implements a half-turbo mode. This processor would have been the better choice in hindsight. Finally, the choice of the *Gumstix* platform results in the inability to use voltage scaling. Deploying voltage scaling would lead to better energy savings, but may add a switching overhead.

7 Conclusion

Within this paper we have shown the integration of real-world power management with a real-time scheduling approach and have reported on the lessons learned from this work.

In the future, we will expand this work beyond the scope of this paper. Core issue in this area is the extension to communicating task sets. The modeling of systems with non rate-based applications (e.g. bursty workloads), is another requirement for real-world deployment. We will investigate the modeling and integration of these tasks within our framework. Furthermore we want to study the effects of deadlines which are shorter than the period on the RBED algorithm in general and on our DVFS extension in particular.

Modelling sporadic tasks as periodic tasks might not be the energy optimal solution. Further investigation is necessary when completely unused reserved time of sporadic tasks can be freed for power management use. As mentioned above, our task model assumes that I/O completions start a job and another blocking operation marks the end of a job. Depending on the hardware, this model may not be sufficiently general. Further investigation of the effects of intra-job blocking on our model remains future work.

8 Acknowledgements

We would like to thank Scott Brandt, Suresh Iyer and Jaeheon Yi for allowing us access to their RBED implementation and documentation which we used to guide our own implementation.

References

[1] S. A. Brandt, S. Banachowski, C. Lin, and T. Bisson, "Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes," in *24th RTSS*, (Cancun, Mexico), Dec 2003.

[2] C. Lin and S. A. Brandt, "Improving soft real-time performance through better slack management," in *26th RTSS*, (Miami, FL, USA), Dec 2005.

[3] D. C. Snowdon, G. van der Linden, S. M. Petters, and G. Heiser, "Accurate run-time prediction of performance degradation under frequency scaling," in *3rd OSPERT*, (Pisa, Italy), Jul 2007.

[4] D. C. Snowdon, S. M. Petters, and G. Heiser, "Accurate On-line Prediction of Processor and Memory Energy Usage Under Voltage Scaling," in *7th Int. Conf. Emb. Softw.*, (Salzburg, Austria), Oct 2007.

[5] M. Fleischmann, "Microprocessor architectures for the mobile internet era," Apr 2002. http://www.marcfleischmann.com/talks/arcs2002.pdf.

[6] A. Weissel and F. Bellosa, "Process cruise control—event-driven clock scaling for dynamic power management," in *CASES*, (Grenoble, France), Oct 8–11 2002.

[7] K. Choi, R. Soma, and M. Pedram, "Dynamic voltage and frequency scaling based on workload decomposition," in *Int. Symp. Low Power Electron. & Design*, pp. 174–179, Aug 2004.

[8] K. Choi, R. Soma, and M. Pedram, "Fine-grained dynamic voltage and frequency scaling for precise energy and performance tradeoff based on the ratio of off-chip access to on-chip computation times," *Trans. CAD ICAS*, vol. 24, pp. 18–28, Jan 2005.

[9] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *18th SOSP*, (Lake Louise, Alta, Canada), pp. 89–102, Oct 2001.

[10] A. Qadi, S. Goddard, and S. Farritor, "A dynamic voltage scaling algorithm for sporadic tasks," in *24th RTSS*, (Washington, DC, USA), p. 52, Comp. Soc. Press, 2003.

[11] A. Dudani, F. Mueller, and Y. Zhu, "Energy-conserving feedback EDF scheduling for embedded systems with real-time constraints," in *LCTES'02*, (Berlin, Germany), pp. 213–222, ACM, 2002.

[12] H. Aydin, V. Devadas, and D. Zhu, "System-level energy management for periodic real-time tasks," in *27th RTSS*, (Rio de Janeiro, Brazil), pp. 313–322, Comp. Soc. Press, Dec 2006.

[13] "OKL4 website." http://www.ok-labs.com/products/okl4.

Middleware for MPSoC Real-Time Embedded Applications: Task Migration and Allocation Services

Elias Teodoro Silva Jr[1,2]; Carlos Eduardo Pereira[1,3]; Flávio Rech Wagner[1]

[1]*Institute of Informatics, Federal University of Rio Grande do Sul, Brazil*
{etsilvajr, flavio}@inf.ufrgs.br
[2]*Federal Center of Technological Education of Ceará, Brazil*
elias@cefetce.br
[3]*Electrical Engineering Department, Federal University of Rio Grande do Sul, Brazil*
cpereira@ece.ufrgs.br

ABSTRACT

The use of MPSoCs (multiprocessor systems-on-chip) is a clear tendency for embedded systems in the current days, especially for consumer markets. Applications are growing in complexity, and multiprocessor platforms can provide performance and flexibility. On the other hand, developers are looking for platforms that help to cope with conflicting design demands, such as low energy consumption, reduced area, timing requirements, and tight time-to-market. This paper proposes to move up the abstraction level to deal with this challenge, by offering a middleware to encapsulate platform details and preserving real-time properties. Task migration and allocation services are emphasized in this paper, and initial results in task migration are presented and evaluated.

1. INTRODUCTION

Real-time embedded systems are expanding and growing in complexity, imposing multiprocessing resources to face high performance and low-energy requirements. MPSoC (Multiprocessor System on Chip) is becoming a widely adopted design style, to achieve tight time-to-market design goals, provide flexibility and programmability, and maximize design reuse. The use of a multiprocessor platform brings with it the well known challenges from parallel and distributed systems [2], related to concurrency. Sometimes, these processors may have a fixed ISA (Instruction Set Architecture); sometimes a mix of processor types is used, like RISC+DSP, for example. Additionally, embedded systems impose restrictions to the solution, like limitations in CPU performance, memory, and power consumption. Therefore, solutions that come from the distributed systems context should be customized to be used for embedded applications.

Developing applications for embedded multiprocessor architectures requires a higher level programming model to reduce software development cost and overall design time [3]. Such a model reduces the amount of architecture details that need to be handled by application software designers and then speeds up the design process. The use of a higher level programming model will also allow concurrent software/hardware design, thus reducing the overall design time.

On the other hand, improving the performance of the overall system requires going through low level programming, exposing architectural properties to the application level.

Since applications are partitioned in processes and processors, a middleware could be used to provide a high level interface, hiding distribution aspects [4]. As a consequence, system resources and application components can be easily reused, saving time for a better application development. In a typical DRE (Distributed Real-time Embedded) system, a middleware usually integrates reusable software components and decreases the cycle-time and effort required to develop high-quality real-time and embedded applications and services [5]. The middleware support has not been investigated in the context of MPSoC applications, but only for DRE systems. Nevertheless, in the context of MPSoCs a middleware could also become an interesting approach to raise the abstraction level, helping to achieve shorter development times. Moreover, energy consumption is a key issue for embedded systems and a high abstraction level development tools should also take such issues into account.

This paper describes a middleware to deal with distributed applications in an MPSoC using a homogeneous ISA (Instruction Set Architecture) and abstracting interfaces between HW-SW implementations and network communication as well. It also includes energy management services, which work transparently, integrated with high level services.

Dynamic task allocation and migration has been shown to be a promising technique to ensure an adequate load balancing among processing units in an MPSoC [6][7], allowing the minimization of some metrics, such as execution time or power consumption. This work proposes

to combine those allocation and migration services with energy management in a transparent way.

The remaining of the paper is organized as follows. Section 2 gives an overview about the development platform. The proposed middleware is presented in Section 3. Task allocation and migration services are described in Section 4. In Section 5, experimental results are presented. Finally, in Section 6 concluding remarks are drawn.

2. Hardware platform

Figure 1 depicts the overall platform architecture, which includes the network and the processor.

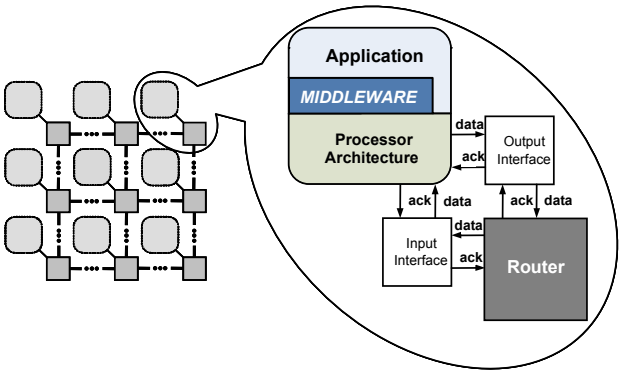


Figure 1: General Platform Architecture

2.1 Configurable processor

Over the last years, Java has gained popularity as a suitable programming language for embedded and real-time systems development. The definition of the Real-Time Specification for Java (RTSJ) standard [1] is the most prominent example of such popularization in the real-time domain.

For this work, a customizable Java processor called FemtoJava [8] is used, which implements an execution engine for Java in hardware, through a stack machine that is compatible with the specification of the Java Virtual Machine (JVM). Different processor organizations are supported, such as multi-cycle, pipeline, and VLIW [9]. For the multi-cycle processor, used for the experiments in this work, all instructions are executed in 3, 4, 7, or 14 cycles, because the microcontroller is cacheless and several instructions are memory bound.

A compiler that follows the JVM specification is used. An environment called Sashimi [8] generates both customized code for the application software and the processor description and allows the synthesis of an ASIP (application-specific integrated processor). The generated code includes the VHDL description of the customized processor core (whose ISA contains only instructions used by the application software), as well as ROM (programs) and RAM (variables) memories and can be used to simulate and/or synthesize the target application. Sashimi eliminates all unreferenced methods and attributes, as well as the unused JVM instructions, automatically customizing and optimizing the final hardware and software code.

2.2 Communication infrastructure

Networks-on-chip [10] have been proposed in recent years as a scalable, high-bandwidth, and energy-efficient communication infrastructure for MPSoCs containing a large number of cores. In this work, the network-on-chip (NoC) SoCIN [11] is used to interconnect the processors inside the MPSoC. SoCIN is based on a flexible router, called RaSoC.

Communication is based on message passing. Messages are sent in packets, which are composed by flits. A flit (flow control unit) is the smallest unit over which the flow control is performed. A flit also coincides with the physical channel word (or phit – physical unit).

SoCIN utilizes wormhole packet switching, so it uses small buffers in the routers, saving size and energy. The routing is XY, which is deadlock free. Each router has 5 bi-directional ports with input buffer size of 4 phits. The phit size is 4 bytes.

The router description provides parameters to perform fine adjustment in the NoC properties, aiming at matching application requirements as well as possible. The cost-performance trade-offs can be explored by changing NoC parameters.

SoCIN can support other devices connected to the routers, besides processors. In spite of that, this work considered only processors connected through the network, using homogeneous ISA and private memory. Other research efforts in our research group have been conducted to use heterogeneous processors and shared memory, but they will not be discussed in this paper.

3. Outline of the MPSoC Middleware

This section presents the middleware proposed to fulfill the requirements of a real-time and embedded system with energy restrictions. An MPSoC is assumed as the target hardware platform.

Within the context of this work, the middleware aims at promoting software and hardware reuse and includes mechanisms that help to express real-time requirements and constraints. Those properties should be fulfilled having in mind limitations in physical resources like energy, memory, and processor performance.

The proposed architecture allows a flexible and broad design space exploration, by acting upon issues like

hardware or software implementation of services and objects and locality of objects in the network.

Figure 2 shows the proposed architecture, which is organized in two abstraction levels: structure and service levels. The structure level offers the more elementary resources of the middleware, namely network communication and multithread management. Using classical definitions, this level could be defined as an RTOS. However, to offer flexibility and enhance overall efficiency, RTOS-like capabilities are included in the middleware. The service level offers a higher abstraction and uses resources implemented at the structure level. It offers basic services, if one considers the complexity of a general purpose distributed system. However, these services are sufficient to support multiprocessor embedded application design, allowing the exploration of different arrangements in the allocation of tasks either at design time or at execution time.

It is important to highlight the monitoring and DVS (Dynamic Voltage Scaling) services, at the structure level. Those services are not part of the original RTSJ standard, but they were defined in the middleware to support some facilities at the service level.

This paper discusses the task migration and task allocation services in detail. A more generic view will be given for the other services.

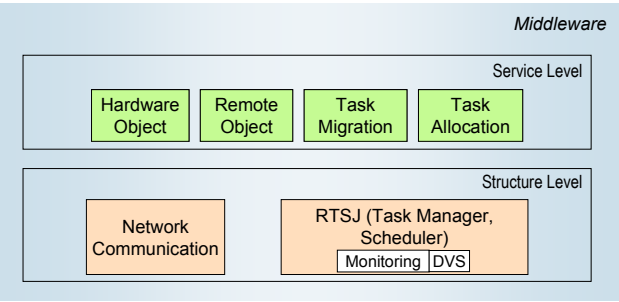


Figure 2: Middleware architecture

3.1 Real-time multithread management

In the context of this work, a thread is a synonym for a schedulable object and is also called task.

The Real-Time Specification for Java (RTSJ) standard [1] defines a set of interfaces and behavioral specifications to allow the development of real-time applications using the Java programming language. Among its major features are: scheduling properties suitable for real-time applications with provisions for periodic and sporadic tasks and support for deadlines and CPU time budgets.

RTSJ allows the use of schedulable objects, which are instances of classes that implement the so called Schedulable interface, such as `RealtimeThread`. It also specifies a set of classes to store parameters that represent a particular resource demand from one or more schedulable objects. For example, the `ReleaseParameters` class (superclass from `AperiodicParameters` and `PeriodicParameters`) includes several useful parameters for the specification of real-time requirements, such as cyclic activation and deadlines. Moreover, it supports the expression of the following elements: absolute and relative time values, timers, periodic and aperiodic tasks, and scheduling policies.

Along with the Java processor there is an API [13] that supports the specification of concurrent tasks and allows the specification of timing constraints, implementing a subset of the RTSJ standard.

The scheduling structure consists of an additional process that is in charge of allocating the CPU for those application-processes that are ready to execute, exactly like in an RTOS. Application developers should choose the most suitable scheduling algorithm at design time. Therefore, a customized scheduler is synthesized with the whole application into the embedded target system.

Currently, four scheduling algorithms are available: EDF, RM, Fixed Priority (software and hardware implementations), and Time-Triggered.

3.1.1 Additional functions to RTSJ

The so called function 'monitoring' aims at measuring resources of the local processor, like available memory and processor utilization. This function is offered to the task allocation service to help its decision when adding a new thread to a processor.

A DVS (Dynamic Voltage Scaling) functionality is added to the schedulers and allows the application to act upon the hardware for energy reduction purposes in a transparent way. The use of DVS algorithms, like the cycle-conserving one [15], opens space for energy reduction at execution time. By using a DVS capability, the scheduler can manage the local processor frequency to the lowest value able to match the deadlines of the threads added to the scheduler. From the designer's point-of-view, it is enough to use a scheduler that is able to manage DVS resources.

3.2 Network communication

The communication API (COM-API) encapsulates transport and datalink layers, providing an interface to the application layer [2].

The communication system provides support to message exchange among applications running in different

processors. The API allows applications to establish a communication channel through the network, which can be used to send and receive messages. The service allows the assignment of different priorities to messages and can run in a multithread environment. From the application point-of-view, the system is able to open and close connections as well as to send and receive messages, being accessed by different threads simultaneously.

The COM-API works together with the RTSJ-API, using processor features to provide communication via a network interface. RTSJ-API provides schedulable objects (for real-time threads) and relative time objects.

In order to offer a larger design space to be explored in the development of application-specific systems, a hardware implementation of the communication service was also developed [18]. It is encapsulated in a class called `HwTransport` and can be used in the same way as the software implementation (called `Transport`). The Java processor interacts with this communication block implemented in hardware as with any other I/O device.

The differences when using hardware and software implementations are transparent to the developer, since they are encapsulated in different classes that implement the same interface.

3.3 Locality abstraction

An important demand for design space exploration in an MPSoC system is to allow the allocation of threads everywhere in the network, making this locality transparent to the application until run-time. This property requires an abstract locality mechanism in order to allow access to other objects even when their location is unknown at development time. Moreover, this mechanism should be integrated with the RTSJ-API in order to offer temporal guarantees for message delivery.

A simplified mechanism for remote method invocation was proposed and implemented based on RMI from standard Java [17]. A conceptual modification was introduced in this mechanism using time bounds for its operations using RTSJ objects. A specific class to encapsulate real-time properties was added (`RealTimeParameters`) both in the client and in the server sides. The thread (`ConnectionHandler`) that deals with connections on the server side is another component to bring predictability to RMI. This thread has real-time properties following RTSJ rules. This means that it will be scheduled according to its real-time properties, like period and deadline. The RTSJ API allows the developer to choose among different scheduling policies, as already mentioned.

Similarly, a maximum execution time is defined for the `ConnectionHandler` thread at development time, using an asynchronous event mechanism, as defined by RTSJ. Thus, the communication operations will not violate the time reserved for the other application threads or tasks.

