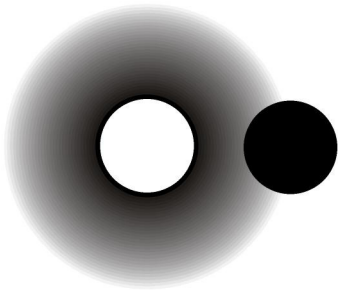


OSPERT 2007

Workshop on Operating Systems Platforms for Embedded Real-Time applications

Workshop Chairs: Scott Brandt and Kevin Elphinstone

National ICT Australia
223 Anzac Parade
Kensington NSW 2052 Australia
Technical Report July 2007

ISSN 1833-9646

National ICT Australia 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.

Table of Contents

Foreword

Providing an operating system platform for time sensitive applications is a challenge. The *Workshop on Operating Systems Platforms for Embedded Real-Time applications* provides a forum for researchers in the area to present and discuss the many issues related to real-time operating systems, including topics such as scheduling, quality of service, component-based development, support for multiprocessors, real-time Linux, and power management.

The workshop attracted high quality submissions. The submitted papers were peer reviewed by an expert program committee. We wish to express our gratitude to the members of the program committee who contributed their time to provide high quality reviews, which eased the job of selecting a quality program.

The resulting program provided the basis for many interactive and insightful discussions during the day, which was a valuable experience for all involved.

The authors were given the opportunity to revise and re-submit final versions of their papers based on the reviews, and on the discussions that developed at the workshop. The papers contained herein are the final versions submitted after the workshop.

Kevin Elphinstone and Scott Brandt
Workshop Chairs

Program Committee

Neil Audsley, University of York, UK
Scott Brandt, University of California, Santa Cruz, USA
Kevin Elphinstone, University of New South Wales, Australia
Gerhard Fohler, University of Kaiserslauten
Michael Hohmuth, AMD Operating Systems Research Center, Germany
Giuseppe Lipari, Scuola Superiore Sant'Anna, Italy
Daniel Mossé, University of Pittsburgh, USA
Stefan Petters, National ICT Australia
Krithi Ramamritham, Indian Institute of Technology, Bombay
Ismael Ripoll, Universidad Politecnica de Valencia, Spain
Andrés Terrasa, Technical University of Valencia, Spain

Fitting Linux Device Drivers into an Analyzable Scheduling Framework

[Extended Abstract]

Theodore P. Baker, An-I Andy Wang, Mark J. Stanovich *
Florida State University Tallahassee, Florida 32306-4530
baker@cs.fsu.edu, awang@cs.fsu.edu, stanovic@cs.fsu.edu

ABSTRACT

API extensions and performance improvements to the Linux operating system now enable it to serve as a platform for a range of embedded real-time applications, using fixed-priority preemptive scheduling. Powerful techniques exist for analytical verification of application timing constraints under this scheduling model. However, when the application is layered over an operating system the operating system must be included in the analysis. In particular, the computational workloads due to device drivers and other internal components of the operating system, and the ways they are scheduled, need to match abstract workload models and scheduling polices that are amenable to analysis. This paper assesses the degree to which the effects of device drivers in Linux can now be modeled adequately to admit fixed-priority preemptive schedulability analysis, and what remains to be done to reach that goal.

Categories and Subject Descriptors

D.4.7 [**Software**]: Operating Systems—*organization and design*; C.3.d [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*real-time and embedded systems*

General Terms

design, verification

Keywords

real-time, Linux, fixed-priority scheduling, preemptive, schedulability, device driver

1. INTRODUCTION

A huge amount of theoretical research has been done on real-time scheduling [26]. This theoretical foundation enables one to design a system that can be guaranteed to meet its timing constraints, provided the implementation adheres closely enough to the abstract models of the theory. More specifically, applying the theory requires that the system workload corresponds to models that have been studied, and that the system schedules the workload according to one of the algorithms whose performance on such workloads has been analyzed. Where a real-time system is implemented on top of an operating system, these requirements apply to all the OS components as well as the user-level code.

In Linux and several other POSIX/Unix-compliant [31] operating systems, progress has been made in providing real-time constructs so that user-level programmers can write applications that adhere to the theory of fixed-priority preemptive scheduling. Examples include preemptive priority-based real-time scheduling of user threads, high-precision software timers, and turning off virtual memory management for certain memory regions. Progress also has been made toward making the OS itself adhere more closely to analyzable models, including major reductions in non-preemptible sections within the kernel. Benchmark numbers on existing real-time Linux distributions, such as Montavista [22] and Timesys [32], suggest they now provide adequate capabilities to design and implement a wide range of hard and firm deadline real-time systems at the application level.

However, until recently, the role of device drivers in schedulability has not received much attention. Every operating system includes device drivers, which are responsible for low-level interactions with I/O devices. For embedded real-time systems, device drivers can be especially critical, in two ways. They can play a direct role in meeting throughput requirements and end-to-end deadlines that involve I/O, by the way in which they schedule I/O operations. Device drivers can also play a role in meeting timing constraints for computations that do not depend on I/O, through *interference*; that is, by blocking or preempting more time-critical computations. So, without well-behaved device drivers, the ability of a system to meet timing constraints may be limited to cases where input and output activities do not have deadlines or throughput constraints, and where there are no "storms" of I/O activity. While these are known facts, and while some techniques have been developed for managing I/O performance and device driver interference, integration of that work with Linux is far from mature, and more work remains to be done.

This paper reviews the remaining work to apply fixed-priority preemptive scheduling theory to Linux applications, including the effects of device drivers. It argues that some engineering problems remain to ensure that the interference effects of device drivers fit analyzable models, and to manage device driver scheduling to meet timing constraints, but that the scheduling theory seems adequate. Much larger problems remain with the analysis of I/O scheduling,

*This material is based upon work supported in part by the National Science Foundation under Grant No. 0509131, and a DURIP grant from the Army Research Office.

including device-specific real-time scheduling policies, and end-to-end schedulability analysis involving multiple resources.

2. FIXED-PRIORITY PREEMPTIVE SCHEDULING THEORY

This section reviews some general scheduling theory concepts and terms, and the basic workload models used in fixed-priority preemptive scheduling theory.

The goal of real-time scheduling is to ensure that, if an action is required to execute within a specified time interval it does so. The theory is expressed in terms of *jobs*, *execution times*, *release times*, and *deadlines*. In those terms, the goal is to ensure that each job receives its required execution time within its *scheduling window*, which is the interval between its release time and its deadline.

A job whose own execution time fits within its scheduling window will complete execution within the window unless it is prevented by *interference* from the execution of other jobs. Verifying that a job will be scheduled within its window requires a way to bound the interference, *i.e*, to bound the set of potentially competing jobs and the amount of time that the scheduler will allow them to execute within the window.

A *task* is an abstraction for a stream of jobs, which are ordinarily required to be executed serially with jobs of the same task. Restrictions on the execution times and release times of jobs within each task serve to bound the interference the task can contribute within the scheduling window of another task.

The most analyzed task model is the *periodic task*, in which a task τ_i is characterized by three parameters: the *worst-case execution time*, e_i, of its jobs; the *period*, p_i, between release times; the *relative deadline*, d_i, which is the length of each job's scheduling window. A relaxation of this model is the *sporadic task*, in which the period is interpreted as just a lower bound on the interval between release times. Much is known about the analysis of sets of periodic and sporadic tasks under various scheduling policies.

Fixed-priority preemptive scheduling is very well understood. This theory, including what is sometimes referred to as Generalized Rate Monotonic Analysis (*e.g.*, [15, 1]) and Response Time Analysis (*e.g.*, [3]) makes it possible to verify that a set of hard-deadline tasks will always meet their deadlines, that soft-deadline tasks will satisfy their average response time constraint, and that the execution time of a task may vary within a certain range without causing a missed a deadline.

The foundation of this analysis is a simple *interference bound*, observed by Liu and Layland [19], who showed that a collection of sporadic or periodic tasks causes the maximum amount of interference for a job of lower priority when the job is released together with jobs of all the higher priority tasks and each task releases a job periodically thereafter. It follows that that the interference due to a task τ_i in any interval of length Δ is bounded above by $e_i \lceil \Delta / p_i \rceil$.

Though initially limited to sets of independent preemptible periodic or sporadic tasks with fixed priorities, FP schedulability analysis has been extended to allow for *blocking effects*, due to locks protecting shared resources and brief intervals of increased priority or non-preemptibility due to other causes. In this broader context, there are two ways one task can interfere with another, namely *preemption interference*, based on having higher priority, and *blocking interference*, based on holding a non-preemptible resource that the other task must acquire before it can continue execution.

Fixed-priority preemptive scheduling analysis also has been extended to arbitrary (aperiodic) tasks by assuming that arriving jobs are queued and executed according to an *aperiodic server scheduling policy*. Several aperiodic server scheduling policies have been devised and studied, including the *polling server* [27], the Priority Exchange and Deferrable Server [29, 17], and the Sporadic Server [28]. Without considering the details of specific aperiodic server scheduling algorithms, one can see how they permit schedulability analysis by recognizing that they all enforce the following two principles:

1. Bandwidth limitation: There is an upper bound on the amount of execution time a task may consume (at a given priority) in a given length of time, analogous to the property of a periodic task that it never demands more than e_i time in each interval of length p_i. This permits computation of an upper bound on the amount of preemption interference the aperiodic task can cause for other tasks in an interval of any given length. For the Polling Server and the Sporadic Server with budget e_i the periodic task interference bound applies.

2. Priority bandwidth guarantee: There is a lower bound on the amount of execution time that a thread can rely on being allowed to contend for at a given priority in a given length of time, also analogous to a periodic task. This can generally be translated into a guaranteed average response time to real-time events, and sometimes used to validate hard timing constraints.

3. FIXED-PRIORITY PREEMPTIVE SCHEDULABILITY IN LINUX

This section reviews Linux facilities that support the design and implementation of real-time applications to fit the theory of fixed-priority scheduling, and discusses how well the implementation matches the theory.

Based on the extensive body of knowledge about fixed-priority preemptive scheduling, POSIX/Unix [31] operating systems standards adopted support for scheduling threads at fixed priority (*SCHED_FIFO* and *SCHED_RR*) and via a variation on the Sporadic Server policy (*SCHED_SPORADIC*). Several off-the-shelf operating systems provide support for these policies. Linux currently provides support for the *SCHED_FIFO* and *SCHED_RR* policies. So far, support for the *SCHED_SPORADIC* policy has only been reported in experiments [18], but it will probably eventually appear in Linux distributions.

Application of fixed-priority preemptive scheduling theory in the context of an OS that has no job or task abstractions requires translation between models. The POSIX/Unix API is expressed in terms of threads. A thread is a subprogram that may continue execution indefinitely, alternating between states of contention for execution and self-suspension. To apply the job and task model to a system composed of threads, one needs to treat each point at which a thread suspends itself (*e.g.*, to wait for a timed event or completion of an input or output operation) as the end of a job, and each point at which a thread wakes up from a suspension as the beginning of a new job.

Since the thread model does not constrain the intervals between job releases or the worst-case execution times between suspensions,

systems programmed with threads present problems for schedulability analysis unless some constraints are imposed on thread control flows and/or scheduling policies. Adequate constraints are enforced by the operating system in the case of the *SCHED_SPORADIC* policy, but guaranteeing that the threads scheduled by the *SCHED_RR* and *SCHED_FIFO* policies adhere to an analyzable release time and worst-case execution time model depends on programmer discipline.

Threads that perform input and output operations require additional consideration. If a thread makes blocking I/O requests, the intervals between job release times will depend on both the raw response time of the I/O device and how the system schedules it. For reasons explained in Section 8, the analysis of I/O scheduling, especially in combination with CPU scheduling, is much more difficult than the analysis of CPU scheduling, and is generally beyond the scope of fixed-priority preemptive scheduling theory. A way to work around this limitation is to move I/O out of time-critical threads, so that the CPU and I/O scheduling problems can be modeled and analyzed independently. In Linux, implicit I/O operations due to page fault activity can be avoided in time-critical threads by using *mlock()* and *mlockall()* to lock virtual memory pages accessed by those threads into physical memory. Explicit I/O operations can be moved out by buffering I/O data and either using asynchronous I/O requests, like *aio_read()*, or delegating the I/O to a separately scheduled server thread. Points at which a time-critical thread requires an I/O operation to be completed are deadlines for the I/O scheduler, to be analyzed separately. For example, consider a periodic thread that requires input and produces output, both to the same disk storage device. The input might be requested in advance, with the next task release time as deadline for the input operation, and the output might be buffered, with a deadline for the output operation several times longer than the task period. The scheduling problem is reduced to scheduling three independent periodic tasks, one using just the CPU, one doing disk reads, and one doing disk writes.

It is essential to bound blocking. Any work that is scheduled outside of the fixed-priority preemptive model is a potential source of blocking interference. For analysis to be successful, intervals of time over which a thread may prevent higher priority threads from preempting must have bounded duration. In particular, it is essential to avoid situations where a high-priority thread can wait for a mutex held by a low-priority thread, while a middle-priority thread executes. Linux provides a way to accomplish this, using mutexes with priority inheritance (*PTHREAD_PRIO_INHERIT*).

So far, it appears that the theory of fixed-priority preemptive scheduling can be applied to real-time systems that make use of the POSIX/-Unix thread scheduling policies under an operating system like Linux, provided the user designs the application to fit the models on which the theory is based. The set of real-time (highest) priority threads must be known. Each of them must either use *SCHED_SPORADIC* to limit its maximum high-priority computational demand or use *SCHED_FIFO* or *SCHED_RR* and be verified to fit a well-behaved workload model such as the periodic or sporadic task. In addition, attention must be paid to other details, such as bounding the length of critical sections, and determining bounds on worst-case execution times. All this may be difficult, but it is possible in principle since all of the code is under the user's control.

However, one must also take into account the code of the operating system, which is not directly visible or controllable by a user but may interfere with the schedulability. The OS must fit the models and constraints of the fixed-priority preemptive scheduling theory. Moreover, it is not enough that the OS admit analysis if the analysis does not eventually lead to an application that meets its timing requirements. The OS must admit analysis that is not overly pessimistic, and it must permit an application designer to actively manage priorities and workloads, within the OS as well as at the application level, to meet the timing requirements.

Of course Linux was not originally designed with these goals in mind, and it has since grown so large and complicated that the notion of attempting major architectural changes is daunting. For that reason, some individuals have given up on the idea of using a general-purpose OS like Linux directly as a platform for hard real-time applications, and developed a variety of layered schemes that provide greater control over timing (for example, RTLinux [4, 33], Linux/RK [23], RTAI [6], Hijack [24]). However, at the same time, others have worked to improve the real-time support of the Linux kernel itself (for example, [11, 12].

Probably the biggest improvement has been in bounding blocking effects due to critical sections within the OS. Ingo Molnar [21] introduced a set of high-preemptibility kernel patches, which greatly reduced the average blocking time due to kernel activities. Deriving an exact analytical upper bound for *worst case* blocking still does not seem practical, but an empirical bound can be obtained by measuring the release-time jitter of a periodic thread with the top real-time priority, over a long time and a variety of system loads. Such experiments for recent Linux releases with real-time patches show that blocking interference appears to be bounded [30]. However, in the absence of enforcement, through static or run-time checks, it is possible that a badly written system component could disable preemption for a longer time than observed in the experiments. Worse, unbounded blocking could occur through locking mechanisms, such as Linux kernel semaphores, that neither disable nor implement priority inheritance. Nevertheless, if the probability of blocking exceeding a given empirical bound (and so causing violation of an application timing constraint) can be shown to be low enough, that may be sufficient for many real-time applications.

Given that blocking interference due the OS is bounded, more or less, the remaining challenge is to bound preemption interference. After elimination of the easy cases, by scheduling the system daemons below the real-time priority level, it seems the remaining potential sources of interference by operating system components with the scheduling of application threads are in the system's device drivers.

4. DEVICE DRIVER INTERFERENCE

Device drivers include code that is scheduled in response to hardware interrupts. For example, consider a user task that makes a blocking call to the operating system to request input from a disk drive via a DMA interface. Typically, the driver would execute in three phases, more or less as follows:

1. The client calls the system, and the system calls the device driver. The device driver initiates an input operation on the device, and blocks the client thread until the input operation completes. The device driver code is scheduled as part of the client thread.

2. The device signals completion of the input operation, via an interrupt. An interrupt handler installed by the device driver performs various operations required by the device and input method, such as acknowledging the interrupt and perhaps

copying data from a kernel buffer to a buffer in the client thread's address space, then unblocks the client thread. The scheduling of the device driver code here is *interrupt-driven*.