3.4 Hardware-object implementation

The boundary of the hardware/software partition plays an important role in meeting design constraints. This boundary is often decided upon at the early stages of development, leading to premature and inadequate design decisions. Moreover, it is hard to move this boundary at later stages. Better design decisions could be made at later stages in the development, when a better understanding of impacts of alternative hardware and software implementations emerges. This is only possible if the design process includes tools that simplify the movement of components' deployment from hardware to software and vice-versa, by defining a uniform programming model for both implementations.

Within the context of this work, a real-time thread can be implemented in two different ways. A software implementation is a Java code executed by the processor, as described in [13]. A hardware implementation executes autonomously, although controlled by the processor. A hardware thread has its own Finite State Machine (FSM) and can run in parallel with the processor.

A hardware component (HwTI – Hardware Thread Interface) is defined as an interface between the processor and the hardware thread. Another hardware component must implement the thread behavior and is called Hardware Thread Behavior (HwTB). HwTI is part of the platform, available to developers, while HwTB is part of the application and must be implemented by developers using a hardware description language. The proposed architecture is introduced in [14], where it is better described.

The communication between the application and the HwTB component is managed in software, by an RTSJ compatible class.

From the software point-of-view, the hardware thread is encapsulated by an object that extends the `RealtimeThread` class from RTSJ. So, the hardware thread will be controlled similarly to other threads implemented in software, by reusing schedulers already available in the RTSJ implementation.

3.5 Energy management

An important demand for MPSoC platforms is energy management, since most of them are powered by batteries. Low energy means a smaller battery, lower weight, lower cost, and so on.

Within the context of this work, low power and low energy are provided by hardware implemented objects that can be included as services or as application components. To reach flexibility, a DVS/DFS (Dynamic Voltage Scaling / Dynamic Frequency Scaling) functionality is included in the task schedulers and exposed to be selected by application developers. The application can define the scheduler to be used in each processor, thus defining if DVS should be used or not.

4. Task allocation and migration

Task allocation and migration are services related to load balancing, and a homogeneous ISA is required. A task is allocated before it starts running and can be migrated during its execution.

4.1 Task migration

Dynamic task allocation has been shown to be a promising technique to obtain load balancing among processing units in an MPSoC [6] [7], allowing the minimization of some variables, like execution time or power consumption. To reach dynamic allocation, a migration task mechanism is required. Two cases are possible: (1) when a new allocation is required in a set of not-empty processors, some tasks could be moved to optimize the new distribution; (2) when a set of tasks is finished, a new arrangement can be made to optimize the overall processors' utilization.

Task-migration approaches usually adopt shared-memory as the communication model in an SMP (symmetrical multi-processing) environment. This work considers an AMP (asymmetrical multi-processing) model, since processors have dedicated local memory resources. Although processors can have different organizations, like pipelines and multi-cycle ones, they share a common ISA, and, thus, tasks can be assigned to different processors. For the adopted platform (a NoC), message exchange is a natural choice due to its scalability. However, a shared-memory model is also under investigation as a communication strategy, but it is not in the scope of this paper.

To offer task migration as a service in the middleware, all communication operations should be submitted to the communication API. Moreover, these communications are submitted to the discipline of a periodic real-time thread with a pre-established maximum cost. These properties make the task migration service independent from the underlying network and adequate to be used in real-time applications.

In the context of the adopted platform, a task is not built at run-time, but it is defined at development time, together with its accessed objects. Thus, its address space is known *a priori*. The middleware can only move quite independent tasks. Currently, if different tasks share the same data, the application is supposed to take care of data coherence after migration. Some mechanisms to solve this situation are still under investigation, since they introduce an important overhead in the communication.

Figure 3 shows the class diagram of the implemented service. The migration service is activated by another service, called task allocation, which decides which task to move and its destination, based on restrictions, like processor or memory utilization, and on objectives, like load balancing.

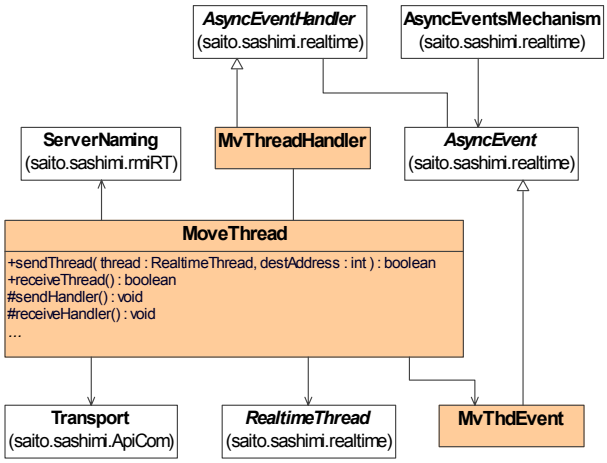


Figure 3: Task migration class diagram

The `MoveThread` class contains the public methods `sendThread()` and `receiveThread()`. They are used to activate task sending and -receiving services. They both return FALSE if the service is not available. When the `sendThread()` method is executed, an event handler (`MvThreadHandler`) is executed and configured to move the task, which is passed as a parameter. The task is sliced into blocks, and the first one is sent. After that, each time an ACK is received (from the receiver), an event is generated and the next block is sent. The event handler follows the asynchronous event management policy, defined by `AsyncEventMechanism`. Although this procedure leads to an increase in the latency of a migration, it ensures a balanced use of the processor, avoiding any interference in other running real-time tasks.

Task migration means to send code (methods) and attributes of the `RealtimeThread` object as well as the

objects referred to in the `RealtimeThread`. The stack is also sent. Being a stack machine, Java preserves task variables in the stack, such that the task context is replicated when the stack is copied. This property makes context copy easy, avoiding the use of checkpoints. In other words, the memory used by the task is confined to the attributes of its classes and to the stack, which contains method variables.

For the adopted platform, the position of objects (code and attributes) in the memory is defined by a post-compilation tool, which can set appropriate attributes of the `RealtimeThread` class. The stack position and size are known by the `RealtimeThread`. The `MoveThread` class obtains those values at run-time before moving the task.

The migration service should be activated in the destination too, as in the origin of the migration, by invoking the `receiveThread()` method.

4.2 Task allocation

The task allocation service consists in pointing out nodes for tasks in the network using a distribution function. In [6] and [7] different distribution algorithms were investigated and some solutions were proposed. Those algorithms have been firstly evaluated in a high abstraction level simulator and afterwards implemented as part of the middleware. The role of the middleware is to offer an interface to the service, thus making easy for the application developer the choice of a distribution algorithm.

Each node of the network should have an instance of the monitoring service (middleware structure level), which is able to inform about the availability of resources (memory, processor time).

Figure 4 shows the class diagram for the task allocation service. First of all, the class `RealtimeThread`, from RTSJ, is extended, thus creating an `XtdRealtimeThread` class. This new class has the properties the task should inform to the allocation service, as memory and processor utilization. In fact, for a periodic task, it is possible to obtain the processor utilization by referring to RTSJ parameters, since a periodic `RealtimeThread` knows its worst case execution time (WCET) and the period as well. The utilization is equal to the WCET divided by the period.

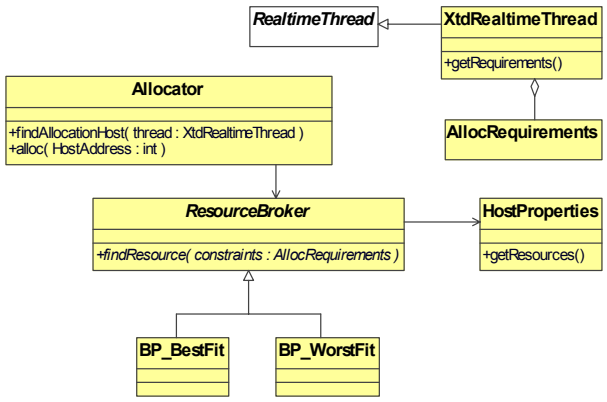


Figure 4: Task allocation class diagram

Task distribution is implemented by the classes `Allocator` and `ResourceBroker`. The design pattern Strategy is used to offer abstract access to different allocation algorithms. In the diagram provided in Figure 4, Bin-Packing Best Fit (BP_BestFit) and Bin-Packing Worst Fit (BP_WorstFit) are shown to illustrate possible algorithms, as proposed in [6]. The `findResource()` method is implemented in the concrete classes to perform a search for a node to allocate a task.

5. EXPERIMENTAL RESULTS

For experimental verification, a SystemC simulator uses an RTL description of the FemtoJava processor. The network is implemented as a TLM (transaction-level) model.

The example presented in this paper is a demonstration of the task migration service. In this example, three synthetic tasks are executed in one processor and one task in another one. This example represents four different applications that do not have communication between them. The tasks are periodic and the migration should not jeopardize their deadlines. After some time, one task (`TaskC`) migrates from the first processor to the second one.

The `AsyncEventMechanism` period was chosen such that the task that migrates (`TaskC`) could do it between two consecutive execution periods. Figure 5 shows the activation times for `TaskC`, where the x-axis represents time in milliseconds. The first two executions occur at the origin processor, while the remaining ones occur at the destination. The third execution experiences latency due to the migration time. One can see that the task promptly recovers its original period (30 ms), as started in the origin. The activation time of a `RealtimeThread` is part of its attributes and is copied in the migration process.

Thus, the scheduler in the destination can keep the original behavior of the task.

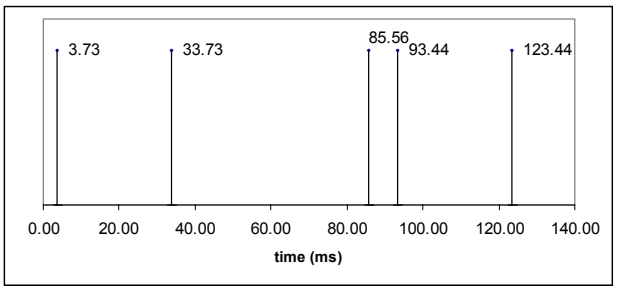


Figure 5: Activation time for a migrated task

The time required to migrate a task can be evaluated from two different points-of-view, as shown in Table 1. The first line shows the computational cost to migrate `TaskC`, it means, the real cost in processing the migration service. The cost grows linearly as the task size increases. The time values depend on the latencies imposed by the communication service, provided by the structure layer. This throughput can be optimized using a hardware-implemented communication service [18] or a processor with higher performance [9].

The migration does not occur in a continuous flow, which could compromise the deadline of other tasks running. On the contrary, the transmission is sliced in blocks controlled by the `AsyncEventMechanism` from RTSJ. It increases total latency observed by the user of the service, as shown in the second line in Table 1. In the origin, it is the time elapsed since the `sendThread()` method is invoked until the service finishes. In the destination, it is the time elapsed since the first block starts to arrive until the `start()` of the `RealtimeThread` in the destination processor. At both sides (origin and destination), the end of the service is transparent to the user, i.e., the methods that activate the service do not retain the flow of the code that invokes them. The total latency grows following the period of the task that implements the `AsyncEventMechanism`.

Table 1: Time measures for task migration

	Origin node	Destination node
Effective cost (ms)	3.27	3.93
Total latency (ms)	53.30	51.42

Using the middleware, developers save development time required to implement capabilities already provided as services. Code provided to implement HW-SW communication, task migration, remote method invocation, and so on can be reused in all projects. Table 2 shows the amount of memory used by some services of the middleware compared with a classical embedded application, an MP3 player. The table shows that the total memory consumed by the middleware is acceptable for real applications.

Table 2: Memory usage

***Middleware* component**	**ROM**	**RAM**
Remote method (server)	2139 Bytes	118 Bytes
Task migration (origin)	2343 Bytes	81 Bytes
COM-API (Pack49-Msg500)	4493 Bytes	6345 Bytes
API-RTSJ + DVS	4849 Bytes	242 Bytes
TOTAL (*middleware*)	13824 Bytes	6786 Bytes
Application	**ROM**	**RAM**
Mp3Player	48548 Bytes	63702 Bytes

6. CONCLUSIONS

Multiprocessor platforms bring new challenges to the development of applications with high quality, matching real-time requirements and keeping a low energy usage. This paper proposes to face this challenge using a middleware to abstract platform details and allowing developers to express real-time requirements.

An MPSoC with homogeneous ISA is considered for task migration and allocation services.

Preliminary results on task migration are presented and evaluated. Results show that this service presents an acceptable cost and offers an adequate abstraction for the application developer. This service runs upon the structure level of the middleware, which is similar to an RTOS.

The next step of this work is to validate and evaluate the task allocation service, based on algorithms previously evaluated in [6].

REFERENCES

[1] Bollella, G.; Gosling, J.; Brosgol, B. The Real-Time Specification for Java. 2001. <http://www.rtj.org/rtsj-V1.0.pdf>.

[2] Martin, G. *Overview of the MPSoC Design Challenge*. In: Design Automation Conference, DAC, 2006, San Francisco, p. 274-279.

[3] Jerraya, A.A.; Bouchhima, A. and Pétrot, F. *Programming models and HW-SW Interfaces Abstraction for Multi-Processor SoC*. In: Design Automation Conference, DAC, 2006, San Francisco, p. 280-285.

[4] Bernstein, P.A. *Middleware: A Model for Distributed System Services*. Communications of the ACM, New York, vol.3, n.2, p.86-97, Feb. 1996.

[5] Schmidt, D.C. *Middleware Techniques and Optimizations for Realtime, Embedded Systems*. In: International Symposium on System Synthesis, 1999, San Jose, CA, p. 12-16.

[6] Wronski, F.; Brião, E.W.; Wagner, F.R. *Evaluating Energy-aware Task Allocation Strategies for MPSoCs*. In: IFIP TC-10 Working Conference on Distributed and Parallel Embedded Systems, DIPES, 2006, Braga, p. 215-224.

[7] Acquaviva, A. et al. *Assessing Task Migration Impact on Embedded Soft Real-Time Streaming Multimedia Applications*. EURASIP Journal on Embedded Systems, New York, v. 2008, n.2, p.1-15, Apr. 2008.

[8] Ito, S.A., Carro, L., Jacobi, R.P. *Making Java Work for Microcontroller Applications*. IEEE Design & Test of Computers, v. 18, n. 5, p. 100-110, Sept/Oct. 2001.

[9] Beck Filho, A.C.S.; Carro, L. *Low Power Java Processor for Embedded Applications*. In: IFIP VLSI-SOC, 2003, Darmstadt, p. 239-244.

[10] Benini, L.; Demicheli, G. *Networks on Chip: A New SoC Paradigm*. IEEE Computer, v.35, n.1, p. 490-504, Jan. 2002.

[11] Zeferino, C.A.; Susin, A.A. *SoCIN: A parametric and scalable network-on-chip*. In: Symposium on Integrated Circuits and Systems Design, SBCCI, 2003. Los Alamitos: IEEE Computer, 2003. p. 169-174.

[12] Silva Jr., E.T.; Freitas, E.P.; Wagner, F.R.; Carvalho, F.C.; Pereira, C.E. *Java Framework for Distributed Real-Time Embedded Systems*, In: 9th IEEE ISORC, Gyeongju, Korea, 2006, p. 85-92.

[13] Wehrmeister, M.A.; Becker, L.B. and Pereira, C.E. *Optimizing Real-Time Embedded Systems Development Using a RTSJ-based API*. In: JTRES 2004, Proceedings Springer LNCS, Cyprus, October 2004, p. 292-302.

[14] Silva Jr., E.T.; Andrews, D.; Pereira, C.E. and Wagner, F.R. *An Infrastructure for Hardware-Software Co-design of Embedded Real-Time Java Applications*. In: 11th IEEE ISORC, Orlando, USA, 2008.

[15] Pillai, P. and Shin, K.G. *Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems*. In Proc. of the 18th ACM Symp. on Operating Systems Principles, 2001, p. 89-102.

[16] Spuri, M. and Butazzo, G. *Efficiente aperiodic service under earliest deadline scheduling*. In: IEEE Real-Time Systems Symposium, RTSS, 1994.

[17] Grosso, W. *Java RMI*. O'Reilly Media, 2001.572 p.

[18] Silva Jr, E.T.; Wagner, F.R.; Freitas, E.P.; Kunz, L. and Pereira, C.E. *Hardware Support in a Middleware for Distributed and Real-Time Embedded Applications*. Journal of Integrated Circuits and Systems, v. 2, n.1, p. 38-44, Mar. 2007.

Scheduling as a Learned Art*

Christopher Gill, William D. Smart, Terry Tidwell, and Robert Glaubius

Department of Computer Science and Engineering
Washington University, St. Louis, MO, USA
{cdgill, wds, ttidwell, rlg1}@cse.wustl.edu

Abstract. Scheduling the execution of multiple concurrent tasks on shared resources such as CPUs and network links is essential to ensuring the reliable and correct operation of real-time systems. For closed hard real-time systems in which task sets and the dependences among them are known *a priori*, existing real-time scheduling techniques can offer rigorous timing and preemption guarantees. However, for open soft-real-time systems in which task sets and dependences may vary or may not be known *a priori* and for which we would still like assurance of real-time behavior, new scheduling techniques are needed.