3. Eventually, execution resumes in the client thread at the point in the device driver code where the client thread blocked. Control flows from the device driver to the kernel and from the kernel back to the user code. While the interrupt from the device plays a role in determining the release time of this phase, the device driver code is scheduled as part of the client thread.

Since the scheduling of interrupt-driven device driver code is outside the direct control of the application, the ability to analyze its effect on the ability of an application to meet timing constraints depends on the design decisions made in the device drivers and operating system kernel.

In popular processor architectures, the hardware schedules interrupt handlers at a priority higher than that of any thread scheduled by the OS[1]. Safe programming practice may also require that an interrupt handler executes non-preemptibly, with interrupts disabled.

In addition, many operating systems schedule interrupt-triggered device-driver code via a two-level mechanism. The Level 1 work, which is executed in interrupt context, is very short; in essence, it just records the fact that the interrupt has occurred, and enqueues the event on a list for later processing. The rest of the work is done at Level 2, in software event handlers.

In Linux, the Level 2 handlers are called *softirq* handlers, though they also go by other names, such "bottom halves" and "tasklets", and "timers". The softirq handlers are executed non-preemptively with respect to the thread scheduler and other softirqs on the same CPU, but with hardware interrupts enabled, in the order they appear in a list. The details of when these handlers are scheduled have changed as the Linux kernel has evolved. As of kernel 2.6.20 the responsibility is divided between two schedulers and priorities. Softirq handlers are executed by *do_softirq()*, which is called typically on the return path from a hardware interrupt handler. If there are still softirqs pending after a certain number of passes through the softirq list (meaning interrupts are coming in fast enough to keep preempting the softirq scheduler), *do_softirq()* returns. Responsibility for continuing execution of softirq handlers is left to be performed at background priority in a scheduled thread (called *ksoftirqd*), or the next time *do_softirq()* is called in response to an interrupt.

Both hardware interrupts and softirq's are intended to provide fast driver response to a particular external event, but can cause problems for schedulability analysis (see Section 6). They can also reduce overall system schedulability. Giving all interrupt-driven work higher priority than all work done by threads introduces a form of priority inversion, where an action that the theory says should logically have higher priority, in order meet its deadline, may be preempted by an action that logically should have lower priority. Executing handlers without preemption introduces another form of priority inversion, where a job that should have higher priority is not able to preempt a job that should have lower priority. Scheduling handlers non-preemptively also introduces a more subtle potential problem, giving up a property of preemptive scheduling that Ha and Liu [10, 9] call *predictability*. This kind of predictability is a necessary basis for schedulability analysis based on just worst-case execution times.

Effective scheduling and analysis requires that the use of mechanisms that are exceptions to the overall system scheduling model, such as hardware interrupts and Linux softirqs, be bounded in duration and frequency so that the overall interference they cause can be modeled. The OS can provide mechanisms for drivers to move work into preemptively scheduled threads (see Section 6), but without creative new architectural provisions for responsibility for bounding interrupt handler execution, it must rely on the designer of each device driver to make use of them.

Some interrupt-driven device driver execution presents special problems, due to I/O operations that are *device-driven*. For example, compare the input operations of an Ethernet interface device with disk input operations. An interrupt due to input of a message by a network device can occur spontaneously, due to the nature of an open system, where requests are generated from external sources. In contrast, an interrupt due to completion of a disk operation normally corresponds to a prior request from the kernel or a user thread, reflecting the characteristics of a closed system (overlooking the case in which a network request results in disk activity); so, the frequency of disk requests may be managed by the application, even if the precise timing of the completion interrupts cannot be predicted. Other kinds of input sources that may have device-driven interrupt-scheduled workloads include asynchronous serial ports, and streaming audio and video devices.

Since some portion of the device-driven workload is executed at high priority, it must be bounded before other work can be guaranteed any level of service. For some devices, such as a video device with periodic input behavior, this is not difficult. For other devices, such as an Ethernet interface, one seems to be forced to choose between bounds based on raw hardware capacity, which are unrealistically high, and bounds based on expected worst-case behavior of the communication partners, which cannot be determined by local analysis and may be unreliable. However, the kernel can help by providing aperiodic server scheduling mechanisms that can limit the CPU time spent on some of the interrupt-driven work of a device, as described below in Section 6. Well-designed device drivers may go further, by applying interrupt management techniques, as described in Section 7.

5. DEVICE DRIVER DEMANDS

Device drivers may have timing constraints, such as to stay synchronized with a device or to avoid losing data. For example, consider a video frame grabber (digitizer) device attached to a camera, which inputs video data continuously at a rate of 30 interlaced frames per second. The kinds of timing constraints such a driver might have would depend on the capabilities of the device.

If the frame-grabber requires programmed I/O – *i.e.*, it is not capable of acting as a direct-memory-access (DMA) bus master – the driver must use the CPU to read the device's on-board frame buffer. That will require very close synchronization. The device driver must read raster lines of pixels from the device fast enough to prevent loss of data when the device wraps around and overwrites one frame with the next. A driver designed to capture a full

[1]There are systems where interrupts can be assigned hardware priority levels and the CPU interrupt level can be varied, so that hardware interrupt levels can be interleaved with software priority levels. For example, this is possible with the Motorola 68xxx family of processors. It is not clear why this good idea has not been more widely adopted. Perhaps it is one of the many cases of patents on fairly obvious little ideas that impede real technological progress.

stream of video data from such a device may be viewed as a periodic task with 1/30-second period. It would have a tight release-time jitter requirement, and a deadline of perhaps half the period, to avoid risk of losing data. If the device generates an interrupt at a known point in each frame, execution of the driver can be driven by that interrupt, but it may need to use another interrupt (via a kernel timer abstraction) to schedule itself at a specified offset from the interrupt. If the device does not generate a frame synchronization interrupt, the driver would need to time its own execution, and the period would probably need to be regulated by the driver to stay in phase with the video source, using a phased-locked-loop control model.

If the frame-grabber is capable of DMA operation, the timing constraints on the driver can be relaxed by providing the device with multiple buffers. The driver may be able to program the device so as to choose which events cause an interrupt to be generated, such as when a new frame has been copied to memory, or when the number of free buffers falls below a threshold. The driver may then be modeled as two virtual tasks: a DMA task (implemented in hardware), and a video driver task (implemented in software) to manage the buffers and synchronization with the consumer of the video data. The DMA task would be periodic and would slow down the CPU by stealing memory cycles from other tasks. If the frame completion interrupt is used, the video driver task can be viewed as a periodic task. Its deadline and jitter requirements would be much looser than the case with programmed I/O, since the additional buffers allow the deadline to be longer than the period. If the threshold interrupt is used, it may be viewed as a sporadic task with a response-time constraint equal to the amount if time it takes the device to consume the number of buffers that is set as the threshold.

Device drivers can have a variety of internal timing constraints. Some cannot be expressed in terms of deadlines, because they are point-to-point within a computation that does not permit giving up control of the CPU. For example, in interactions between the CPU and device there may be a *minimum* delay for the device to process information from the CPU before it is able to accept the next command. If the delay is shorter than the precision of the kernel timer mechanism, achieving adequate throughput may require that the driver busy-wait. There are also point-to-point constraints that dictate non-preemptible execution, such as a *maximum* delay between actions in a sequence of interactions, beyond which the device goes into an error state that requires it to be re-initialized and the entire sequence to be restarted.

In general, device driver internal timing constraints must be validated along with other system timing constraints, and limit the measures that might otherwise be taken to reduce the interference that a device driver causes for other real-time tasks. However, some internal timing constraints that are treated as hard in the design of a driver might better be considered as soft in the context of a particular application. The usual device driver writer's perspective is to treat the needs of the device as top priority. In some cases that is wrong. For example, a device driver writer might decide to disable preemption rather than risk having to reset a device and lose time or data. In the context of an application where the overall function of the device is not time-critical, and some data loss is acceptable, this non-preemptible section might cause a more critical timing constraint to be missed. It is a challenge to design device drivers in a way that provides configuration mechanisms for an application designer to participate in resolving such trade-offs.

The scheduling of device driver execution often imposes a link between the quality of I/O service provided by the driver and the amount of interference the device driver causes for other tasks. It may be blocking interference, such as where disabling preemption within a driver prevents a recoverable device malfunction and loss of data. It may also be preemption interference, at the level of interrupt management and thread scheduling. That is, allowing a device to generate more interrupts or giving higher priority to a device driver thread may allow the device driver to respond to requests for attention from a device, and that may result in less idle time for the device, a shorter response time for the next I/O request to be processed, a higher overall throughput rate, and a reduction in lost data. The right balance in such trade-offs will depend on the application context, so mechanisms are needed for applications designers to configure the scheduling of device-driver execution.

6. DEFERRING WORK TO A THREAD

The less processing is done at hardware interrupt priority the shorter the potential duration of CPU priority inversion, and the better actual system scheduling fits the theoretical ideal. The Linux softirq mechanism might appear to help in this regard, by deferring some of the interrupt-triggered work, but it actually hurts. In contrast to the use of either pure interrupts (generally non-preemptible, top priority) or regularly scheduled (preemptible, lower priority) threads, this kind of complicated mixed scheduling mechanism is very difficult to model and analyze. If one ignores the role of the *ksoftirqd* server, softirqs might be modeled as a per-CPU thread that is scheduled at a fixed priority, lower than the hardware interrupt priority and higher than any other thread. A problem is that the workload of this thread, which is generated by many different kernel components for a great variety of purposes, does not conform to any analyzable model. The demotion of softirq service to be performed at background priority by *ksoftirqd* during heavy bursts of activity helps to bound this load for non-real-time purposes, but it is not done in a way that can be precisely modeled like the well-understood real-time aperiodic server scheduling algorithms.

Clearly, better schedulability can be achieved by moving the work of softirq handlers to one or more regularly scheduled threads. If a device is only used by non-real-time components of the system, the response time of the driver to interrupts will probably not be critical. (This assumption is the basis of the RTLinux [4] practice of deferring all of the Linux interrupt handlers to background processing.) In such cases, it is sufficient to schedule the device server in the background, at low enough priority not to preempt any of the threads that have timing constraints. An exception may be where the device is a bottleneck, but not necessarily, since the techniques described below for reducing interrupts and batching device-driven work by the driver may also result in higher throughput. Of course, if the I/O has throughput or deadline requirements, background scheduling is not sufficient.

It might appear that the interrupt-driven work of device drivers could be moved directly to their client threads, to be scheduled at a priority consistent with the continuation processing after the I/O operation. That does not work very well, for several reasons. One is the technical difficulty of unblocking the client task from inside an interrupt handler without unsafe race conditions. Another reason is that there may be several threads of different priorities sharing access to the same device, so in that case there may be I/O priority inversion, as higher-priority threads with pending I/O requests wait for their requests to be served until the interrupt-driven work of a prior request is executed by a lower priority client. Reducing CPU priority inversion due to interrupt-driven work with-

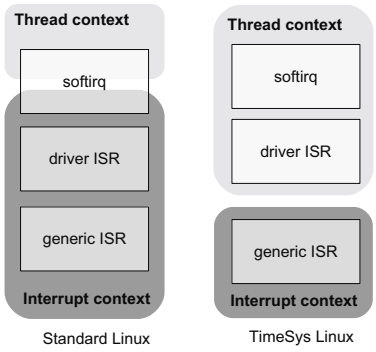


Figure 1: Linux *softirq* handling schemes.

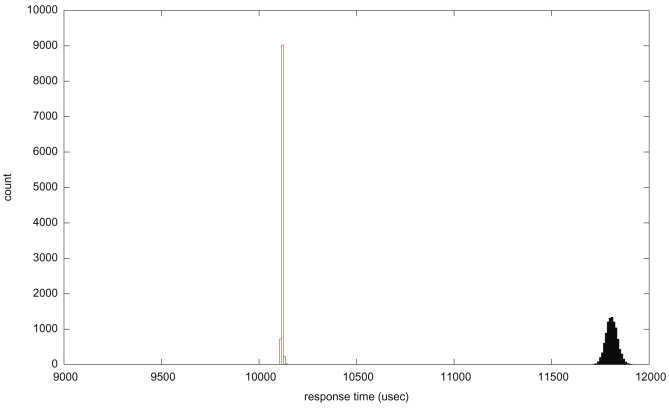


Figure 2: Response time distributions of a task with 100 msec. period and 10 msec. execution time, with and without device driver interference.

out creating such I/O priority inversion can be accomplished by moving work from the interrupt handler to a regularly scheduled server thread of appropriate priority, below interrupt level, but high enough to provide the desired level of I/O service.

Real-time variants of the Linux kernel, including those of Timesys and Montavista, have been modified to execute device-driver interrupt handlers and softirq handlers only from server threads. This is illustrated in Figure 1. For example, in the Timesys kernel, one thread is dedicated to processing interrupts for message-receipt events on the network device, and another to processing message-send events. The priority of each server thread can be set to a level that fits the priority the service it provides. If the priority is lower than that of any real-time thread, preemption interference effects due to softirq's can be ignored in schedulability analysis.

The difference in driver interference effects between running the device-driver server threads at low versus high priority are illustrated in Figure 2. These super-imposed histograms show the response time distribution for a periodic thread, with period of 100 milliseconds and an execution time of 10 milliseconds, running at high real-time priority. I/O load was provided by sending ping packets to the system, at random intervals between 10 and 2000 microseconds, and compiling a Linux kernel at normal user priority. The response times that form a spike between 10 and 10.5 milliseconds are from experiments in which the device driver server threads ran at lower priority than the periodic task. The response times that form a hump between 11.5 and 12 milliseconds are from experiments in which the device driver server threads ran at higher priority.

The idea of using a single interrupt server thread serving multiple interrupts by using a prioritized work queue appears in a 1990 patent by Youngblood [34]. The idea of assigning a thread of appropriate priority to each interrupt appears in a patent by Kleiman in 1996 [14]. The idea of allowing the priority of interrupt server threads to float, at the maximum priority of the set of threads that have devices open that use the corresponding interrupt, appears on the LynxOS 1995 patent, by Bunnell [5]. Regardless of which of these solutions is used, one can model the interrupt-triggered execution of a driver by two tasks, one that has short jobs at interrupt priority, and another that has longer jobs at a lower priority.

Zhang and West [35] proposed a variation of the LynxOS approach. The essential difference is that instead of scheduling the softirq server at the maximum priority of the threads that have an open file description served by the device, they use the maximum priority of the client threads that currently seem to be awaiting service by the device. The processing time for a particular softirq can then be charged against the client thread that it serves. This approach makes sense for device driver execution that can logically be charged to a client thread, but not all I/O has that property. Moreover, it suffers a potential priority inversion problem that is similar to the case if the bottom half were executed directly by the client thread. Consider a system with three real time processes, at three different priorities. Suppose the low priority process initiates a request for a stream of data over the network device, and that between packets received by the low priority process, the middle-priority process (which does not use the network device) wakes up and begins executing. The network-device server thread would have too low priority to preempt the middle-priority process, and so a backlog of received packets would build up in the DMA buffers. Next, suppose the high priority process wakes up and during its execution, attempts to read from the network device. This will finally raise the device server's priority to that of the high priority process. However, since the network device driver handles packets in FIFO order, the bottom half is forced to work through the backlog of the low-priority process's input before it gets to the packet destined for the high priority process. This additional delay could be enough to cause the high priority process to miss its deadline. That would not happen if the low-priority packets were cleared out earlier, as if the device bottom half had been able to preempt the middle-priority task. The LynxOS technique of doing priority inheritance through *open()* operations does not have this problem.

These ideas address the problems of finding the right priority for the interrupt-driven work of the device driver, but they do not address the problem of how to bound the interference due device-driven I/O.

Facchinetti *et al.* [7] recently proposed a way of doing this, but without addressing the priority problem. The system executes all Level 2 interrupt service as one logical thread, at the highest system priority. The thread implements an *ad hoc* aperiodic server scheduling policy, based on budgeting service time at the granularity of individual handler executions. This imposes a bound on the interference the server can cause lower priority threads in any scheduling window. Otherwise, the Level 2 handlers are executed like normal Linux softirq handlers, without preemption by threads

or other Level 2 handlers, in interrupt context, ahead of the application thread scheduler. Since all devices share the same budget and share the same priority, the system does not distinguish different priority levels of I/O, and handles all I/O in FIFO order. This can have undesirable consequences. For example, suppose the network interface is flooded with incoming packets, causing the Level 2 interrupt server thread to exhaust its budget. If a high priority task then requests disk I/O, completion of the disk I/O will be delayed until the Level 2 interrupt sever budget is replenished, and the high priority task may not meet its deadline.