Our recent work has shown that modeling non-preemptive resource sharing between threads as a Markov Decision Process (MDP) produces (1) an analyzable utilization state space, and (2) a representation of a scheduling decision policy based on the MDP, even when task execution times are loosened from exact values to known distributions within which the execution times may vary. However, if dependences among tasks, or the distributions of their execution times are not known, then how to obtain the appropriate MDP remains an open problem.

In this paper, we posit that this problem can be addressed by applying focused reinforcement learning techniques. In doing so, our goal is to overcome a lack of knowledge about system tasks by observing their states (e.g., task resource utilizations) and their actions (e.g., which tasks are scheduled), and comparing the transitions among states under different actions to obtain models of system behavior through which to analyze and enforce desired system properties.

1 Introduction

Scheduling the execution of multiple concurrent tasks on shared resources such as CPUs and network links is essential to ensuring the reliable and correct operation of real-time systems. For closed hard real-time embedded systems in which the characteristics of the tasks the system must run, and the dependences among the tasks are well known *a priori*, existing real-time scheduling techniques can offer rigorous timing and preemption guarantees.

* This research was supported in part by NSF grant CNS-0716764 (Cybertrust) titled "CT-ISG: Collaborative Research: Non-bypassable Kernel Services for Execution Security" and NSF grant CCF-0448562 (CAREER), titled "Time and Event Based System Software Construction".

However, maintaining or even achieving such assurance in open soft real-time systems that must operate with differing degrees of autonomy in unknown or unpredictable environments, remains a significant open research problem. Specifically, in open soft-real-time domains such as semi-autonomous robotics, the sets of tasks a system needs to run (e.g., in response to features of the environment) and the dependences among those tasks (e.g., due to different modes of operation triggered by a remote human operator) may vary at run-time.

Our recent work [1] has investigated how modeling interleaved resource utilization by different threads as a Markov Decision Process (MDP) can be used to analyze utilization properties of a scheduling decision policy based on the MDP, even when task execution times are loosened from exact values to known distributions within which their execution times may vary. However, if we do not know the distributions of task execution times, or any dependences among tasks that may constrain their inter-leavings, then how to obtain the appropriate MDP remains an open problem.

In this paper, we discuss that problem in the context of open soft real-time systems such as semi-autonomous robots. Specifically, we consider how limitations on the observability of system states interacts with other concerns in these systems, such as how to handle transmission delays in receiving commands from remote human operators, and other forms of *operator neglect*. These problems in turn motivate the use of learning techniques to establish and maintain appropriate timing and preemption guarantees in these systems. Section 2 first surveys other work related to the topics of this paper. Sections 3 and 4 then discuss the problems of limited state observability and operator neglect, respectively, for these systems. In Section 5 we postulate that dynamic programming in general, and focused reinforcement learning based on realistic system limitations in particular, can be used to identify appropriate MDPs upon which to base system scheduling policies that enforce appropriate timing and preemption guarantees for each individual system. Finally, in Section 6 we summarize the topics presented in this paper, and describe planned future work on those topics.

2 Related Work

A variety of thread scheduling policies can be used to ensure feasible resource use in closed real-time systems with different kinds of task sets [2]. Most of those approaches assume that the number of tasks accessing system resources, and their invocation rates and execution times, are all well characterized. Hierarchical scheduling techniques [3–6] allow an even wider range of scheduling policies to be configured and enforced, though additional analysis techniques [1] may be needed to ensure real-time properties of certain policies.

Dynamic programming is a well-proven technique for job shop scheduling [7]. However, dynamic programming can only be applied directly when a complete model of the tasks in the system is known. When a model presumably exists but is not yet known, *reinforcement learning* [8] (also known as approximate dynamic programming) can instead offer iteratively improving approximations of an optimal solution, as has been shown in several computing problem domains [9–11]. In this paper we focus on a particular variant of reinforcement learning in which convergence of the approximations towards optimal is promoted by restricting the space of learning according to realistic constraints induced by the particular scheduling problem and system model being considered.

3 Uncertainty, Observability, and Latency

Our previous work on scheduling the kinds of systems that are the focus of this paper [1] considered only a very basic system model, in which multiple threads of execution are scheduled non-preemptively on a single CPU, and the durations of threads execution intervals fall within known, bounded distributions. For such simple systems, it was possible to exactly characterize uncertainty about the results of scheduling decisions in order to obtain effective scheduling policies. As we scale this approach to larger, more complicated systems, such as cyber-physical systems, we will need to address a number of sources of uncertainty, including variability in task execution intervals, partial observability of system state, and communication latency. In this section we define these terms, and outline the challenges that they present.

3.1 Uncertainty

Our previous work on scheduling the kinds of systems that are the focus of this paper [1] considered only a very low-level and basic system model. In this model multiple threads of execution are scheduled non-preemptively on a single CPU. The durations of the threads' execution intervals are drawn from known, bounded probability distributions. However, even in this simple setting, the variability of execution interval duration for a given thread means that the exact resource utilization state of each thread can only be accurately measured *after* a scheduling decision is implemented and the thread cedes control of the CPU. This means that our scheduling decisions must be made based on estimates of likely resource usage for the threads, informed by our knowledge of the probability density functions that govern their execution interval lengths.

This kind of uncertainty is the norm rather than the exception in many semi- or fully-autonomous real-time systems where responses to the environment trigger different kinds of tasks (e.g., a robot exploring an unfamiliar building may engage different combinations of sensors during wall following maneuvers). Our previous work [1] has shown that construction of an MDP over a suitable abstraction of the system state is an effective way to perform this stochastic planning. Our knowledge of task duration distributions can be embedded into an MDP model; we can then use well-established techniques to to formulate suitable scheduling strategies in which desired properties such as bounded sharing of resources are enforced rigorously.

In order to scale this approach to larger, more complicated systems, it is necessary to cope with a greater degree of uncertainty about the outcomes of scheduling decisions. As systems increase in size and complexity, and particularly when the system interacts with other systems or the real world through communication or sensors and actuators, uncertainty about system activities' resource utilization and progress will grow. In conjunction with this increase in complexity, we are decreasingly likely to be able to provide good models of this uncertainty in advance. Instead, it will be necessary to discover and model it empirically during execution. Our current approach can be extended to cover this situation by iteratively estimating these models, and designing scheduling policies based on these models. However, explicitly constructing these models may be unnecessary, as techniques exist for obtaining asymptotically optimal policies directly from experience [12].

3.2 Partial Observability

Much as variability in execution times limits the ability to predict the consequences of actions, in many important semi-autonomous systems it also may not be possible to know even current system states exactly. Often, it will be the case that our measurements of resource utilization are noisy, and the actual values must be inferred from from other data. A high-level example of this is determining the location of a mobile robot indoors. In such settings, there often is no "position sensor" that can be used to provide the exact location of the robot.[1] Instead, we must use other sensors to measure the distances to objects with known positions, correlate these with a pre-supplied map, and calculate likely positions. Because of measurement error in these sensors, imperfect maps, and self-similarities in the environment, this can often lead to multiple very different positions being equally likely.

[1] Outdoors, GPS receivers may get close to being such sensors, but their signals cannot penetrate buildings and even some outdoor terrain features reliably.

In such a situation the system's state (e.g., location in the robotics example) is said to be *partially observable*, and is characterized by the presence of system state variables that are not directly observable: there is some process that makes observations of these state variables, but there may be many different observations corresponding to any particular value of the state variable.

Partially observable systems are naturally modeled by an extension of MDPs, called Partially Observable MDPS, or POMDPs [13]. Control policies for POMDPs can be derived by a reduction to a fully observable MDP by reasoning about *belief states*. In short, given a POMDP we can construct a continuous-state MDP in which each state is a probability distribution over the states of the POMDP, corresponding to the belief that the system is in a particular configuration of the original POMDP. The state of this new MDP evolves according to models of state and observation evolution in the POMDP. Since states in this reduced MDP model correspond to distributions over system states in the original partially observable system, the MDP state space is quite large. It will be necessary to make extensive abstraction of the original problem in order to efficiently derive effective scheduling policies in such cases.

3.3 Observation Lag

A further complication is that state observations may incur temporal delays. For example, even if a robot could measure its position exactly, the environment may transition through a number of states while the robot is making that measurement. The effectiveness and safety of collision avoidance and other essential activities thus may be limited by delays in state observation and action enactment, and thus must be implemented and scheduled with such delays in mind. In our previous work, we addressed task execution interval length by explicitly encoding time into the system state; however, as systems grow larger and more abstract such an approach is likely to result in intractably large state spaces.

As with the case of partial observability of state, there is an extension to the theory of Markov decision processes that addresses these situations. The resulting system is called a Semi-Markov decision process, or SMDP [14]. In an SMDP the controller observes the current system state and issues a decision that executes for some stochastic interval. During this execution, the system state may change a number of times. Once the previous decision terminates, the control policy may make another decision. In the robotics example above, the controller decides to poll the position sensor; meanwhile, the system continues on some trajectory through the state space. Once the system is done polling the position sensor, it then makes another decision based on its current belief state. Methods for finding optimal solutions for MDPs have been extended to the SMDP case.

4 Neglect Tolerance

Although we are currently focused on thread-scheduling and other low-level phenomena, the general class of problems in which we are interested extends up to larger, more integrative systems. In particular, we are interested in problems involving scheduling of whole system behaviors, where the state space is much larger and more complex, and where the system is interacting with the physical world. The canonical example of such a system is an autonomous mobile robot capable of performing several, often conflicting, behaviors. The robot must schedule these behaviors appropriately to achieve some high-level task, while keeping itself (and potentially people around it) safe.

Behaviors must be scheduled and sequenced to avoid conflicts while attempting to optimize multiple criteria such as task completion time and battery life. This is a real-time systems problem, although it is performed at time-scales much longer than usually considered in the real-time systems research literature. The robot's sensors, actuators, and computational resources are shared. Behaviors must often complete by some deadline or at a certain frequency to avoid disaster. For example, to avoid obstacles, the proximity sensors must be polled at a certain rate, to allow the robot to take actions in time to avoid a collision. To make matters worse, these deadlines are often state-dependent: the faster a robot moves, the more frequently it must poll its sensors.

Robot systems also often have (potentially hard) deadlines on the execution of single actions. For example, consider a robot driving up to an intersection. There is a critical time period during which it must either stop or make a turn to avoid crashing into a wall. In the field of Human-Robot Interaction, when the human directly tele-operates the robot, and essentially acts as the behavior scheduling agent, this problem is closely tied to the idea of *neglect tolerance* [15]. This is a measure of the ill effects of failing to meet a timing deadline. Systems with a low neglect tolerance must be constantly monitored and guided by the human operator. Systems with a high neglect tolerance can be ignored for much of the time without catastrophic effects.

The systems that we describe in this section suffer from all of the problems we described above: uncertainty, observability, and latency. They also have much larger state and action spaces, are less well understood, are much harder to capture with formalized models in any tractable way, and have stochasticity that is likely hard to model parametrically. In our previous research, scheduling experts and machine learning experts have needed to spend a lot of time together, crafting the formalization of the problem, and examining the solutions obtained. This interaction between domain experts and machine learning specialists will become even more important as we scale to larger systems. In

particular, the large, often ill-defined state spaces of these problems must be mapped into manageable representations over which optimization and learning techniques will work well. This often requires deep and specific insights into the problem domain, coupled with equally deep insights into what representations are likely to work well in practice.

There is a direct connection between the concepts of neglect tolerance and real-time scheduling. Both require guarantees of execution time: the latter in the completion of a task, and the former in the reception of a control input from a human operator. The time-scale of the robot control problem, however, is several orders of magnitude larger than those typically considered in many real-time systems. It is also a dynamic and situational deadline: the appropriate timing of the input depends critically on the features of the environment in which the robot finds itself and on its own internal parameters, such as speed limits. This means that it is extremely hard to model and analyze these concerns using traditional techniques from real-times systems theory.

Our work thus far has focused on problems in which the scheduling decision maker is the only active agent. Tasks under scheduler control may behave stochastically, but their behavior is believed to be consistent with a model that depends on a small number of parameters. Incorporating a human or other adaptive agent into the schedulers environment represents a significant new extension of that direction, as evidenced by the field of multiagent systems. Formal guarantees in the theory of Markov decision processes break down in these settings, because it is unlikely that a human decision maker will follow a sufficiently consistent (and stationary) policy. For example, if we train an agent to interact with one operator, the learned policy is unlikely to be optimal for another operator who may be more or less prone to different kinds and gradations of neglect. For these reasons, we intend to focus our future work on the issues mentioned in Section 3 in the single agent case, but with an eye towards extending eventually into multiagent settings.

5 Learning

Scheduling decisions in our approach are based on a *value function*, which captures a notion of long-term utility. Specifically, we use a *state-action value function*, Q, of the form

$$Q\left(s,a\right) = R\left(s,a\right) + \gamma \sum_{s'} \left[P^{a}_{s,s'} \max_{a'} Q\left(s',a'\right) \right].$$

$Q(s,a)$ gives the expected long-term value of taking action a from state s, where $R(s,a)$ is the reward received on taking action a from state s, and $P^{a}_{s,s'}$ is the

probability of transitioning from state s to state s' on action a. Given this value function, the control policy is easy to compute: $\pi(s) = \arg\max_a Q(s, a)$. If we know both the transition function and the model, then we can solve for the value function directly [14], using techniques from dynamic programming.

Identifying complete distributions of task times and inter-task dependencies in real-world systems is a daunting task to begin with, and in some open real-time systems doing so *a priori* may not be possible due to varying modes of operation at run-time. To address this problem, we are investigating how to use *reinforcement learning* (RL) in developing effective thread scheduling policies, which can be encoded and enforced easily and efficiently.

Whereas dynamic programming assumes all models are provided in advance, RL is a stochastic variant of dynamic programming in which models are learned through observation. In RL, control decisions are learned from direct experiences [16, 8]. Time is divided into discrete steps and at each time step, t, the system is in one of a discrete set of states, $s_t \in S$. The scheduler observes this state, and selects one of a finite set of actions, $a_t \in A$. Executing this action changes the state of the system on the next time step to $s_{t+1} \in S$, and the scheduler receives a *reward* $r_{t+1} \in \mathbb{R}$, reflecting how good it was to take the action a_t from state s_t in a very immediate sense. The distribution of possible next states is specified by the *transition function*, $T : S \times A \to \Pi(S)$, where $\Pi(S)$ denotes the space of probability distributions over states. The rewards are given by the *reward function*, $R : S \times A \to \mathbb{R}$. The resulting model is exactly a Markov Decision Process (MDP) [14].

If either the transition function or the value function is unknown, we must resort to reinforcement learning techniques to estimate the value function. In particular, well-known algorithms exist for iteratively calculating the value function in the case of discrete states and actions, based on observed experiences [17–19].

6 Conclusions and Future Work

In this paper we have presented an approach that uses focused reinforcement learning to address important open challenges in scheduling open soft real-time systems such as semi-autonomous robots. We have discussed how different forms of state observability limitations and operator neglect can affect how well the system state can be characterized, and have postulated that reinforcement learning can obtain approximate but suitable models of system behavior through which appropriate scheduling can be performed.

Throughout this paper, we have focused mainly on practical problems in the domain of semi-autonomous real-time systems. In particular, both physi-

cal limits and policy restrictions help to narrow the space in which learning is performed, and thus help to focus the learning techniques for more rapid convergence from feasible solutions towards optimal ones. Our near-term future work will focus on how particular combinations of state observability and different time scales of operator interaction and neglect induce different concrete problems to which different configurations of focused reinforcement learning can be applied. The results of these investigations are likely to have impacts outside the particular class of systems we are considering (e.g., to open systems more generally), and to other problem domains (e.g., for protection against denial of service attacks or quality-of-service failures, which is the domain from which this research emerged).

References

1. Tidwell, T., Glaubius, R., Gill, C., Smart, W.D.: Scheduling for reliable execution in autonomic systems. In: Proceedings of the 5th International Conference on Autonomic and Trusted Computing (ATC-08), Oslo, Norway (2008)
2. Liu, J.W.S.: Real-time Systems. Prentice Hall, New Jersey (2000)
3. Goyal, Guo, Vin: A Hierarchical CPU Scheduler for Multimedia Operating Systems. In: 2^{nd} Symposium on Operating Systems Design and Implementation, USENIX (1996)
4. Regehr, Stankovic: HLS: A Framework for Composing Soft Real-time Schedulers. In: 22^{nd} IEEE Real-time Systems Symposium, London, UK (2001)
5. Regehr, Reid, Webb, Parker, Lepreau: Evolving Real-time Systems Using Hierarchical Scheduling and Concurrency Analysis. In: 24^{th} IEEE Real-time Systems Symposium, Cancun, Mexico (2003)
6. Aswathanarayana, T., Subramonian, V., Niehaus, D., Gill, C.: Design and performance of configurable endsystem scheduling mechanisms. In: Proceedings of 11th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS). (2005)
7. Held, M., Karp, R.M.: A dynamic programming approach to sequencing problems. Journal of the Society for Industrial and Applied Mathematics **10**(1) (1962) 196–210
8. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. Adaptive Computations and Machine Learning. The MIT Press, Cambridge, MA (1998)
9. Tesauro, G., Jong, N.K., Das, R., Bennani, M.N.: On the use of hybrid reinforcement learning for autonomic resource allocation. Cluster Computing **10**(3) (2007)
10. Littman, M.L., Ravi, N., Fenson, E., Howard, R.: Reinforcement learning for autonomic network repair. In: Proceedings of the 1st International Conference on Autonomic Computing (ICAC 2004). (2004) 284–285
11. Whiteson, S., Stone, P.: Adaptive job routing and scheduling. Engineering Applications of Artificial Intelligence **17**(7) (2004) 855–69
12. Watkins, C.J.C.H., Dayan, P.: Q-learning. Machine Learning **8**(3-4) (1992) 279–292
13. Kaelbling, L.P., Littman, M.L., Cassandra, A.R.: Planning and acting in partially observable stochastic domains. Artificial Intelligence **101**(1–2) (1998) 99–134
14. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley Interscience (1994)
15. Crandall, J.W., L., C.M.: Developing performance metrics for the supervisory control of multiple robots. In: Proceedings of the ACM/IEEE International Conference on Human-Robot Interaction (HRI '07). (2007) 33–40

16. Kaelbling, L.P., Littman, M.L., Moore, A.W.: Reinforcement learning: A survey. Journal of Artificial Intelligence Research **4** (1996) 237–285
17. Rummery, G.A., Niranjan, M.: On-line Q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR 166, Engineering Department, Cambridge University (1994)
18. Sutton, R.S.: Learning to predict by the method of temporal differences. Machine Learning **3** (1988) 9–44
19. Watkins, C.J.C.H., Dayan, P.: Q-learning. Machine Learning **8** (1992) 279–292

Towards Hierarchical Scheduling on top of VxWorks*

Moris Behnam†, Thomas Nolte, Insik Shin, Mikael Åsberg
MRTC/Mälardalen University
P.O. Box 883, SE-721 23 Västerås
Sweden

Reinder J. Bril
Technische Universiteit Eindhoven (TU/e)
Den Dolech 2, 5612 AZ Eindhoven
The Netherlands

Abstract

Over the years, we have worked on hierarchical scheduling frameworks from a theoretical point of view. In this paper we present our initial results of the implementation of our hierarchical scheduling framework in a commercial operating system VxWorks. The purpose of the implementation is twofold: (1) we would like to demonstrate feasibility of its implementation in a commercial operating system, without having to modify the kernel source code, and (2) we would like to present detailed figures of various key properties with respect to the overhead of the implementation. During the implementation of the hierarchical scheduler, we have also developed a number of simple task schedulers. We present details of the implementation of Rate-Monotonic (RM) and Earliest Deadline First (EDF) schedulers. Finally, we present the design of our hierarchical scheduling framework, and we discuss our current status in the project.

1 Introduction

Correctness of today's embedded software systems generally relies not only on functional correctness, but also on extra-functional correctness, such as satisfying timing constraints. System development (including software development) can be substantially facilitated if (1) the system can be decomposed into a number of parts such that parts are developed and validated in isolation and (2) the correctness of the system can be established by composing the correctness of the individual parts. For large-scale embedded real-time systems, in particular, advanced methodologies and techniques are required for temporal isolation all through design, development, and analysis.

Hierarchical scheduling has shown to be a useful mechanism in supporting modularity of real-time software by providing temporal partitioning among applications. In hierarchical scheduling, a system can be hierarchically divided into a number of subsystems that are scheduled by a global (system-level) scheduler. Each subsystem contains a set of tasks that are scheduled by a local (subsystem-level) scheduler. The Hierarchical Scheduling Framework (HSF) allows for a subsystem to be developed and analyzed in isolation, with its own local scheduler, and then at a later stage, using an arbitrary global scheduler, it allows for the integration of multiple subsystems without violating the temporal properties of the individual subsystems analyzed in isolation. The integration involves a system-level schedulability test, verifying that all extra-functional (including timing) requirements are met. Hence, hierarchical scheduling frameworks naturally support *concurrent development* of subsystems. Our overall goal is to make hierarchical scheduling a cost-efficient approach applicable for a wide domain of applications, including automotive, automation, aerospace and consumer electronics.

Over the years, there has been a growing attention to HSFs for real-time systems. Since a two-level HSF [9] has been introduced for open environments, many studies have been proposed for its schedulability analysis of HSFs [14, 17]. Various processor models, such as bounded-delay [20] and periodic [23], have been proposed for multi-level HSFs, and schedulability analysis techniques have been developed for the proposed processor models [1, 7, 11, 18, 22, 23, 24]. Recent studies have been introduced for supporting logical resource sharing in HSFs [3, 8, 12].

Up until now, those studies have worked on various aspects of HSFs from a theoretical point of view. This paper presents our work towards a full implementation of a hier-

*The work in this paper is supported by the Swedish Foundation for Strategic Research (SSF), via the research programme PROGRESS.

†Contact author: moris.behnam@mdh.se

archical scheduling framework, and we have chosen to implement it in a commercial operating system already used by several of our industrial partners. We selected the VxWorks operating system, since there is plenty of industrial embedded software available, which can run in the hierarchical scheduling framework.

The outline of this paper is as follows: Section 2 presents related work on implementations of schedulers. Section 3 present our system model. Section 4 gives an overview of VxWorks, including how it supports the implementation of arbitrary schedulers. Section 5 presents our scheduler for VxWorks, including the implementation of Rate Monotonic (RM) and Earliest Deadline First (EDF) schedulers. Section 6 presents the design, implementation and evaluation of the hierarchical scheduler, and finally Section 7 summarizes the paper.

2 Related work

Looking at related work, recently a few works have implemented different schedulers in commercial real-time operating systems, where it is not feasible to implement the scheduler directly inside the kernel (as the kernel source code is not available). Also, some work related to efficient implementations of schedulers are outlined.

Buttazzo and Gai [4] present an implementation of the EDF scheduler for the ERIKA Enterprise kernel [10]. The paper discusses the effect of time representation on the efficiency of the scheduler and the required storage. They use the Implicit Circular Timer's Overflow Handler (ICTOH) algorithm which allows for an efficient representation of absolute deadlines in a circular time model.

Diederichs and Margull [6] presents an EDF scheduler plug in for OSEK/VDX based real-time operating systems, widely used by automotive industry. The EDF scheduling algorithm is implemented by assigning priorities to tasks according to their relative deadlines. Then, during the execution, a task is released only if its absolute deadline is less than the one of the currently running task. Otherwise, the task will be delayed until the time when the running task finishes its execution.

Kim *et al.* [13] propose the SPIRIT uKernel that is based on a two-level hierarchical scheduling framework simplifying integration of real-time applications. The SPIRIT uKernel provides a separation between real-time applications by using partitions. Each partition executes an application, and uses the Fixed Priority Scheduling (FPS) policy as a local scheduler to schedule the application's tasks. An offline scheduler (timetable) is used to schedule the partitions (the applications) on a global level. Each partition provides kernel services for its application and the execution is in user mode to provide stronger protection.

Parkinson [21] uses the same principle and describes the VxWorks 653 operating system which was designed to support ARINC653. The architecture of VxWorks 653 is based on partitions, where a Module OS provides global resource and scheduling for partitions and a Partition OS implemented using VxWorks microkernel provides scheduling for application tasks.

The work presented in this paper differs from the last two works in the sense that it implements a hierarchical scheduling framework in a commercial operating system without changing the OS kernel. Furthermore, the work differs from the above approaches in the sense that it implements a hierarchical scheduling framework intended for open environments [9], where real-time applications may be developed independently and unaware of each other and still there should be no problems in the integration of these applications into one environment. A key here is the use of well defined *interfaces* representing the collective resource requirements by an application, rich enough to allow for integration with an arbitrary set of other applications without having to redo any kind of application internal analysis.

3 System model

In this paper, we only consider a simple periodic task model $\tau_i(T_i, C_i, D_i)$ where T_i is the task period, C_i is a worst-case execution time requirement, and D_i is a relative deadline ($C_i \leq D_i \leq T_i$). We assume that all tasks are independent of each others, i.e., there is no sharing of logical resources between tasks.

The HSF schedules subsystems $S_s \in \mathcal{S}$, where $\mathcal{S}$ is the set representing the whole system of subsystems. Each subsystem S_s consists of a set of tasks and a local scheduler (RM or EDF), and the global (system) scheduler can also be either RM or EDF. The collective real-time requirements of S_s is referred to as a *timing-interface*. The subsystem interface is defined as (P_s, Q_s), where P_s is a period, and Q_s is a *budget* that represents an execution time requirement that will be provided every period P_s.

4 VxWorks

VxWorks is a commercial real-time operating system that was developed by Wind River with a focus on performance, scalability and footprint. Many interesting features are provided with VxWorks, which make it widely used in industry, such as; Wind micro-kernel, efficient task management and multitasking, deterministic context switching, efficient interrupt and exception handling, POSIX pipes, counting semaphores, message queues, signals, and scheduling, pre-emptive and round-robin scheduling etc. (see [28] for more details).

The VxWorks micro-kernel supports the priority preemptive scheduling policy with up to 256 different priority levels and a large number of tasks, and it also supports the round robin scheduling policy.

VxWorks offers two different modes for application-tasks to execute; either kernel mode or user mode. In kernel mode, application-tasks can access the hardware resources directly. In user mode, on the other hand, tasks can not directly access hardware resources, which provides greater protection (e.g., in user mode, tasks can not crash the kernel). Kernel mode is provided in all versions of VxWorks while user mode was provided as a part of the Real Time Process (RTP) model, and it has been introduced with VxWorks version 6.0 and beyond.

In this paper, we are considering kernel mode tasks since such a design would be compatible with all versions of VxWorks and our application domains include systems with a large legacy in terms of existing source codes. We are also considering the fixed priority preemptive scheduling policy for the kernel scheduler (not the round robin scheduler). A task's priority should be set when the task is created, and the task's priority can be changed during the execution. Then, during runtime, the highest priority ready task will always execute. If a task with priority higher than that of the running task becomes ready to execute, then the scheduler stops the execution of the running task and instead executes the one with higher priority. When the running task finishes its execution, the task with the highest priority among the ready tasks will execute.

When a task is created, an associated Task Control Block (TCB) is created to save the task's context (e.g., CPU environment and system resources, during the context switch). Then, during the life-cycle of a task the task can be in one or a combination of the following states [27] (see Figure 1):

- **Ready state**, the task is waiting for CPU resources.
- **Suspended state**, the task is unavailable for execution but not delayed or pending.
- **Pending state**, the task is blocked waiting for some resource other than the CPU.
- **Delayed state**, the task is sleeping for some time.

Note that the kernel scheduler sorts all tasks that are ready to execute in a queue called the *ready queue*.

4.1 Scheduling of time-triggered periodic tasks

A periodic task is a task that becomes ready for execution periodically once every n-th time unit, i.e., a new instant of the task is executed every constant period of time. Most commercial operating systems, including VxWorks, do not directly support the periodic task model [19]. To implement a periodic task, when a task finishes its execution, it sleeps until the beginning of its next period. Such periodic behaviour can be implemented in the task by the usage of timers. Note that a task typically does not finish its execution at the same time always, as execution times and response times vary from one period to another. Hence, using timers may not be easy and accurate as the task needs to calculate the absolute time for next period, whenever it finishes its execution. This is because preemption may happen between the time measurement and calling the sleep function.

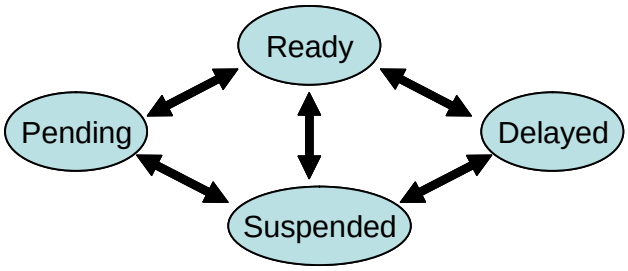


Figure 1. The application task state.

In this project we need to support periodic activation of *servers* in order to implement the hierarchical scheduling framework. The reason for this is that we base our hierarchical scheduling framework around the periodic resource model [23], and a suitable implementation of this is by the usage of periodic servers [16, 25] that charge their budget every constant period, i.e., the servers behave like periodic tasks.

4.2 Supporting arbitrary schedulers

There are two ways to support arbitrary schedulers in VxWorks:

1. Using the VxWorks custom kernel scheduler [26].
2. Using the original kernel scheduler and manipulating the ready queue by changing the priority of tasks and/or activating and suspending tasks.

In this paper, we are using the second approach since implementing the custom kernel scheduler is a relatively complex task compared with manipulating the ready queue. However, it will be interesting to compare between the two methods in terms of CPU overhead, and we leave this as a future work.

In the implementation of the second solution, we have used an Interrupt Service Routine (ISR) to manipulate the tasks in the ready queue. The ISR is responsible for adding tasks in the ready queue as well as changing their priorities according to the hierarchical scheduling policy in use. In the remainder of this paper, we refer to the ISR as the User Scheduling Routine (USR). By using the USR, we can implement any desired scheduling policy, including common

ones such as Rate Monotonic (RM) and Earliest Deadline First (EDF).

5 The USR custom VxWorks scheduler

This section presents how to schedule periodic tasks using our scheduler, the User Scheduling Routine (USR).

5.1 Scheduling periodic tasks

When a periodic task finishes its execution, it changes its state to suspended by explicitly calling the suspend function. Then, to implement a periodic task, a timer could be used to trigger the USR once every new task activation time to release the task (to put it in the ready queue).

The solution to use a timer triggering the USR once every new period can be suitable for systems with a low number of periodic tasks. However, if we have a system with n periodic tasks such a solution would require the use of n timers, which could be very costly or not even possible. In this paper we have used a scalable way to solve the problem of having to use too many timers. By multiplexing a single timer, we have used a single timer to serve n periodic tasks.

The USR stores the next activation time of all tasks (absolute times) in a sorted (according to the closest time event) queue called Time Event Queue (TEQ). Then, it sets a timer to invoke the USR at the time equal to the shortest time among the activation times stored in the TEQ. Also, the USR checks if a task misses its deadline by inserting the deadline in the TEQ. When the USR is invoked, it checks all task states to see if any task has missed its deadline. Hence, an element in the TEQ contains (1) the absolute time, (2) the id of task that the time belongs to, and (3) the event type (task next activation time or absolute deadline). Note that the size of the TEQ will be $2 * n * B$ bytes (where B is the size in bytes of one element in the TEQ) since we need to save the task's next period time and deadline time.

When the USR is triggered, it checks the caused of the triggering. There are two causes for the USR to be triggered: (1) a task is released, and (2) the USR will check for deadline misses. If a task has been released, the USR will do the following:

- Update the next activation time associated with the task that cause triggering of the USR in the TEQ and re-insert it in the TEQ according to the updated times.
- Set the timer equal to the shortest time in the TEQ so that the USR will be triggered at that time.
- Change the state of the task to Ready and change priorities of tasks if required depending on the scheduler if it is EDF or RM.

If the USR will check for deadline misses, then it will:

- Update the next absolute deadline time, associated with the task that caused triggering of the USR, and re-insert it in the TEQ according to the updated times.
- Set the timer with the shortest time in the TEQ to trigger the USR at that time.
- Check the state of the task to see if it is Ready. If so, the task missed its deadline, and the deadline miss function will be activated.

Updating the next activation time and absolute deadline of a task in the TEQ is done by adding the period of the task that caused the USR invocation to the previous task activation time. The USR does not use the system time as a time reference. Instead it uses a time variable as a time reference. The reason for using a time variable is that we can, in a flexible manner, select the size of variables that save absolute time in bits. The benefits of such an approach is that we can control the size of the TEQ since it saves the absolute times, and it also minimizes the overhead of implementing 64 bits operations on 32 bit microprocessor [4], as an example. The reference time variable t_s is initialized (i.e., $t_s = 0$) at the first execution of the USR. The value of t_s is updated every time that the USR executes and it will equal to the time from the TEQ that triggered the USR.

When a task τ_i is released for the first time, the absolute next activation time is equal to $t_s + T_i$ and its absolute deadline is equal to $t_s + D_i$.