Lewandowski *et al.* [18], proposed a similar approach, but based on using multiple server threads at different priorities, scheduled by an aperiodic server policy at the thread level. They suggest the Sporadic Server policy, since that is already supported for user-level threads by the POSIX/Unix real-time API. This has the virtue of both limiting the interrupt-driven interference that softirqs can cause, regardless of the rate at which the device attempts to push input at the system, while allowing different devices to be served at different priorities, and with different CPU bandwidths. It does not require any modification to existing device drivers.

Lewandowski *et al.* also suggest a way of empirically estimating an upper bound on device driver interference, which can be used directly in schedulability analysis, or used to calibrate the scheduling parameters of a sporadic server.

7. MANAGING INTERRUPTS

Just deferring Level 2 interrupt handling may not be enough. With device-driven input devices, such as a high-speed network interface card, there are situations where the Level 1 hardware interrupt handling alone could cause real-time tasks to miss deadlines. A defense against such an interrupt storm is to disable interrupts. This technique is the basis of the new Linux API for network devices (NAPI) [20], which is implemented by at least one driver. Once an interrupt is received from the device the interrupt is left disabled, at the device level. The Level 2 processing loop, which moves data from the DMA buffers to other buffers and passes them up the protocol stack, runs with the interrupt disabled. It is only re-enabled when the server thread executing the Level 2 loop polls, discovers it has no more work, and so suspends itself.

This mechanism was originally introduced to reduce so-called *receive live-lock*, where a system is so busy handling packet interrupts that it has no time left to process the data, but it has proven to have other benefits. By preventing unnecessary interrupts, it avoids the context-switch overhead for some packets, reducing the net execution time per packet, and so can sustain higher data rates. In addition, when the Level 2 packet processing is done by a thread at low priority level, if packets arrive faster than the server thread can handle them the DMA buffers will fill up and the device will drop packets until the thread has caught up.

The net effect is that interrupts are throttled. A job with higher priority than the Level 2 receive-processing thread can never be preempted by more than one interrupt from the network device. Since the Level 1 interrupt handler is very short, the worst-case interference for high priority tasks is not only bounded, but very small.

The hybrid polling/interrupt technique used in NAPI can be generalized to manage the rate of interrupts from other types of devices. However, barring device malfunctions that cause a stuck interrupt, it should only be needed for devices that are similar to network devices in the sense of spontaneously initiating interrupts. Many other classes of devices will only interrupt to indicate completion of an operation requested earlier by a client, so the rate of interrupts can be managed by a client, by managing the rate of requests.

As time goes on, hardware devices that are capable of generating interrupts at a high rate may provide throttling capabilities directly. That already appears to be the case with the Intel 8254x and 8257x gigabit Ethernet controllers [13], which provide several choices of operating modes in which hardware interrupts may be throttled back to fit a sporadic task model.

Although interrupt throttling ameliorates the problem of interrupt storms, and budgeting time for processing of Level 2 interrupt handling bounds direct interference from the device driver for top priority threads, these methods only indirectly address (via dropped messages) the broader problem of managing the amount of work being accepted into the system. That is, even at acceptable levels of hardware interrupt and softirq activity, some form of early admission control may be needed to throttle the workload of application tasks and to prevent possible resource exhaustion (*e.g.*, buffer space) that might lead to subsequent scheduling interference. Of course, such admission control requires CPU time also, and must be taken into account in the analysis of interference.

8. I/O SCHEDULING EFFECTS

The discussion so far leaves out I/O scheduling, that is, determination of the order and times at which I/O requests are served by each device. Some devices in real-time embedded systems – such as primitive sensors and actuators – do not require I/O scheduling and can perform operations immediately in response to a command from a thread, with no scheduling and very predictable response time. However, there are other I/O devices – such as mass storage devices, network devices, and radars – need scheduling. These are typically devices that need to be shared between threads, have highly variable response times, and may logically perform operations in more than one order. The quality of such I/O scheduling can affect the ability of an application to meet both response time and throughput requirements.

Device drivers may be involved in doing I/O scheduling. They are also affected by I/O scheduling, since the timing and order of I/O operations partially determines the workload of the device driver. Consider a blocking read operating to a disk. The time at which the disk actually performs the operation depends on what other operations are queued for the disk, the order in which they are served, and how long it takes to process each of them.

Schedulablility analysis for I/O is difficult because responsibility for I/O scheduling is distributed among different implementation layers. Device driver software may make some I/O scheduling decisions, but the service order and response times seen at the level of an application task may also be influenced by the device itself and by higher-level system software. For example, while a disk device driver may determine the order in which it passes on the I/O requests it receives, the order in which it receives those requests may be determined by higher-level operating system components, and the order in which the requests it sends out are actually served may be affected by the disk drive itself, by an intermediary controller, a multi-device driver and possibly a logical volume manager in the case of RAID systems, and by filesystem layout. The actual completion time of an I/O request is further complicated by the additional implicit requests for file system metadata associated with the requested I/O. Critical but infrequent error recovery mechanisms

at various levels can also be triggered to perform journal recovery, parity reconstruction, and bad sector forwarding. Full response-time analysis for disk I/O requests will require consideration of the net effect of all these levels. Similarly complex multi-layer interactions are potentially involved in determining service order for other important classes of devices, such as data communication interfaces and radars.

While it may be possible to concentrate the I/O scheduling implementation at one level, there may be a penalty in reduced control over scheduling, or increased overhead and reduced throughput. These are potentially complex trade-offs that need to be resolved for each type of device, and for each application.

Another aspect of I/O scheduling that makes it difficult to analyze is non-preemptivity. Operations on most I/O devices cannot be interrupted, once started. For example, a network interface device would not typically provide the option of interrupting transmission of one message in order start another, nor would that make sense, given the high overhead that such preemption would incur. Similarly, given the long time it takes to get a disk head into position to access a given sector, it would not makes sense to preempt a disk drive in the midst of an I/O operation. As mentioned above, one consequence of non-preemptivity is that the scheduling effects of execution time variation in cannot be bounded by just considering the shortest and longest cases [10, 9].

The difficulty of even single-device analysis is exacerbated by the fact that I/O times can be context-sensitive. For example, in the case of a disk drive the time to access a given block depends on the position of the disk head relative to the block location and the content of the driver's local cache. That, in turn, depends on what operation was scheduled before.

Another important issue that impacts schedulability analysis involving I/O response times is timing constraints that span multiple jobs, involving several different processors and precedence constraints. For instance, a network video server might read video frames from a frame grabber or a local disk, perform some computation on the video data (say trans-coding or frame skipping), and transport the requested data across the network. In this example, there is a precedence ordering among jobs on the different devices, and each job depends upon the successful and timely completion of a previous task in the sequence. In addition, to achieve a reasonable perceptual quality, the entire sequence of tasks needs to be repeated regularly.

I/O scheduling also can involve multiple conflicting quality–of-service criteria. For example, priority-based scheduling of disk I/O can reduce response time for a few high-priority requests, but at the cost of increased total processing time (for head movement and rotational latency), which means reduced throughput. Algorithms that provide good average throughput provide very hard-to-predict response times. So, if a system has tasks with both response time and throughput requirements, perhaps in the same task, it is not clear what to do.

For the reasons given above and others, when I/O scheduling is considered together with scheduling of the CPU and possibly other devices, the analysis problem becomes extremely difficult. The theory, so far, has very little useful to say about these problems. Some research has been done on limited aspects of the end-to-end I/O scheduling problem (*e.g.*, [2, 16, 25, 8, 36]), from the perspective of either resource allocation to support QoS or co-ordinated scheduling of specific set of resources. However, a comprehensive theory of multi-resource allocation and schedulability analysis does not yet seem to exist.

9. CONCLUSION

The state of practice in Linux seems close to providing adequate support for constructing a variety of real-time systems to meet timing constraints by design, and verifying them, based on preemptive scheduling theory. However, there remains some work to be done at the level of device drivers.

One of the advantages of a mature, widely used, general-purpose operating system like Linux is that it already has a large collection of device drivers. There are good reasons for trying to reuse some of these drivers in a real-time application. Techniques exist that permit modeling the role of some existing device drivers in system schedulability, including ways of bounding the interference real-time application tasks may experience due to the Level 2 interrupt processing, with only minor changes to the way softirq handing is scheduled by the kernel. With further change, in the design of device drivers, the interference due to Level 1 interrupt processing may also be bounded.

These solutions do require tailoring of how Level 2 interrupt handling is scheduled and how Level 1 interrupts are throttled, to fit the needs of an application. That currently can only be done by modification of the kernel and/or device driver code. It is possible in an open-source system, but is still an obstacle to widespread use. For the adoption of such techniques with proprietary operating systems, it is a more significant roadblock.

This situation could be improved by expanding the device driver and user programming interfaces, to provide more visibility and control over device driver scheduling and interrupt handling at the application level.

It may also be advantageous to provide enforcement mechanisms to improve the real-time quality of device drivers, which are developed independently by a large number of individuals, who are not all fully aware of what effects their device driver might have on the schedulability of other system components, or the relative importance of driver-internal timing constraints as compared to other requirements in a particular real-time application.

Problems that are more serious exist with regard to achieving and verifying end-to-end timing requirements that span I/O operations, which need to be solved better in theory before one can talk seriously about how to support the theory better in an operating system. A reasonable way to make progress is to look at restricted cases, such as computations involving just the CPU and one I/O resource, either modeling existing Linux I/O scheduling policies for the device or modifying the device scheduling policy to make real-time performance more analyzable.

REFERENCES

[1] Advanced Informatics. SCAN-schedulability analysis tool. `http://www.spacetools.com/tools4/space/272.htm`.

[2] D. P. Anderson. Meta-scheduling for distributed continuous media. Technical Report CSD-90-599, ECE Department, University of California at Berkeley, Dec. 1990.

[3] N. C. Audsley, A. Burns, M. Richardson, and A. J. Wellings. Hard real-time scheduling: the deadline monotonic

approach. In *Proc. 8th IEEE Workshop on Real-Time Operating Systems and Software*, pages 127–132, Atlanta, GA, USA, 1991.

[4] M. Barabanov. A Linux-based real-time operating system. Master's thesis, New Mexico Instituted of Techology, Albuquerque, NM, June 1997.

[5] M. Bunnell. Operating system architecture using multiple priority light weight kernel task based interrupt handling, u. s. patent 5,469,572. `http://www.upsto.gov`, 1995.

[6] P. Cloutier, P. Mantegazza, S. Papacharalambous, I. Soanes, S. Hughes, and K. Yaghmour. DIAPM-RTAI position paper, nov 2000. In *RTSS 2000 Real-Time Operating Systems Workshop*. IEEE Computer Society, 2000.

[7] T. Facchinetti, G. Buttazzo, M. Marinoni, and G. Guidi. Non-preemptive interrupt scheduling for safe reuse of legacy drivers in real-time systems. In *Proc. 17th IEEE Euromicro Conference on Real-Time Systems*, Palma de Mallorca, July 2005.

[8] K. Gopalan and T. Chiueh. Multi-resource allocation and scheduling for periodic soft real-time applications. In *Proc. Multimedia Computing and Networking*, San Jose, CA, USA, Jan. 2002.

[9] R. Ha. *Validating timing constraints in multiprocessor and distributed systems*. PhD thesis, University of Illinois, Dept. of Computer Science, Urbana-Champaign, IL, 1995.

[10] R. Ha and J. W. S. Liu. Validating timing constraints in multiprocessor and distributed real-time systems. In *Proc. 14th IEEE International Conf. Distributed Computing Systems*, pages 162–171, Poznan, Poland, June 1994. IEEE Computer Society.

[11] A. C. Heursch, D. Grambow, A. Hosrtkotte, and H. Rzehak. Steps towards a fully preemptable Linux kernel. In *Proc. 27th IFAC/IFIP/IEEE Workshop on Real-Time Programming*, Lagow, Poland, May 2003.

[12] A. C. Heursch, D. Grambow, D. Roedel, and H. Rzehak. Time-critical tasks in Linux 2.6: Concepts to increase the preemptability of the Linux kernel. In *Linux Automation Konferenz*, Germany, Mar. 2004. University of Hanover.

[13] Intel Corporation. Interrupt moderation using intel gigabit ethernet controllers (AP-450). Application Note, Intel Corporation, Apr. 2007.

[14] S. Kleiman. Apparatus and method for interrupt handling in a multi-threaded operating system kernel. U. S. Patent 5,515,538, 1996.

[15] M. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A Practioner's Handbook for Real Time Analysis: Guide to Rate Monotonic Analysis for Real Time Systems*. Kluwer, Boston-Dordrecht-London, 1993.

[16] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, and J. Hansen. A scalable solution to the multi-resource QoS problem. In *Proc. IEEE Real-Time Systems Symposium*, Phoenix, AZ, USA, Dec. 1999.

[17] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in a hard real-time environment. In *Proc. 8th IEEE Real-Time Systems Symposium*, pages 261–270, 1987.

[18] M. Lewandowski, M. Stanovich, T. Baker, K. Gopalan, and A. Wang. Modeling device driver effects in real-time schedulability analysis: Study of a network driver. In *Proc. 13th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 57–68, Bellevue, WA, Apr. 2007.

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

[20] J. Mogul and K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.

[21] I. Molnar. Preemptive kernel patches. `http://people.redhat.com/mingo/`.

[22] Montavista, Inc. Montavista Linux. `http://www.mvista.com`.

[23] S. Oikawa and R. Rajkumar. Linux/RK: A portable resource kernel in linux. In *Real-Time Systems Symposium Work-in-Progress*, Dec. 1998.

[24] G. Palmer and R. West. Hijack: Taking control of cots systems for real-time user-level services. In *Proc. 13th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 133–146, Bellevue, Washington, Apr. 2007. IEEE Computer Society Press.

[25] S. Saewong and R. Rajkumar. Cooperative scheduling of multiple resources. In *Proc. IEEE Real-Time Systems Symposium*, Phoenix, AZ, USA, Dec. 1999.

[26] L. Sha, T. Abdelzaher, K. E. Årzén, A. Cervin, T. P. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2–3):101–155, Nov. 2004.

[27] L. Sha, J. P. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritizing preemptive scheduling. In *Proc. 7th IEEE Real-Time Sytems Symposium*, 1986.

[28] B. Sprunt, L. Sha, and L. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1(1):27–60, 1989.

[29] J. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in real-time environments. *IEEE Trans. Computers*, 44(1):73–91, Jan. 1995.

[30] G. H. Thaker. Real-time OS periodic tests. `http://www.atl.external.lmco.com/projects/QoS/RTOS\_html/periodic.html`.

[31] The Open Group. The single Unix specification, version 3. `http://www.unix.org/version3/`.

[32] TimeSys, Inc. Embedded Linux development products. `http://www.timesys.com`.

[33] V. Yodaiken. The RTLinux manifesto. In *Proc. 5th Linux Expo*, Raleigh, NC, 1999.

[34] L. Youngblood. Interrupt driven prioritized queue. U. S. Patent 4,980,820, 1990.

[35] Y. Zhang and R. West. Process-aware interrupt scheduling and accounting. In *Proc. 27th Real Time Systems Symposium*, Rio de Janeiro, Brazil, Dec. 2006.

[36] Y. Zhou and H. Sethu. On achieving fairness in the joint allocation of processing and bandwidth resources: Principles and algorithms. Technical Report DU-CS-03-02, Drexel University, 2003.

Experimental results of aperiodic fixed-priority preemptive policies in RT-Linux

L. Búrdalo, A. Espinosa, A. Terrasa and A. García-Fornes.
{lburdalo,aespinos,aterrasa,agarcia}@dsic.upv.es

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Valencia, Spain

ABSTRACT

The aim of this work is to complement the previous work on scheduling soft tasks in hard real-time systems based on fixed-priority preemptive scheduling. This work provides an experimental view of the most representative policies in the literature, measuring both their performance and overhead. The performance of aperiodic scheduling policies has been traditionally evaluated by means of simulations that compute the average response times of soft tasks. In our study, the response times of soft tasks is measured for each scheduling policy in several different task sets running on top of a real-time operating system (RT-Linux). The overhead of each execution is also measured in order to determine how the overhead can affect the performance of such policies.

1. INTRODUCTION AND PURPOSE

Aperiodic fixed-priority preemptive scheduling policies have been studied and compared by many authors, and their results have already shown considerable improvements in the response times of soft tasks; however, few of them have been incorporated to real applications. For instance, POSIX [3] only supports one of the existing *server-based* scheduling policies (the *Sporadic Server*), and does not support any of the *slack-stealing-based* or *dual-priorities-based* scheduling policies.

Previous works on scheduling policies are mainly based on simulation because real systems are more difficult to study. The use of simulations has many advantages. First of all, implementation of scheduling policies is more affordable in simulation tools than inside the kernel of a RTOS. Second, simulation makes it possible to easily generate a high number of task sets where system parameters like task periods or execution times can follow any probabilistic distribution. Finally, results can be extracted directly without having a tracing facility integrated in the RTOS kernel.