To avoid time consuming operations, e.g., multiplications and divisions, that increase the system overhead inherent in the execution of the USR, all absolute times (task periods and relative deadlines) are saved in system tick unit (system tick is the interval between two consecutive system timer interrupts). However, depending on the number of bits used to store the absolute times, there is a maximum value that can be saved safely. Hence, saving absolute times in the TEQ may cause problems related to overrun of time, i.e., the absolute times become too large such that the value can not be stored using the available number of bits. To avoid this problem, we apply a wrapping algorithm.

Evaluating the time at which to trigger the USR again (next time) is done by $TEQ[1] - t_s$ where $TEQ[1]$ is the first element in the queue after updating the TEQ as well as sorting it, i.e., the shortest time in the TEQ. The USR checks to see if there are more than one task that have the same current activation time and absolute deadline. If so, the USR serves all these tasks to minimize the unnecessary overhead of executing the USR several times.

5.2 RM scheduling policy

Each task will have a fixed priority during run-time when Rate Monotonic (RM) is used, and the priorities are assigned according to the RM scheduling policy. If only RM

is used in the system, no additional operations are required to be added to the USR since the kernel scheduler schedule all tasks directly according to their priorities, and the higher priority tasks can preempt the execution of the lower priority task. Hence, the implementation overhead for RM will be limited to the overhead of adding a task in the ready queue and managing the timer for the next period (saving the absolute time of the new period and finding the shortest next time in the TEQ) for periodic tasks.

The schedulability analysis for each task is as follows [15];

$$\forall \tau_i, 0 < \exists t \leq T_i \;\; \texttt{dbf}(i,t) \leq t. \tag{1}$$

And $\texttt{dbf}(i,t)$ is evaluated as follows

$$\texttt{dbf}(i,t) = C_i + \sum_{\tau_k \in \texttt{HP}(i)} \left\lceil \frac{t}{T_k} \right\rceil C_k, \tag{2}$$

where $\texttt{HP}(i)$ is the set of tasks with higher-priority than that of τ_i.

Eq. (2) can be easily modified to include the effect of using the USR on the schedulability analysis. Note that the USR will be triggered at the beginning of each task to release the task, so it behaves like a periodic task with priority equal to the maximum possible priority (the USR can preempt all application tasks). Checking the deadlines for tasks by using the USR will add more overhead, however, also this overhead has a periodic nature as the task release presented previously.

Eq. (3) includes the deadline and task release overhead caused by the USR in the response time analysis as shown bellow,

$$\texttt{dbf}(i,t) \;=\; C_i + \sum_{k \in HP(i)} \left\lceil \frac{t}{T_k} \right\rceil C_k + \sum_{j \in \Gamma} \left\lceil \frac{t}{T_j} \right\rceil (X_R + X_D) \tag{3}$$

where X_R is the worst-case execution time of the USR when a task is released and X_D is the worst-case execution time of the USR when it checks for deadline misses (currently, in case of deadline missing, the USR will only log this event into a log file).

5.3 EDF scheduling policy

For EDF, the priority of a task changes dynamically during run-time. At any time t, the task with shorter deadline will execute first, i.e., will have the highest priority. To implement EDF in the USR, the USR should update the priorities of all tasks that are in the Ready Queue when a task is added to the Ready Queue, which can be costly in terms of overhead. Hence, on one hand, using EDF on top of commercial operating systems may not be efficient depending on the number of tasks, due to this sorting. However, the EDF scheduling policy provides, on other hand, better CPU utilization compared with RM, and it also has a lower number of context switches which minimizes context switch related overhead [5].

In the approach presented in this paper, tasks are already sorted in the TEQ according to their absolute times due to the timer multiplexing explained earlier. Hence, as the TEQ is already sorted according to the absolute deadlines, the USR can easily decide the priorities of the tasks according to EDF without causing too much extra overhead for evaluating the proper priority for each task.

The schedulability test for a set of tasks that use EDF is shown in Eq. (4) [2] which includes the case when task deadlines are allowed to be less than or equal to task periods.

$$\forall t > 0, \quad \sum_{i}^{n} \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor \cdot C_i \leq t \tag{4}$$

The overhead of implementing EDF can also be added to Eq. (4). Hence, Eq. (5) includes the overhead of releasing tasks as well as the overhead of checking for deadline misses.

$$\forall t > 0, \quad \sum_{i}^{n} \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor \cdot C_i + \sum_{j \in \Gamma} \left\lceil \frac{t}{T_j} \right\rceil (X_R + X_D) \leq t \tag{5}$$

5.4 Implementation and overheads of the USR

To implement the USR, we have used the following VxWorks service functions;

- Q_PUT - insert a node into a multi-way queue (ready queue).
- Q_REMOVE - remove a node from a multi-way queue (ready queue).
- taskCreat - create a task.
- taskPrioritySet - set a tasks priority.

We present our initial results inherent in the implementation of the USR, implementing both the Rate Monotonic (RM) scheduler as well as the Earliest Deadline First (EDF) scheduler. The implementations were performed on ABB robot controller with a Pentium 200 MHz processor running the VxWorks operating system version 5.2. To trigger the USR for periodic tasks, we have used watchdog timers where the next expiration time is given in number of ticks. The watchdog uses the system clock interrupt routine to count the time to the next expiration. The platform provides system clock with resolution equal to $4500 ticks/s$.

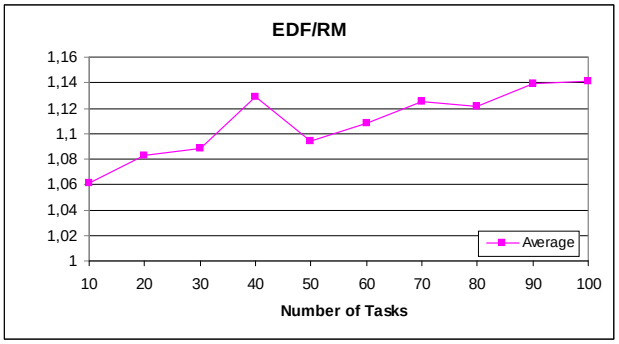


Figure 2. EDF normalized against RM, for average USR execution time.

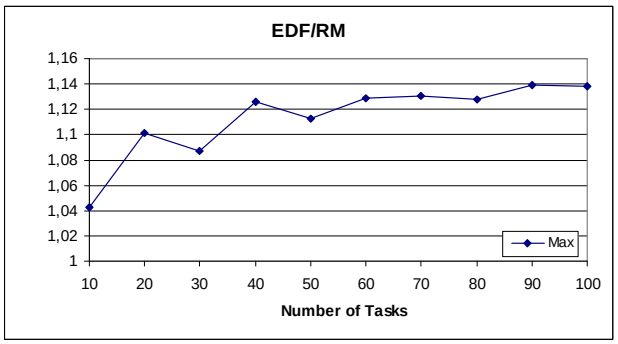


Figure 3. EDF normalized against RM, for maximum USR execution time.

The measurement of the execution time of the USR is done by reading a timestamp value at the start as well as at the end of the USR's execution. Note that the timestamp is connected to a special hardware timer with resolution $12000000 ticks/s$.

Table 1 shows the execution time of the USR when it performs RM and EDF scheduling, as well as deadline miss checking, as a function of the number of tasks in the system. The worst case execution time for USR will happen when USR deletes and then inserts all tasks from and to TEQ and to capture this, we have selected a same period for all tasks. The table shows the minimum, maximum and average out of 50 measured values. Comparing between the results of the three cases (EDF, RM, deadline miss), we can see that there is no big difference in the execution time of the USR. The reason for this result is that the execution of the USR for EDF, RM and deadline miss checking all includes the overhead of deletion and re-inserting the tasks in the TEQ, which is the dominating part of the overhead. As expected, EDF causes the largest overhead because it changes the priority of all tasks in the ready queue during run-time. Figures 2-3 show that EDF imposes between $6 - 14\%$ extra overhead compared with RM.

6 Hierarchical scheduling

A Hierarchical Scheduling Framework (HSF) supports CPU sharing among subsystems under different scheduling policies. Here, we consider a two-level scheduling framework consisting of a global scheduler and a number of local schedulers. Under global scheduling, the operating system (global) scheduler allocates the CPU to subsystems. Under local scheduling, a local scheduler inside each subsystem allocates a share of the CPU (given to the subsystem by the global scheduler) to its own internal tasks (threads).

We consider that each subsystem is capable of exporting its own interface that specifies its collective real-time CPU requirements. We assume that such a subsystem interface is in the form of the periodic resource model (P_s, Q_s) [23]. Here, P_s represents a *period*, and Q_s represents a *budget*, or an execution time requirement within the period $(Q_s \leq P_s)$. By using the periodic resource model in hierarchical scheduling frameworks, it is guaranteed [23] that all timing constraints of internal tasks within a subsystem can be satisfied, if the global scheduler provides the subsystem with CPU resources according to the timing requirements imposed by its subsystem interface. We refer interested readers to [23] for how to derive an interface (P_s, Q_s) of a subsystem, when the subsystem contains a set of internal independent periodic tasks and the local scheduler follows the RM or EDF scheduling policy. Note that for the derivation of the subsystem interface (P_s, Q_s), we use the demand bound functions that take into account the overhead imposed by the execution of USR (see Eq. (3) and (5)).

6.1 Hierarchical scheduling implementation

Global scheduler: A subsystem is implemented as a periodic server, and periodic servers can be scheduled in a similar way as scheduling normal periodic tasks. We can use the same procedure described in Section 5 with some modifications in order to schedule servers. Each server should include the following information to be scheduled: (1) server period, (2) server budget, (3) remaining budget, (4) pointer to the tasks that belong to this server, and (5) the type of the local scheduler (RM or EDF). Moreover, to schedule servers we need:

- **Server Ready Queue** to store all servers that have non zero remaining budget. When a server is released at the beginning of its period, its budget will be charged to the maximum budget Q, and the server will be added

Number of tasks	X_R (RM)			X_R (EDF)			X_D (Deadline miss check)		
	Max	Average	Min	Max	Average	Min	Max	Average	Min
10	71	65	63	74	70	68	70	60	57
20	119	110	106	131	118	115	111	100	95
30	172	158	155	187	172	169	151	141	137
40	214	202	197	241	228	220	192	180	175
50	266	256	249	296	280	275	236	225	219
60	318	305	299	359	338	331	282	268	262
70	367	352	341	415	396	390	324	309	304
80	422	404	397	476	453	444	371	354	349
90	473	459	453	539	523	515	415	398	393
100	527	516	511	600	589	583	459	442	436

Table 1. USR execution time in μs, the maximum, average and minimum execution time of 45 measured values for each case.

in the Server Ready Queue. When a server execute its internal tasks for some time x, then the remaining budget of the server will be deceased with x, i.e., reduced by the time that the server execute. If the remaining budget becomes zero, then the server will hand over the control to the global scheduler to select and remove the highest priority server from Server Ready Queue.

- **Server TEQ** to release the server at its next absolute periodic time since we are using periodic servers and also track their remaining budgets.

The Server Ready Queue is managed by the routine that is responsible for scheduling the servers. Tracking the remaining budget of a server is solved as follows; whenever a server starts running, it sets a deadline equal to the current time plus its remaining budget. When a server is preempted by another server, it updates the remaining budget by subtracting the time that has passed since the last release. When the server executes its internal tasks until the time when the server deadline event triggers, it will set its remaining budget to zero, and the scheduling routine removes the server from the Server Ready Queue.

Local scheduler: When a server is given the CPU resources, the ready tasks that belong to the server will be able to execute. We have investigated two approaches to deal with the tasks in the Ready Queue when a server is given CPU resources:

- All tasks that belong to the server that was previously running will be removed from the Ready Queue, and all ready tasks that belong to the new running server will be added to the Ready Queue, i.e., swapping of the servers' task sets.
- The priority of all tasks that belong to the preempted server will be set to a lower (the lowest) priority, and the priority of all tasks that belong to the new running server will be raised as if they were executing exclusively on the CPU, scheduled according to the local scheduling policy in use by the subsystem.

The advantage of the second approach is that it can give the unused CPU resources to tasks that belong to other servers. However, the disadvantage of this approach is that the kernel scheduler always sorts the tasks in the Ready Queue and the number of tasks inside Ready Queue using the second approach will be higher which may impose more overhead for sorting tasks. In this paper, we consider the first approach since we support only periodic tasks. When a server is running, all interrupts that are caused by the local TEQ, e.g., releasing tasks and checking deadline misses, can be served without problem. However, if a task is released or its deadline occurs during the execution of another server, the server that includes the task, may miss this event. To solve this problem, whenever a server is preempted or finishes its budget, it disables the timer from the local TEQ. Then, when the server starts running again, it will check for all past events in the local TEQ and serve them.

Note that the time wrapping algorithm described in section 5.1 should take into account all local TEQ's for all servers and the server event queue, because all these event queues share the same absolute time.

Figure (4) illustrates the implementation of hierarchical scheduling framework which includes an example with three servers S_1, S_2, S_3 with global and local RM schedulers, the priority of S_1 is the highest and the priority of S_3 is the lowest. Suppose a new period of S_3 starts at time t_0 with a budget equal to Q_3. Then, the USR will change the state of S_3 to Ready, and since it is the only server that is ready to execute, the USR will;

- add the time that the budget will expire which equals to $t_0 + Q_3$ into the server event queue and also add the next period event in the server event queue.

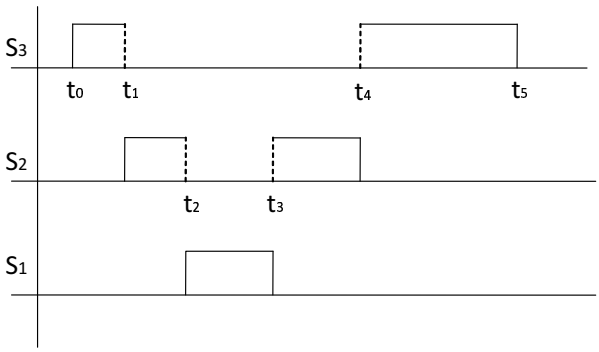


Figure 4. Simple servers execution example.

Number of servers	Max	Average	Min
10	91	89	85
20	149	146	139
30	212	205	189
40	274	267	243
50	344	333	318
60	412	400	388
70	483	466	417
80	548	543	509
90	630	604	525
100	689	667	570

Table 2. Maximum, average and minimum execution time of USR with 100 measured values as a function of the number of servers.

- check all previous events that have been occurred while the server was not active(by checking if there is task release or deadline check even in the time interval $[t^*, t_0]$ where t^* is the latest time at which the budget of S_3 has been expired).
- start the local scheduler.

At time interval t_1 the server S_2 becomes Ready and it has higher priority than S_3. So S_2 will preempt S_3 and in addition to the previously explained action, the USR will remove all tasks that belongs to S_3 from ready queue and save the remaining budget which equals to $Q_3 - (t_1 - t_0)$ and then remove the budget expiration event from the server event queue. Note that when S_3 executes next time it will use the remaining budget to calculate the budget expiration event.

The USR execution time depends on the number of the servers, and the worst case happens when all servers are released at the same time. In addition, the execution time of USR also depends on the number of the ready tasks in both the currently running server to be preempted and the server to preempt. The USR removes all ready tasks that belong to the preempted server from ready queue and adds all ready tasks that belong to the preempting server with highest priority into the ready queue. Here, the worst case scenario is that all tasks of both servers are ready at that time. Table 2 shows the execution time of USR (when a server is released) as a function of the number of servers using RM as a global scheduler at the worst case, where all the servers are released at the same time, just like the case shown in the previous section. Here, we consider each server has a single task in order to purely investigate the effect of the number of servers on the execution time of USR.

6.2 Example

In this section, we will show the overall effect of implementing HSF using simple example, however, the results from the following example are specific for this example because as we showed in the previous section that the overhead is a function of many parameters such as number of servers, number of tasks, servers periods and budgets (affect the number of preemptions). In this example we use RM as a local and global scheduler, the servers and associated tasks parameters are shown in Table 3 assuming for all tasks $T_i = D_i$.

The measured overhead utilization is about 2.85% and the measured release jitter for task τ_3 in server S_3 (which is the lowest priority task in the lowest priority server) is about $49ms$ and the measured worst case response time is $208.5ms$ and the finishing time jitter is $60ms$.

7 Summary

This paper has presented our work on the implementation of our hierarchical scheduling framework in a commercial operating system, VxWorks. We have chosen to implement it in VxWorks so that it can easily be tested in an industrial setting, as we have a number of industrial partners with applications running on VxWorks and we intend to use them as case studies for an industrial deployment of the hierarchical scheduling framework.

This paper demonstrates the efficacy of hierarchical scheduling framework through its implementation over VxWorks. In particular, it presents several measurements of overheads that its implementation imposes. It shows that a hierarchical scheduling framework can effectively achieve the clean separation of subsystems in terms of timing interference (i.e., without requiring any temporal parameters of other subsystems) with reasonable implementation overheads.

In the next stage of this implementation project, we intend to implement synchronization protocols in hierarchical scheduling frameworks [3]. In addition, our future work

$S_1(P_1 = 5, Q_1 = 1)$			$S_2(P_2 = 6, Q_2 = 1)$			$S_3(P_3 = 70, Q_3 = 20)$		
τ_i	T_i	C_i	τ_i	T_i	C_i	τ_i	T_i	C_i
τ_1	20	1	τ_1	25	1	τ_1	140	7
τ_2	25	1	τ_2	35	1	τ_2	150	7
τ_3	30	1	τ_3	45	1	τ_3	300	30
τ_4	35	1	τ_4	50	1			
τ_5	40	7	τ_5	55	7			
-	-	-	τ_6	60	7			

Table 3. System parameters in μs.

includes supporting sporadic tasks in response to specific events such as external interrupts. Instead of allowing them to directly add their tasks into the ready queue, we consider triggering the USR to take care of such additions. We also plan to support aperiodic tasks while bounding their interference to periodic tasks by the use of some server-based mechanisms. Moreover, we intend to extend the implementation to make it suitable for more advanced architectures including multicore processors.

Acknowledgements

The authors wish to express their gratitude to the anonymous reviewers for their helpful comments, as well as to Clara Maria Otero Pérez for detailed information regarding a similar implementation and suggestions of improving our work.

References

[1] L. Almeida and P. Pedreiras. Scheduling within temporal partitions: response-time analysis and server design. In *Proc. of the Fourth ACM International Conference on Embedded Software*, September 2004.

[2] S. Baruah, R. Howell, and L. Rosier. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Journal of Real-Time Systems*, 2:301–324, 1990.

[3] M. Behnam, I. Shin, T. Nolte, and M. Nolin. Sirap: A synchronization protocol for hierarchical resource sharing in real-time open systems. In *Proceedings of the 7th ACM and IEEE International Conference on Embedded Software (EMSOFT'07)*, Salzburg, Austria, October 2007.

[4] G. Buttazzo and P. Gai. Efficient implementation of an edf scheduler for small embedded systems. In *Proceedings of the 2^{nd} International Workshop Operating System Platforms for Embedded Real-Time Applications (OSPERT'06) in conjunction with the 18^{th} Euromicro International Conference on Real-Time Systems (ECRTS'08)*, Dresden, Germany, July 2006.

[5] G. C. Buttazzo. Rate Monotonic vs. EDF: Judgment day. *Real-Time Systems*, 29(1):5–26, January 2005.

[6] F. S. G. W. C. Diederichs, U. Margull. An application-based edf scheduler for osek/vdx. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, 2008.

[7] R. I. Davis and A. Burns. Hierarchical fixed priority preemptive scheduling. In *Proceedings of the 26^{th} IEEE International Real-Time Systems Symposium (RTSS'05)*, December 2005.

[8] R. I. Davis and A. Burns. Resource sharing in hierarchical fixed priority pre-emptive systems. In *Proceedings of the 27^{th} IEEE International Real-Time Systems Symposium (RTSS'06)*, December 2005.

[9] Z. Deng and J. W.-S. Liu. Scheduling real-time applications in an open environment. In *Proc. of IEEE Real-Time Systems Symposium*, pages 308–319, December 1997.

[10] Evidence Srl. ERIKA Enterprise RTOS. URL: http://www.evidence.eu.com.

[11] X. Feng and A. Mok. A model of hierarchical real-time virtual resources. In *Proc. of IEEE Real-Time Systems Symposium*, pages 26–35, December 2002.

[12] N. Fisher, M. Bertogna, and S. Baruah. The design of an edf-scheduled resource-sharing open environment. In *Proceedings of the 28^{th} IEEE International Real-Time Systems Symposium (RTSS'07)*, pages 83–92, December 2007.

[13] D. Kim, Y. Lee, and M. Younis. Spirit-ukernel for strongly partitioned real-time systems. In *Proc. 7th International Conference on Real-Time Computing Systems and Applications (RTCSA 2000)*, Cheju Island, South Korea, December 2000. IEEE Computer Society.

[14] T.-W. Kuo and C. Li. A fixed-priority-driven open environment for real-time applications. In *Proc. of IEEE Real-Time Systems Symposium*, pages 256–267, December 1999.

[15] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Proc. of IEEE Real-Time Systems Symposium*, pages 166–171, 1989.

[16] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings of 8^{th} IEEE International Real-Time Systems Symposium (RTSS'87)*, pages 261–270, San Jose, California, USA, December 1987. IEEE Computer Society.

[17] G. Lipari and S. Baruah. Efficient scheduling of real-time multi-task applications in dynamic systems. In *Proc. of IEEE Real-Time Technology and Applications Symposium*, pages 166–175, May 2000.

[18] G. Lipari and E. Bini. Resource partitioning among real-time applications. In *Proc. of Euromicro Conference on Real-Time Systems*, July 2003.
[19] J. Liu. Real-time systems. *Prentice Hall*, 2000.
[20] A. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *Proc. of IEEE Real-Time Technology and Applications Symposium*, pages 75–84, May 2001.
[21] L. K. P. Parkinson. Safety critical software development for integrated modular avionics. In *Wind River white paper. URL http://www.windriver.com/whitepapers/*, 2007.
[22] S. Saewong, R. Rajkumar, J. Lehoczky, and M. Klein. Analysis of hierarchical fixed-priority scheduling. In *Proc. of Euromicro Conference on Real-Time Systems*, June 2002.
[23] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proc. of IEEE Real-Time Systems Symposium*, pages 2–13, December 2003.
[24] I. Shin and I. Lee. Compositional real-time scheduling framework. In *Proc. of IEEE Real-Time Systems Symposium*, December 2004.
[25] B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1(1):27–60, June 1989.
[26] Wind River. VxWorks KERNEL PROGRAMMERS GUIDE 6.2.
[27] Wind River. VxWorks PROGRAMMERS GUIDE 5.5.
[28] Wind River. Wind River VxWorks 5.x. http://www.windriver.com/.

Estimating Context Switch Cost: A Practitioner's Approach

Robert Kaiser
Distributed Systems Lab,
University of Applied Sciences, Wiesbaden, Germany
kaiser@informatik.fh-wiesbaden.de

Abstract

Context switch cost is a limiting factor for many real-time systems: In order to improve desired properties such as quick response or low jitter, a system should support high switch rates, but the resulting switch overheads become prohibitive at a certain point. In order to find a favourable tradeoff between real-time properties and the amount of CPU resources that have to be sacrificed to achieve them, it would be necessary to determine the expected context switch cost for a given application and system configuration. This problem, however, is intractable in most practical situations.

In this work, we attempt a practitioner's approach to quantifying context switch cost: Based on a simple model of system behaviour, we present methods to estimate the expected overheads. To validate our methods we compare the behaviour of real systems against that of our model. These experiments also yield parameters along the way which are then used to configure our methods so they reflect realistic scenarios. The overheads can be attributed to individual tasks in the system, so, if there are tasks with different timing requirements, each of them can use its own, specifically adapted estimation. As a demonstration, we integrate our method into a simulation of two different proportional share scheduling algorithms and use it to compute the estimated system overhead as a function of the scheduler's minimal time allocation.

1 Introduction

This work is motivated by our research on the applicability of virtualisation to real-time computing: Virtual machine monitors (VMMs) such as Xen ([1]) or VMWare ([14]), besides their other responsibilities, also act as schedulers, switching the physical CPU between multiple clients (which are referred to as *virtual machines* in the context of virtualisation). The goal of the scheduler is to give each client the notion of having continuous access to a portion of the physical CPU's computational resources. Therefore, VMMs typically use some form of proportional share scheduling ([15, 5, 4]). However, since a CPU can only be allocated to one activity at a time, real proportional share schedulers can only approximate the continuous CPU availability that the idealised model ([11]) postulates by switching the CPU between clients in a round robin fashion. The quality of the approximation improves as the time slice length is made smaller. Also, if any of the virtual machines host real-time tasks that have to provide service while their enclosing virtual machine is inactive, these service requests must wait until the next virtual machine time slice. The resulting delay is also proportional to the time slice length ([7]). These are reasons to make time slices as small as possible. But the system overhead that is caused by the increased switch rate rises as well, reaching unacceptable levels at some point. From a system design perspective, it would be desirable to be able to quantify this tradeoff, i.e. given an application and a certain loss of performance that is deemed to be acceptable, what would be the smallest possible time slice length, and what would be the corresponding longest possible delay that a real-time task might have to wait for the CPU?

To answer these questions, it is necessary to understand the mechanisms that contribute to the performance loss, namely the cost of on-line scheduling and the cost of context switching. Especially the latter is hard to formalise, which is why it has often been neglected. In this contribution, we present methods to *estimate* the overhead of context switches using empirical data. Obviously, this approach has the drawback being imprecise and we can only resort to experimentation to support our claims. Therefore, applying our methods to hard real-time problems which need to formally prove that no deadline can possibly be missed, would be inappropriate. The advantage, however, is that the estimations can be computed with little effort and that only superficial knowledge about the client is required. The computations are simple enough to allow inclusion into on-line schedulers, which could then dynamically take into account the estimated switch cost

when making scheduling decisions.

Besides our own use case (i.e. virtual machine monitors), our approach could also prove useful when comparing scheduling approaches which differ in the amount of context switches they tend to require (e.g. [3]). Also, for scheduling approaches which are based on proportional sharing and assume applicability of the idealised model of continuous CPU availability (e.g. [12]), our approach could indicate the range of scheduling frequencies for which this assumption is actually valid. Furthermore, schedulers for symmetric multiprocessor (and multicore) systems could benefit: The same effects that contribute to the system overhead in a uniprocessor system are also responsible for the cost of migrating tasks between cores in a multicore system. Thus, a multicore scheduler could make a better judgement wether or not a possible task migration would be worth the effort.

2 Background

We denote processing capacities or "amounts of service" as a potential of a CPU to perform a certain amount of work by executing program code. If a CPU progresses through program code at a constant rate r, the capacity corresponding to a time interval $[t_1, t_2], t_2 \geq t_1$ would be $r \cdot (t_2 - t_1)$. In reality, though, the progress rate of a processor may change over time due to pipeline stalls, varying cache locality or changing clock frequencies. Thus, with $r(t)$ denoting the CPU progress rate as a function of time, a more general definition of the processing capacity of a CPU corresponding to time interval $[t_1, t_2]$ is:

$$W_{cpu}(t_1, t_2) = \int_{t_1}^{t_2} r(\tau) d\tau \tag{1}$$

We can also specify an equivalent constant average progress rate $\bar{r}(t_1, t_2)$ such that during the same time interval $[t_1, t_2]$, the same amount of progress is made:

$$\bar{r}(t_1, t_2) = \frac{W_{cpu}(t_1, t_2)}{t_2 - t_1} = \frac{\int_{t_1}^{t_2} r(\tau) d\tau}{t_2 - t_1} \tag{2}$$

A program will make the fastest progress if it runs alone on a machine without interruptions. If, instead, it has to share the CPU with other tasks, its progress rate will degrade to some extent, not only because of the other tasks using the CPU, but also because of the additional cost of maintaining the multi-tasking environment. The *system overhead* is the latter part of the degradation. Two major causes contribute to this overhead:

1. Scheduling overhead: The scheduler must be invoked from time to time to decide wether or not to switch to another task. This is usually implemented by means of a timer interrupt which triggers when the current task's time slice expires. Processing of this interrupt entails direct cost. i.e. the time taken by the CPU to enter/leave the operating system, to save/restore processor context, and –last but not least– to execute the scheduling algorithm. Note that this overhead does not depend on the outcome of the scheduler's decision: The task's context is stored upon entry into the operating system kernel and is restored upon kernel exit. Wether it is the previous task that is being resumed or another one, the effort of restoring the processor context after interrupt processing is the same.

2. Context switch overhead: In contrast to the scheduling overhead, context switch overhead occurs only in conjunction with a task switch. It is caused by the instruction, data and TLB caches that exist in all modern CPU architectures: Whenever a task is switched to, it will usually find these caches in a more or less "polluted" state, i.e. they will contain entries which were loaded by previous tasks and which are not useful (or not accessible) to the new task. As the new task references new program code and data objects, the CPU loads them into the respective caches, which requires additional effort. The newly loaded objects evict objects from the caches that were previously loaded by other tasks. For data objects that were modified while in the cache, this means that they must now be stored back to main memory, which leads to even more additional CPU effort. Moreover, in some systems[1], each task has its own, MMU-protected address space. Therefore, switching between them also involves an address space switch, which, for some popular machine architectures, requires that the CPU's translation lookaside buffer (TLB) be flushed. Thus, as the new task executes, the processor must perform expensive page table walks whenever a new page of memory is referenced ([9]).

While a task switch is always preceded by a scheduler invocation, not every scheduler invocation is necessarily followed by a task switch. Therefore, we consider the cost of scheduler invocation and the cost of task switches separately.

2.1 Scheduling overhead

Scheduling overhead is perceived by the clients as a window of time during which none of them makes any progress. For off-line schedulers, the execution time of the scheduler is constant (the next task to be activated is simply taken

[1] Virtual machine monitors being a prime example

from a table). Also, many on-line scheduling algorithms exist which are able to determine the next task in constant run-time. The code which the CPU executes to enter or leave the scheduler is typically of limited complexity and well-known length for any given operating system and platform. Therefore, we assume that all scheduler invocations are of uniform length. In the following, we denote this time window length as s. To attribute this overhead to individual tasks, we make a convention that each scheduler invocation is entirely at the cost of the task that was active before the scheduler was invoked.

If a task is able to use a CPU exclusively during a time interval $[t_1, t_2]$, the amount of service it thus consumes is $W_{cpu}(t_1, t_2)$ as defined in equation (1). If, however, during that interval, there is a sub-interval of length s (i.e. a scheduler invocation) during which it does not progress, the task will take s time units longer for the same amount of service.

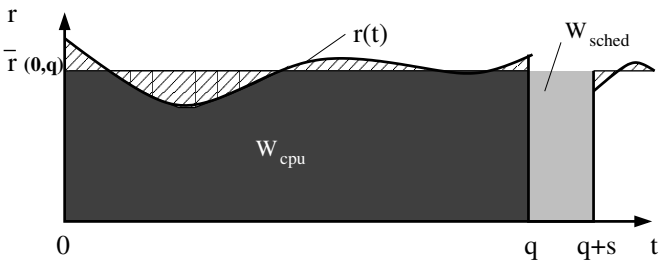


Figure 1. Service vs. scheduling overhead.

We consider a task that is activated at time $t = 0$ and then run for the duration of a time slice (or *quantum*) of length q (See figure 1). When the quantum expires, i.e. at time $t = q$, the task is preempted and the scheduler is invoked. At time $t = q + s$, either the same task or another task is executed. If there had not been a scheduler invocation, the task would have been able to proceed at its current rate also during the time interval $[q, q + s]$. Thus, the loss of service suffered by the task and thus the cost of the scheduler invocation is the CPU capacity corresponding to this time interval:

$$W_{sched} = s \cdot \bar{r}(0, q)$$

The *relative overhead*, i.e. the ratio between the amount of lost service and the total amount of service delivered by the CPU during the time interval $[0, q + s]$ is:

$$O_{sched} = \frac{W_{sched}}{W_{cpu}(0, q) + W_{sched}} = \frac{s}{s + q} \qquad (3)$$

Most proportional share schedulers use time slices of fixed length. For these, the overhead is constant. Other, variable quantum schedulers (for example WRR, see section 4) use different quantum lengths for their tasks. For each individual task though, the quantum is fixed, so each task has an individual but constant overhead in this case.

2.2 Estimating context switch overhead

Unlike scheduling overhead, context switch overhead does not manifest itself as a distinct time window. Instead, the CPU appears to be "slowed down" following a context switch, i.e. the currently active task progresses slower than it would if there had not been a context switch because cache load, cache write-back and TLB refill operations are taking place in parallel to program execution. The extent of the slowdown depends on the state of the CPU, the state of the task, and on its program code. If all of this detailed information were available, a model of the system could predict the cache behaviour. But such an approach is seldom feasible: the scheduler usually does not have sufficient knowledge about its clients. Moreover (considering the use case of a VMM), the activities of the clients and those of the scheduler are generally not correlated. Thus, a context switch may interrupt a client at arbitrary points, and the cache behaviour would have to be evaluated for every conceivable state of each client. Even for small sets of client programs this problem is intractable.