Nevertheless, simulation tools also have some drawbacks. It is not easy to model or simulate the time that an operating system needs to decide which task to execute next, and this time is different in each specific scheduling policy. For this reason, most simulation-based studies do not take this extra overhead into account or simplify it too much (in several studies, context switches are considered cost-free). On the other hand, it is very common to generate the soft workload by using *unit load*. This implies that a system performs exactly the same when each one of 10 tasks must be executed during 1 time unit each (10 units) than when 1 task must be executed during 10 time units.

The overhead has been considered differently in previous studies. Some studies include the overhead in the feasibility analysis of the scheduling policy, others estimate the overhead by means of the number of context switches introduced by the policy, and still others do not consider the overhead in any way. Unfortunately, in some scheduling policies, overhead is not negligible at all, specially when the system has to face high loads or a scheduling policy tends to be highly preemptive. As a result, it is difficult to apply the conclusions from previous studies to real applications.

The aim of this work is to complement the previous work on scheduling soft tasks in hard real-time systems based on fixed-priority preemptive scheduling. This work provides an experimental view of the most representative policies in the literature, measuring both their performance and overhead. The performance of aperiodic scheduling policies has been traditionally evaluated by means of the average response times of soft tasks. The better the system performs, the lower the response times of soft tasks are, always taking into account that hard deadlines cannot be missed. In our study, the response times of soft tasks are measured for each scheduling policy in several different task sets running on top of a real-time operating system (RT-Linux). The overhead of each execution is also measured in order to determine how the overhead can affect the performance of such policies.

This paper is structured as follows: Section 2 presents some previous studies about soft task scheduling policies and their main conclusions. Section 3 briefly describes our framework for measuring the performance and the overhead of real-time applications over the RT-Linux RTOS. Section 4 presents the design of the experiments carried out for this work and presents a discussion of the results of these experiments. Finally, Section 5 presents the main conclusions of the paper, as well as some future lines of work.

2. PREVIOUS STUDIES

Although the number of scheduling policies and the policies themselves differ in each particular study and the results from the different studies are difficult to compare, their main conclusions can be summarized.

Server-based policies improve the results obtained by scheduling soft tasks in background when the system load is not too high; however, as the system load grows, these policies tend to perform like background scheduling. The *Deferrable Server* (DS) and the *Sporadic Server* (SS) are the most popular of this famiy of policies. They mainly differ in the way the capacity of the server is replenished.

Previous works on these two policies do not come to the same conclusions. Studies in [14, 17, 4, 19] conclude that SS is better than DS because it allows larger capacities and gets higher utilization values, while [15] shows larger response times for SS than for DS. Finally, [6] concludes that both policies have similar response times and can get similar utilization values.

The number of context switches is studied in [15] as an approximation of the overhead of the different policies. Server-based policies produce a much higher number of context switches than the other scheduling policies considered in the study.

Slack-stealing-based policies are taken into account in different studies, although most of them are not suitable for a real system due to the high overhead that these policies introduce in the system. However, since [11] proved that the *Dynamic Slack Stealing* (DSS) policy was optimal (it minimizes the response times of soft tasks without missing any hard deadlines), it has been used as a reference to compare different policies. The only scheduling policy in this family which can be included in a real time system is *Dynamic Approximate Slack Stealing* (DASS), which is studied in [10] to show a performance very near to DSS until the total load in the system (hard+soft+overhead) gets to 90%, at which point the system performance starts to degrade. Results in [12] show that DASS is very near to DSS in all cases, although overhead is considered to be negligible.

The *Dual Priorities* scheduling policy (DP) is compared to other scheduling policies in [8, 9, 12, 13]. The results are very similar in all the studies: DP gets lower response times than background scheduling and server-based policies. In fact, DP performs in a similar way to DASS when the system load is less than 90%, although response times are better for slack-stealing-based policies. [13] also shows that DP performs better than background scheduling if and only if soft load is served in FIFO order.

In general, all of the authors conclude that specific scheduling policies obtain better results than background scheduling (average response times of soft tasks are smaller), although almost none of them take the extra overhead into account and all of them point out that, when implemented over a real system, DASS involves a much higher overhead than server-based or DP policies.

3. FRAMEWORK AND RTOS

This study has been carried out using our framework [7] for the automatic generation of workloads and automatic processing of the results. This framework has been extended in order to incorporate the DASS scheduling policy and the effects of gain time in some of the scheduling policies. Gain time is the amount of time which, even though it is included in the *worst case execution time* of a task, it is not used by that task at run time in any of the activations of the task.

The framework was built around the RTOS RT-Linux GPL [2, 1, 5], version *3-2-pre1*. This framework uses *RTL-Posix-Trace* [20], which is a package that adds most of the tracing support defined in the POSIX Trace standard to RT-Linux. *RTL-Posix-Trace* provides the RTOS kernel with a standard mechanism and a set of trace events such as scheduler interruptions, context switches, creation, destruction, activation and suspension of tasks. which allow the analysis of the execution of an application by processing the generated trace-events log file. *RTL-Posix-Trace* is described in more detail in [20] and [18].

The framework receives the specification of the experiment to be carried out as input, including some definition parameters such as the number of hard and soft tasks in the system, the percentage of hard and soft load in the system, and the scheduling policies. These parameters of the experiment can either be fixed/random values or a range of values, so that it is possible to include in the experiment all the different combinations of the parameter values.

In this work, for each possible combination of the parameters, a *synthetic application* is generated. In this application, hard tasks consume between 95% and 100% of their *worst case execution time* and the set of hard tasks is schedulable.

Each *synthetic application* is compiled and executed over RT-Linux using each of the scheduling policies. When launched, these *synthetic applications* start with an initial critical instant and then continue running until the end of the task set hyperperiod. As a consequence, all the scheduling policies can be studied on equal terms.

After their execution, the corresponding trace-events log files are processed to calculate a set of *metrics* which are graphically represented as a function of the different parameters of the experiment (for instance, response times of tasks or average time spent by the scheduler in each interruption).

4. EXPERIMENTS

Two different experiments have been carried out. The aim of the first one is to compare the specified scheduling policies with each other and to study how much and in what way the different parameters of a real time

system can affect its performance and overhead depending on the scheduling policy being used. The aim of the second experiment is to study how *gain time* support can improve the system behaviour. All the experiments were run on a Pentium III 700Mhz with 512Mb of RAM.

Figure 1 shows how each scheduling policy is represented in graphs.

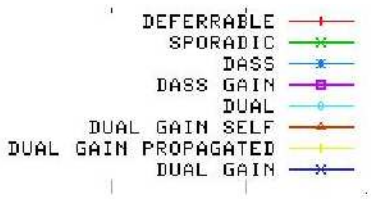


Figure 1: **Representation of each scheduling policy**

4.1 Scheduling policies

In order to make this study as general as possible, the most representative scheduling policies specific to systems with hard and soft load have been taken into account: *Deferrable Server*, *Sporadic Server*, *Dynamic Approximate Slack Stealing*, *Dual Priorities*, and *background scheduling*.

Deferrable Server [17] and *Sporadic Server* [4] have been considered since they are the most representative of all server-based scheduling policies. The *Dynamic Approximate Slack Stealing* policy [10] is the only slack-stealing-based policy that can be implemented and incorporated to a RTOS, since temporal and spatial overheads of exact and static techniques, respectively, make these policies impracticable (more information can be found in [16] and [11]). The *Dual Priorities*, which is described in [9], has also been included in this study.

When using *Dynamic Approximate Slack Stealing* or *Dual Priorities*, hard tasks can take advantage of the situation when a higher priority spends less than its *worst case execution time*. In this paper, this is called *propagated gain time*. When using *Dual Priorities*, any hard task that has started executing at its low priority level can delay its *promotion time*. In this paper, this is called *self gain time*.

To consider the extra overhead due to gain time support, both *Dynamic Approximate Slack Stealing* and *Dual Priorities* are considered with and without giving support to gain time.

4.2 Design: Experiment 1

The parameters of this experiment are the following:

- Number of hard tasks: 4, 8, 12 and 16
- Number of soft tasks: 1
- Percentage of hard load: 20%, 40% and 60%
- Percentage of soft load: 10%, 20%, 30%, 40%, 50% and 60%

When using server-based policies, the soft task is always executed at the highest priority level until the server runs out of capacity.

The number of possible combinations of these values is 72. For each possible combination 5 different task sets were generated, making a total of 360 different *synthetic applications* that were run once with each scheduling policy.