Therefore, in this work, an empirical approach is attempted: we start by reasoning about the behaviour of the caches during a context switch, defining formal models to approximate the expected behaviour. Then, by experimentally measuring the actual progress rate over time for a number of test scenarios, we derive parameters that are subsequently used to calibrate the models so they can reproduce the real-world dynamic behaviour. These findings are then used in a simulation of proportional share scheduling to estimate the expected system overheads. Of course, this heuristic approach can not be expected to deliver exact or even provably correct results. Nevertheless, pursuing this path is considered worthwhile, since its results will likely be closer to reality than those of otherwise correct approaches which are based on the neglection of system overheads.

Model of cache behaviour

Figure 2 shows a possible development of a task's progress rate following a context switch. The task is being switched to at time $t = 0$. Clearly, the worst case situation with respect to cache effectiveness is the case known as "cache thrashing": A program accesses its memory resident objects in such an unfortunate pattern, that every object gets evicted from the cache before being used a second time. Thus, every access to a memory object causes a cache miss. The program does not gain any speed from the cache: it never finds any useful contents in it. Its progress rate is therefore minimal (r_{min}) [2]. Conversely, the best case situation oc-

[2] However, because of the on-demand nature of caches, this progress rate must be greater than zero: cache loading only takes place as a side-effect of code execution, i.e. there is no cache loading activity without program progress.

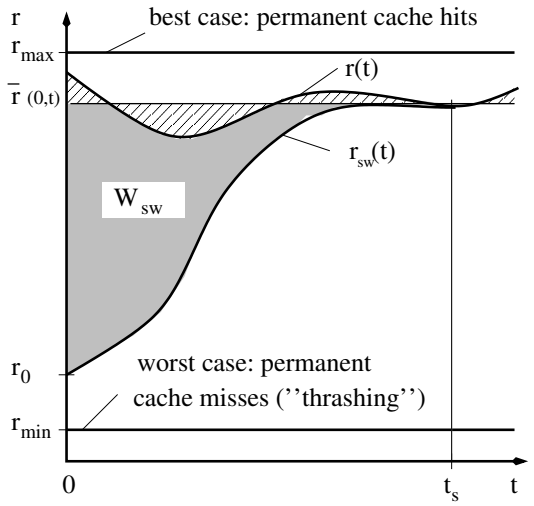


Figure 2. Progress rate after a context switch.

curs when a program either (a) finds all of its objects readily loaded in the cache (e.g. because other tasks have not evicted them), or (b) does not use any memory resident objects at all. In both cases, the program executes at a the same progress rate $r(t)$ that it would also have had if there had not been a context switch at all. In between those two extremes, a realistic case would be that a task executes at a low rate (r_0) initially. This initial progress rate depends on the amount of useable contents already in the cache at time $t = 0$, so, in the worst case, the situation is equal to that of "cache thrashing" and the minimum possible initial progress rate is $r_0 = r_{min}$. As the program executes and more and more of its memory-resident objects are being loaded into the cache, the progress rate will gradually increase until, at time $t = t_s$, the original progress rate $r(t)$ is reached. At time $t = q$, the scheduler is invoked again. If the quantum is short ($q < t_s$), the original progress rate can not be reached within the time slice. In this case, if the scheduler switches to another task, the cache contents loaded so far are lost again. However, if the task is continued, the cache loading continues where it left off.

The amount of work that the task would have had done during a time interval $[0, t]$ if not interrupted, is $W_{cpu}(0, t) = t \cdot \bar{r}(0, t)$ according to equation (1). With $r_{sw}(t)$ describing the task's actual progress rate after a context switch as a function of time, the amount of service that the task actually has received at time t since the context switch at time $t = 0$ is:

$$W_{task}(t) = \int_0^t r_{sw}(\tau)d\tau \tag{4}$$

Thus, the amount of service that is lost as a result of the context switch, is the difference:

$$W_{sw}(t) = t \cdot \bar{r}(0, t) - \int_0^t r_{sw}(\tau)d\tau \tag{5}$$

This amount of lost service corresponds to the area between the curves in figure 2. The relative context switch overhead is the ratio between this lost service and the total amount of service that the CPU has delivered during the time interval $[0, t]$:

$$O_{sw}(t) = \frac{W_{sw}(t)}{W_{cpu}(0, t)} = 1 - \frac{1}{t}\int_0^t \frac{r_{sw}(\tau)}{\bar{r}(0, t)}d\tau$$

The CPU can be thought of as being shared between two activities: The actual *payload task* that it executes and an imaginary *cache task* maintaining the caches. We define the *payload share*, $f(t)$ as the ratio between a task's current progress rate and the average progress rate over interval $[0, t]$ that it assumes when not interrupted:

$$f(\tau) := \frac{r_{sw}(\tau)}{\bar{r}(0, t)} \tag{6}$$

With this, we get:

$$O_{sw}(t) = 1 - \frac{1}{t}\int_0^t f(\tau)d\tau \tag{7}$$

Equation (7) yields the context switch overhead of a task at time t after a single context switch at time $t = 0$. For $t \geq t_s$, that is, when all needed objects have been loaded into the cache, the progress rate is the same as if there had not been a task switch. Assuming that $f(t) \approx 1$ in this case, if the task is run for one quantum $q \geq t_s$, the context switch overhead is:

$$O_{sw} = \frac{t_s}{q} - \frac{1}{q}\int_0^{t_s} f(\tau)d\tau = \frac{1}{q}\int_0^{t_s} 1 - f(\tau)d\tau \tag{8}$$

Note that for large values of q, the overhead becomes negligible. Note also that in both the best case (i.e. no cache loading at all) as well as the worst case (i.e. permanent cache thrashing), $f(t) = 1 \,\forall\, t$. Thus, in both cases, the corresponding context switch loss is zero. While this may seem confusing, especially for the cache thrashing case, it does make sense: a cache thrashing program does not draw any benefit from the cache, so it does not lose anything when the cache contents are lost in a context switch. Thus, regarding context switch cost, cache thrashing is in fact not a worst case.

Generally, in order to exactly quantify the context switch overhead, the development of the payload share as a function of time, $f(t)$, would be needed. If such a function were known for a given configuration, equations (7) or (8) could be used to compute the overhead per quantum allocation, which could then be used, e.g. in a scheduler simulation, to compute the overall average overhead for the simulated scheduler. However, $f(t)$ depends heavily on the actions taken by the particular program that is executed, so there is no generic case. Nevertheless, it is possible to identify best and worst case situations. Also, a sensible average case approximation lying in between can be suggested. Of course, this average case can never be guaranteed, but it is still useful to estimate the overhead that a system is likely to exhibit.

Since equations (7) and (8) deliver the overhead per quantum (where each individual quantum is associated with a particular task), the overheads can also be attributed to tasks, and different payload share functions can be applied to reflect each task's individual time criticality. A VMM, for instance, may have to schedule several virtual machines hosting subsystems with differing timing requirements (e.g. soft/hard real-time, non-real-time). In this situation, the scheduler can adapt its assumptions to each client individually: For a hard real-time client, it would assume a worst case scenario to be on the safe side, whereas for soft or non-real-time clients, it would use a less pessimistic average case function to compute the expected overhead.

As mentioned, in figure 2, the overhead corresponds to the area between the curves of $r(t)$ and $r_{sw}(t)$. The area between $r_{sw}(t)$ and the average progress rate $\bar{r}(0,t)$ has the same size. To construct a situation that yields the worst context switch cost, this area must be maximised, which is achieved for a rectangular shape: $r_{sw}(t)$ would have to be constantly minimal for some time, and then jump to the maximum rate. This is achieved by a "cache flooding" program that permanently accesses a limited set of memory resident objects small enough to fit entirely into the cache. If none of the task's objects are in the cache, such a program runs at the same minimal progress rate as a thrashing program until all objects have been cached. From that point on, it runs at the maximum sustained progress rate:

$$f_{flood}(t) = \begin{cases} f_0, & 0 \leq t < t_s \\ 1, & t \geq t_s \end{cases} \tag{9}$$

Inserting $f_{flood}(t)$ into equation (8) yields:

$$O_{flood} = \frac{t_s}{q} \cdot (1 - f_0) \tag{10}$$

Thus, the relationship between context switching overhead and the quantum in this worst case scenario is characterised by just two parameters, which can be obtained by experimentation:

- t_s: The time by which the task has loaded all of its memory objects into the cache. This time should be related (e.g. proportional) to the total size of these objects. In the rest of this paper we refer to this amount as the the task's "work space size"[3].
- f_0: The ratio between the lowest and the average progress rate of the program. This will depend heavily on the code of the program itself (relative frequency of memory accesses within the program code).

The assumption of a cache flooding task will generally deliver pessimistic results. It is useful for estimating the scheduling overhead for hard real-time tasks, but for soft real-time problems where the occasional deadline miss is acceptable or for tasks that are not subject to any timing restrictions, a less pessimistic assumption would deliver better results. In natural sciences there are many examples of processes which show an exponential decay behaviour (e.g. radioactive decay, or the transfer of thermal energy between bodies of different temperature). Thus, using an exponential decay function to describe the temporal behaviour of cache overhead seems reasonable:

$$f_{avg}(t) = 1 + (f_0 - 1) \cdot e^{-kt} \tag{11}$$

where:

$$k = \frac{1}{t_s} \cdot ln\left(\frac{1 - f_0}{\epsilon}\right)$$

This function converges towards the average progress rate (i.e. $\lim_{t \to \infty} f_{avg}(t) = 1$), but does not reach it in finite time. Thus, t_s is defined here as the time by which $f_{avg}(t)$ approximates the average progress rate by an error of ϵ, i.e. $f_{avg}(t_s) = 1 - \epsilon$. Inserting $f_{avg}(t)$ into equation (7) for $t = q$ yields:

$$O_{avg} = \frac{f_0 - 1}{k \cdot q} \cdot e^{-kq} \tag{12}$$

This relationship between context switching overhead and the quantum is characterised by similar parameters as the worst case scenario of the cache flooding task: Again, t_s can be assumed to be related to the task's work space size, while f_0 will -in the worst case- be the same as in the case of cache flooding, or higher, if the task can be expected to find useable contents in the cache.

3 Measurements

A series of measurements was made which were designed to reproduce the worst case behaviour with respect

[3]We explicitly avoid the term "working set" here as it is often used to refer to the set of *pages* actively used by a program.

to context switch cost (i.e. "cache flooding") as described in the previous section. The goal of these measurements is twofold: Firstly, it needs to be shown that the introduced model is actually applicable to reality, secondly, some realistic values for the parameters in equations (10) and (12) are to be collected.

3.1 Methodology

A method is needed to determine the development of a task's progress rate over time. A straightforward way to state "progress" would be to measure the average number of machine instructions executed by the processor per time unit. This is problematic, though, because instructions are neither uniform in the amount of "useful work" they do, nor in the amount of time (or number of machine cycles) that they take to execute. There is no such thing as an "average instruction" for any given machine architecture, let alone one that would allow comparisons across different architectures. However, to quantify switch overhead, it is not really necessary to measure absolute progress: The overhead depends on the payload share function, i.e. the ratio between a task's slowest progress rate and its average (sustained) progress rate.

Thus, rather than single instructions, short sequences of test code are measured which are designed to perform comparable uniform amounts of work, such as to store or to load a certain amount of data to or from memory. The test code sequences are executed in tight loops, so, the smallest unit of progress that can be measured in this way is one execution of the loop. Also, because of the limited resolution of the available timer hardware and because of the additional CPU effort involved in reading the timer value, the measurement works by executing a programmable number of loops and then measuring the time taken for this number. The result of each such measurement is thus not the *rate* of progress, but the *amount of service* accumulated over the measured time interval (i.e. the integral over the rate). From, this, the desired progress rate can be computed as the derivative with respect to time.

The applied measurement method is a simplified variant of the one used by John and Baumgartl ([6]): Prior to executing the test code, the CPU's caches are prepared: They can be left untouched, invalidated, or filled with irrelevant data, either by reading or by writing[4]. After conditioning the caches in this way, a selectable test code is executed for a programmable number of loops. The time taken to execute the test code is then returned as result. This sequence is repeated for different loop counts. The selectable test codes either read or write a region of (not previously cached) scratch memory, the size of which can be configured. This size is thus the total amount of data that the program loads into the cache, i.e. the program's "work space size". The sequence of the addresses that are read or written can be chosen to be either random or to access consecutive cache lines. The latter is intended to reproduce worst case behaviour (i.e. cache misses until all of the work space are in the cache).

Two different IA-32 machines running Linux were used as test platforms. To prevent unwanted interruptions of the test codes, they had to be run in privileged mode with disabled interrupts, which is why the test code was implemented as a kernel module in Linux.

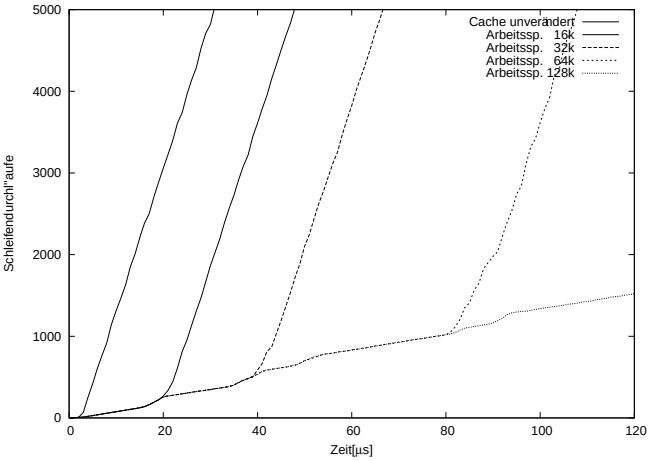


Figure 3. Program progress after a cache fill.

Figure 3 shows the exemplary results of five measurements made with different work space sizes. In the first case (labelled "Cache untouched"), the cache was not changed prior to the test. In the other four cases, the caches were write-filled initially. Then, consecutive addresses were written to, up to the indicated work space sizes. The curves measured for 16k, 32k and 64k work space size follow a similar pattern: The number of executed loops increases slowly at first, until reaching a point where all of the selected work space has been loaded into the cache. At this point, speed increases drastically. The time when this transition occurs, depends on the selected work space size. In the case of 128k work space size, the transition does not occur within the shown measurement range. In the "Cache untouched" case, no transition occurs: the speed is constantly high from the beginning.

The curves shown in figure 4 have been computed from those in figure 3 as the derivative with respect to time. Thus, since figure 3 shows the progress made by the programs over time, this figure now shows the progress *rate* as a function of time.

[4]The additional method of cache "flooding" used by John and Baumgartl was not implemented here: Flooding can be used to produce an absolute (but not very realistic) worst case situation that is specific to the IA-32 architecture's two-level cache structure.

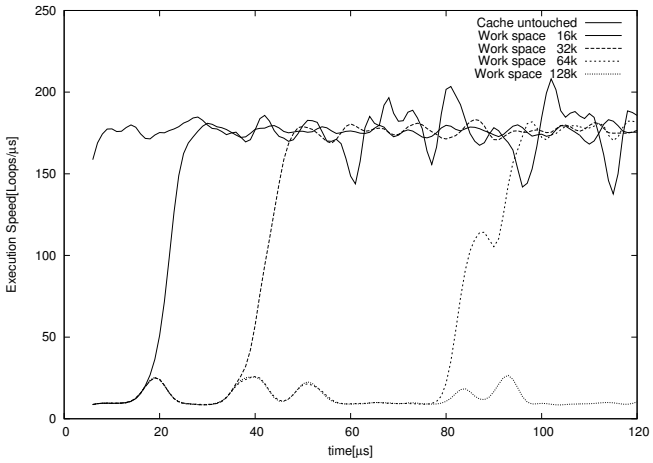


Figure 4. Progress rate after a cache fill.

3.2 Discussion of results

All measured curves follow similar patterns: Execution speed is low at first, then it jumps abruptly and continues at a higher, sustained progress rate. In all cases (except for the 128k work space case), the average sustained execution range that is assumed after the jump, is the same. These measurements have been repeated on different machines, for different work space sizes, cache preparation methods and memory access patterns.

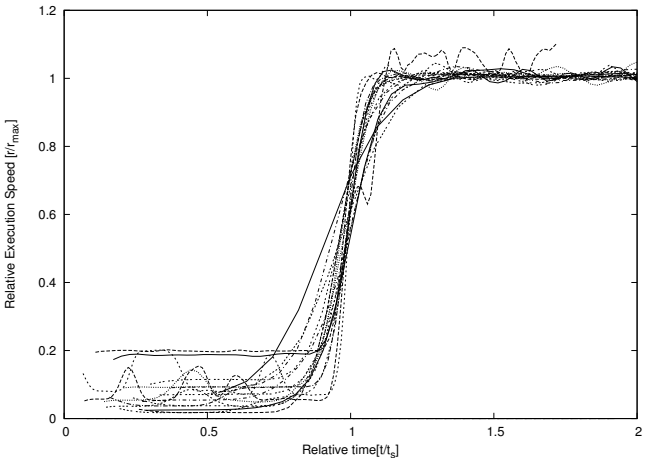


Figure 5. Normalised progress rates.

Figure 5 shows all curves that where measured with the "consecutive cache lines" addressing pattern. In this picture, all progress rate values have been normalised by the average sustained progress rates. Thus, the vertical axis in figure 5 indicates the payload share as per definition in equation (6). Also, the horizontal (time) axis is normalised by the individual times at which the transition to the higher

Machine	Access	WSS	Cache	f_0	t_s
A	cw	16k	fill	0.11	$22\mu s$
A	cw	32k	fill	0.11	$43\mu s$
A	cw	64k	fill	0.09	$85\mu s$
A	cw	16k	invd	0.17	$12\mu s$
A	cw	32k	invd	0.18	$20\mu s$
A	cw	64k	invd	0.20	$35\mu s$
A	rw	16k	fill	0.26	$30\mu s$
A	rw	32k	fill	0.36	$38\mu s$
B	cw	16k	fill	0.09	$25\mu s$
B	cw	32k	fill	0.08	$38\mu s$
B	cw	64k	fill	0.12	$91\mu s$
B	cw	16k	invd	0.17	$16\mu s$
B	cw	32k	invd	0.12	$29\mu s$
B	cw	64k	invd	0.21	$55\mu s$
B	rw	16k	fill	0.26	$30\mu s$
B	rw	32k	fill	0.36	$38\mu s$

A = Celeron @ 2.5 GHz, B = Pentium M @ 1.5 GHz
cw = consecutive write, rw = random write

Table 1. Results of measurements.

progress rate was observed. The figure demonstrates that the behaviour in all measured cases is similar: The payload share is low initially and stays low until all of the work space are in the cache. At this point it jumps (more or less abruptly) to assume a (roughly) constant high value. This is the behaviour that is to be approximated by a function like $f_{flood}(t)$ as per equation (9). The initial (low) progress rate varies between the measured cases. It depends on the method of cache preparation that was applied: if the cache is write-filled, the progress rate is lower, because the "dirty" cache entries have to stored back in memory.

Table 1 shows values for the ratio between the minimum and average observed progress rates, f_0 and the time t_s at which the sustained execution rate is reached. These values have been computed from the measured progress rate curves. There is a clear dependency between t_s and the work space size. On the Celeron machine, it is approximately proportional as one would have expected. On the Pentium M, however, it also increases with work space size, but not proportionally. Also, the progress rate ratio is not constant for measurements which used identical code and cache preparation methods. This is because the absolute average progress rates measured on the Pentium M are much higher for small work space sizes. These unexpected observations are suspected to be related to the IA-32 architecture's complex two-level cache structure: the Pentium M has larger caches than the Celeron, and, if the work space of a test is small enough to fit entirely into the level 1 cache, the program runs faster than if it only fits into the larger level 2 cache. This assertion, however, will have to be cross-

checked by performing the tests on a different architecture with a less complex cache structure.

In summary, although some of the experimentally obtained values clearly need a more thorough investigation, it can be said that the behaviour of the "cache flooding" case introduced in section 2.2 has been reproduced in the experiment. The values shown in table 1 can thus be used as parameters to the cache flooding model given by equation (10) to approximate real-world behaviour.

4 Computing overheads by simulation

As mentioned in the introduction, proportional share schedulers are a particularly interesting target for studying the effects of switching overheads on their performance. From the large number of proportional share scheduling algorithms that have been proposed in the literature, two specific ones have been selected for simulation. The reason for choosing these two algorithms was mainly because of their property of constant run-time scheduling cost[5]:

Weighted round robin (WRR) scheduling is probably the simplest known proportional share scheduling algorithm: It executes its tasks cyclically for the duration of individual quanta in a round-robin fashion. Each task's quantum is scaled according to its weight:

$$q_i = Q \cdot \frac{w_i}{\sum_j w_j} \tag{13}$$

Where Q is the sum of all quanta $\sum_i q_i$. Tasks are arranged in a circular queue. Each time the scheduler is invoked, it selects the next task from the queue and executes it for the duration of its quantum q_i. Thus, every scheduler call leads to a task switch. The schedule is periodic with the period being equal to the sum of all task's individual quanta, Q, plus the time spent for scheduling in between the quanta.

In contrast to WRR, most other proportional share schedulers use a fixed quantum. When the scheduler is invoked, it evaluates the status of all tasks and decides which of them should receive the next quantum. Several practical algorithms have been proposed (e.g. [2, 13, 10]) which differ in their methods of collecting task status data and the way they make their scheduling decisions. For this simulation, the *Virtual Time Round-Robin* (VTRR) scheduler [10] was chosen as example.

VTRR works by sorting all tasks by their individual weights and executing them in a round-robin fashion, each for the duration of a quantum. Whenever a task is found to have accumulated more than its ideal share of service, the scheduler jumps back to the beginning of the run queue, skipping the remaining tasks. The task selected by the scheduler may be the same that was active before, so unlike WRR, a scheduler invocation does not always lead to a task switch.

For fixed-quantum schedulers, the quantum, q, is the parameter defining the time scale for the system. The equivalent parameter for a variable-quantum scheduler would be the sum of all individual quanta, Q. When comparing fixed-quantum schedulers with variable-quantum schedulers, a provision must be made how to map a given q to an equivalent Q (and vice versa). An obvious choice would be to define $Q = N \cdot q$, where N is the number of tasks. However, for uneven weight distributions (i.e. large range of weights), this leads to minimal quantum values for the variable-quantum scheduler which may be unrealistically small with respect to the (fixed) time needed to execute the scheduler: When considering scheduling overheads, the relation between the smallest of quanta and the scheduling time plays a central role. Therefore, for all comparisons made in this work, a mapping between q and Q was chosen such that the smallest of the individual quanta q_i used by the WRR scheduler equals the fixed quantum q used by VTRR.

A program was implemented to simulate two selected scheduling algorithms (WRR and VTRR). Input to this program are the number of tasks, their individual weights, the quantum duration (q), the time taken for a scheduler invocation (s) and a payload share function. The two payload share functions $f_{flood}(t)$ and $f_{avg}(t)$ given by equations (9) and (11) are built into the simulator, but it also allows arbitrary functions to be specified as a list of value pairs. Thus, experimentally obtained functions can be fed into the simulator as well. The simulator runs the specified system (virtually) for a given amount of time. While running, it collects data about the various accumulated overheads.

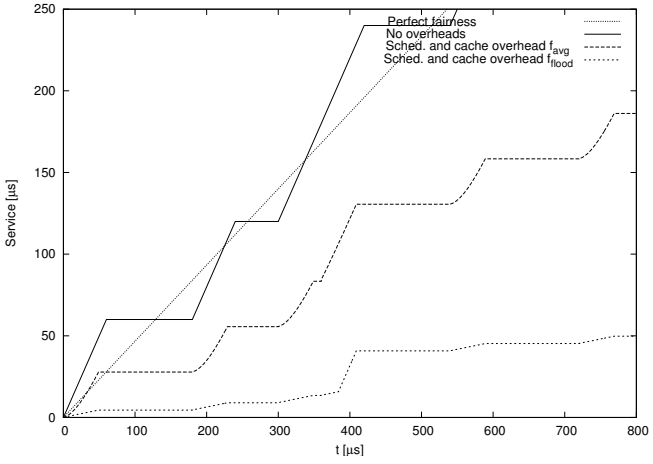


Figure 6. Simulated VTRR service allocation with and without overheads.

[5]Note that, since we only consider static systems, the cost of run-time insertions/removals of clients need not be considered

As an example, figure 6 shows the amount of service allocated to a task by a VTRR scheduler. The line marked "No overheads" indicates the service when neither scheduling nor task switching cost are accounted for. The line marked "Perfect fairness" shows the idealised continuous assignment of service that the scheduler approximates. The lines marked "Sched. and cache overhead f_{avg}" and "Sched. and cache overhead f_{flood}" show the amount of service that is actually available to the task after subtracting the overheads of scheduling and task switches. For this example, a scheduler execution time of $s = 10\mu s$, and a quantum duration of $q = 50\mu s$ were chosen. The figure shows how the actual resource allocation is significantly lower than the idealised one [6]. At $t = 350\mu s$, the figure shows a case where the scheduler is invoked but does not switch tasks: The task stalls while the scheduler executes, but when resumed, it continues at the same rate as it did just before the interruption. Either the average function $f_{avg}(t)$ or the cache flooding function $f_{flood}(t)$ [7] was used as payload share function. The curves show how the resource allocation increases linearly for the cache flood case and how it converges towards linear increase in the average case.

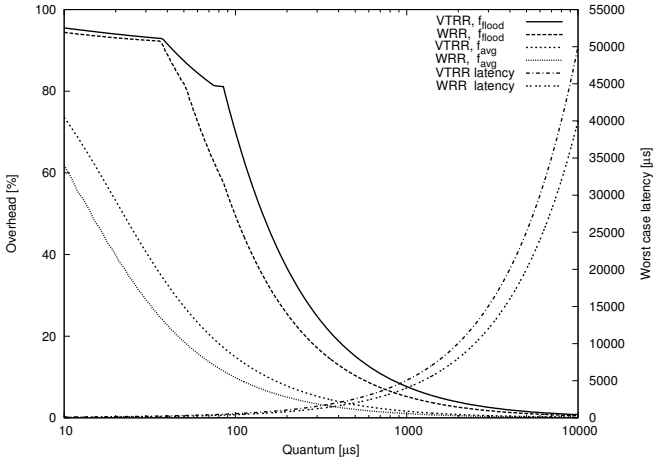


Figure 7. System overhead and worst case task latency as a function of quantum.

Figure 7 shows the overall system overhead computed for a system with three tasks (weights: 3, 5 and 7) for the WRR and the VTRR scheduling algorithms. For the two lines marked "f_{flood}", the cache flooding function and the parameters measured on the Celeron machine for a 64K work space (i.e. third line in table 1) were used to compute the cache overhead. For the other two lines, marked "f_{avg}", the average function with the Celeron parameters for a 16K work space (first line in table 1) was used. The quantum ranges from 10 μs to 10 ms (note the logarithmic scale). The picture indicates that for quantum sizes in the range typically used by contemporary general purpose operating systems (i.e. one to ten milliseconds), the system overhead is well below 10%, even under worst case assumptions. Thus, it may in fact be considered negligible for these systems. For sub-millisecond quantum sizes, though, the overhead increases significantly, especially under worst case assumptions. Figure 7 also shows the maximum latencies, i.e. the time intervals for which a task may not have access to the CPU. Remember, that for a real-time program hosted by a virtual machine, this value defines the jitter that such a program will exhibit. It does not depend on the payload share function, therefore, only two curves (for WRR and VTRR) are shown. The latency increases proportionally with the quantum size and the number of tasks. To achieve a low jitter for real-time programs, a small quantum has to be chosen which inevitably leads to high system overhead. In the shown example, if a real-time client can accept a 5 millisecond jitter, the quantum must be chosen no larger than 1 millisecond, resulting in a worst case overhead of 7.6%.

[6] Admittedly, the quantum chosen for this example is extremely short. This was done to show the detailed progression of the allocated resources over time. With a less extreme quantum setting (and the appropriate scaling), all curves would appear as lines with different slopes.

[7] See equations (11) and (9)

W_i	Sch	$q[\mu s]$	$WSS/f()$	$O_1[\%]$	$O_2[\%]$	$O_3[\%]$
2/5/8	wrr	10	16k/avg	76.48	48.51	34.43
2/5/8	vtrr	10	16k/avg	76.48	76.48	67.27
1/1/1	n/a	10	16k/avg	76.47	76.48	76.48
2/5/8	wrr	10	64k/flood	95.50	93.57	92.80
2/5/8	vtrr	10	64k/flood	95.50	95.50	95.50
1/1/1	n/a	10	64k/flood	95.50	95.50	95.50
2/5/8	wrr	100	16k/avg	15.66	6.64	4.22
2/5/8	vtrr	100	16k/avg	15.69	15.70	13.21
1/1/1	n/a	100	16k/avg	15.67	15.70	15.69
2/5/8	wrr	100	64k/flood	79.40	33.60	21.31
2/5/8	vtrr	100	64k/flood	79.41	79.41	53.09
1/1/1	n/a	100	64k/flood	79.41	79.41	79.41
2/5/8	wrr	1000	16k/avg	1.71	0.69	0.43
2/5/8	vtrr	1000	16k/avg	1.71	1.72	1.44
1/1/1	n/a	1000	16k/avg	1.69	1.72	1.71
2/5/8	wrr	1000	64k/flood	8.65	3.48	2.18
2/5/8	vtrr	1000	64k/flood	8.65	8.66	5.82
1/1/1	n/a	1000	64k/flood	8.64	8.66	8.65

Table 2. Per task overheads.

Table 2 lists per-task overheads for a number of parameter combinations. This demonstrates that the overheads are not necessarily distributed according to the weights of the tasks. In this example, identical payload share functions where used for all tasks, however, our simulator would also allow to use different functions for each individual

task. Note that for uniform weight distributions (i.e. $W_i = 1/1/1$), VTRR and WRR deliver the same schedule and thus identical results. Therefore, they are not listed separately in the table.

5 Conclusion and further work

In this work, we described a method to estimate scheduling and switch overheads for multi-tasking systems. The method is based on a combination of simulation and experimentation. For two selected proportional share scheduling algorithms, our simulator can estimate the overall system overhead as well as separate context switch and scheduler overheads associated to individual tasks. The results indicate that with contemporary hardware, sub-millisecond quantum size, which would enable acceptably low jitter for real-time programs, can only be achieved by sacrificing a significant portion of the system's computational resources. For our use case of real-time programs hosted by virtual machines, this means that, as long as proportional share scheduling is used to switch between VMs, we can not obtain competitive real-time performance. Thus, VMMs will have to adopt different scheduling strategies if they are to support real-time applications (Some suggestions are presented in [8]).

Measurements, so far, have only been made for two different IA-32 platforms, thus, a future activity will be to apply these measurements to a broader selection of architectures featuring different cache structures.

While the scenario of the worst case payload share function has been reproduced experimentally, the same can not be said about the "average" payload share function: At this time, this function is purely artificial. To construct an experiment that will allow to measure cache overheads under real-world computational loads will be more difficult than to reproduce the worst case as we have done. A possible approach could be to integrate a configurable fixed timeslice scheduler into an already existing VMM (e.g. Xen) and perform similar measurements while the system runs a selection of characteristic benchmarks. Any payload share functions derived in this way can then be integrated into our simulator to improve its estimations.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. H, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization, 2003.

[2] J. C. R. Bennett and H. Zhang. WF^2Q: Worst-Case Fair Weighted Fair Queueing. In *Proceedings of INFOCOM'96, San Francisco, CA*, pages 120–128, Mar 1996.

[3] G. C. Buttazzo. Rate monotonic vs. EDF: judgment day. *Real-Time Syst.*, 29(1):5–26, 2005.

[4] L. Cherkasova, D. Gupta, and A. Vahdat. Comparison of the three CPU schedulers in Xen. *SIGMETRICS Perform. Eval. Rev.*, 35(2):42–51, 2007.

[5] K. J. Duda and D. R. Cheriton. Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 261–276, New York, NY, USA, 1999. ACM Press.

[6] T. John and R. Baumgartl. Worst Case Behavior of CPU Caches. 6th Real-time Linux Workshop, Lanzhou, China, October 2006.

[7] R. Kaiser. Scheduling Virtual Machines in Real-time Embedded Systems. OSPERT 2006, Dresden, Germany, 2006.

[8] R. Kaiser. Alternatives for scheduling virtual machines in real-time embedded systems. In *IIES08 - 1st EuroSys 2008 ACM SIGOPS Workshop on Isolation and Integration*, Glasgow, Scotland, April 2008.

[9] J. Liedtke. Improved Address-Space Switching on Pentium Processors by Transparently Multiplexing User Address Spaces. Technical Report 933, Nov. 1995.

[10] J. Nieh, C. Vaill, and H. Zhong. Virtual-Time Round-Robin: An O(1) Proportional Share Scheduler. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 245–259, Berkeley, CA, USA, 2001. USENIX Association.

[11] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Trans. Netw.*, 1(3):344–357, 1993.

[12] A. Singh and K. Jeffay. Co-Scheduling Variable Execution Time Requirement Real-Time Tasks and Non Real-Time Tasks. In *ECRTS '07: Proceedings of the 19th Euromicro Conference on Real-Time Systems*, pages 191–200, Washington, DC, USA, 2007. IEEE Computer Society.

[13] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke, and C. G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *RTSS '96: Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96)*, page 288, Washington, DC, USA, 1996. IEEE Computer Society.

[14] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14. USENIX Association, 2001.

[15] C. A. Waldspurger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Operating Systems Design and Implementation*, pages 1–11, 1994.