In order to observe differences in the overhead, *Dynamic Approximate Slack stealing* and *Dual Priorities* were considered with and without gain support.

4.3 Results: Experiment 1

This section shows the results of the entire experiment. The performance of each scheduling policy is measured by the average response times of the soft tasks; and the RTOS overhead is measured by the number of context switches, the average amount of time spent by the RTOS during an interruption of the scheduler, and the total amount of time spent by the scheduler during the execution of each application (from the initial critical instant to the first hyperperiod).

For each *synthetic application*, the values of the metrics obtained using each scheduling policy are divided by the value obtained using background scheduling. The resulting ratio values are used to calculate an average value which will represent that particular combination of values of the experiment parameters.

By doing so, results of different *synthetic applications* or even different experiments are comparable, and it is possible to group them in a single graph as a function of any of the experiment parameters. Thus, differences between each policy and background scheduling and differences among the policies themselves are clearer.

4.3.1 Soft task response times

Figures 2, 3 and 4 show the average response times of the soft tasks as a function of the percentage of hard load, number of hard tasks, and percentage of soft load, respectively.

As can be observed, all the scheduling policies have lower response times than background scheduling in all cases, since the ratio values are always lower than 1.

Differences between all the scheduling policies and background scheduling become higher as the percentage of hard load becomes higher (Figure 2), which means that, as this percentage grows, all the scheduling policies behave better than background scheduling. It can also be observed that DS and SS separate when hard load values become high. The reason for that is that the capacity of DS is much smaller than the capacity of

SS, due to the fact that the feasibility test of DS is more restrictive than the one of SS.

On the other hand, differences between each scheduling policy and background scheduling tend to be smaller as the number of hard tasks (Figure 3) and the percentage of soft load (Figure 4) become higher.

It can also be observed that server-based policies always obtain higher response times than non-server-based policies and, although SS provides lower response times for the soft task, both server-based policies are very similar.

The results for DASS and DP are also very similar; however, when the number of hard tasks or the percentage of hard or soft load gets high, response times are better for DP.

Since hard tasks were generated so that they requested almost all their *worst case execution time* in each activation, very little gain time appears at run time. This explains the lack of significant improvement when supporting gain time.

These results confirm what [6] concludes since both DS and SS obtain similar response times; however, response times obtained with SS are lower than for DS in all cases.

DP also performs better than expected from the results in [8, 9, 12]. Although DASS and DP get similar response times (even more similar than in previous studies), as the system load grows, DP starts outperforming DASS.

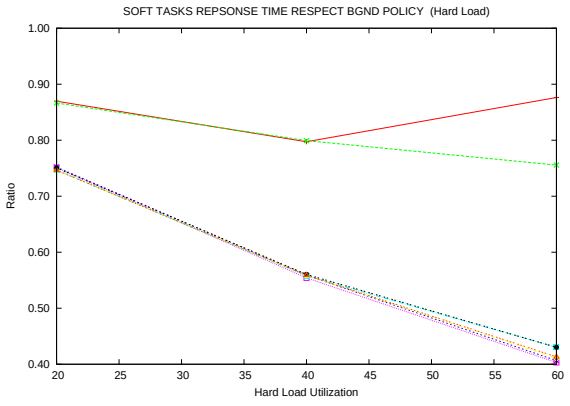


Figure 2: **Average response time of soft tasks as a function of the hard load in the system. Relative values.**

4.3.2 Number of context switches

Figures 5, 6 and 7 show the number of context switches as a function of the the percentage of hard load, number of hard tasks, and percentage of soft load in the system.

All the graphs show ratio values that are very close to 1, which means that there is little difference between

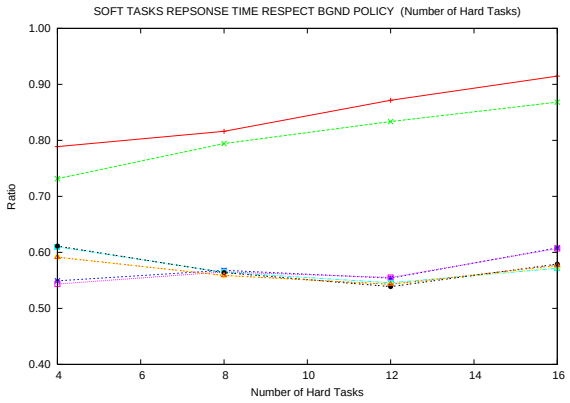


Figure 3: **Average response time of soft tasks as a function of the number of hard tasks in the system. Relative values.**

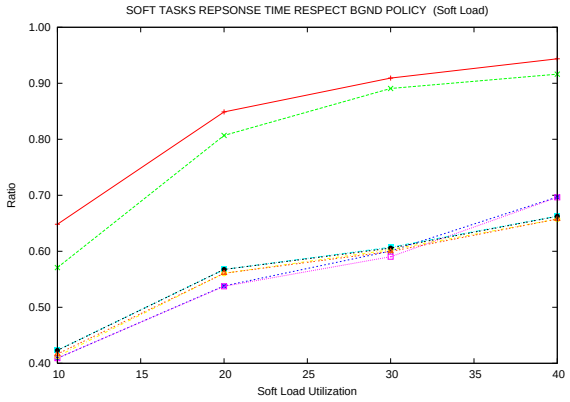


Figure 4: **Average response time of soft tasks as a function of the soft load in the system. Relative values.**

the number of context switches of each scheduling policy and background scheduling. However, ratio values tend to become higher as the percentage of hard or soft load becomes higher, while it tends to be lower as the number of hard tasks is greater (background scheduling provokes more context switches than the other policies as the number of hard tasks grows).

For DASS, supporting gain time always leads to more context switches, specially for high values of the parameters of the experiment. Hard tasks always consume almost all of their *worst case execution time*, but not all. As a consequence, small amounts of gain time appear whenever a hard task finishes executing. If this happens when there is no slack available in the system, system slack becomes greater than 0, which usually provokes a context switch. However, since gain time is so small, the system runs out of slack soon, which provokes another context switch.

Server-based policies are the most preemptive of all, and only DASS with gain time support provokes more context switches for high values of hard tasks, hard load or soft load.

The number of context switches was already used as a measure of the system overhead in [15], where server-based policies were also the ones with more context switches; however, differences among policies are less significant than those shown in [15] (the number of context switches for server-based policies got up to 5 times the value for background scheduling). This may be because, in [15], an extra task was considered in the system that preempted hard tasks in order to let soft ones execute.

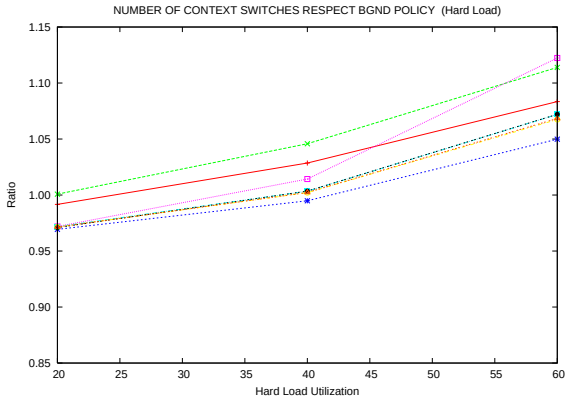


Figure 5: **Number of context switches as a function of the hard load in the system. Relative values.**

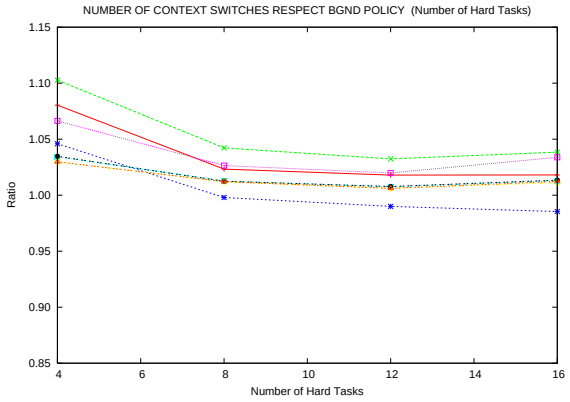


Figure 6: **Number of context switches as a function of the number of hard tasks in the system. Relative values.**

4.3.3 Scheduler overhead

The scheduler overhead has been measured in two different ways: the average amount of time spent by the RTOS scheduler in a single interruption and the total amount of time spent by the scheduler during the execution of each *synthetic application*.

Figures 8, 9 and 10 show the average amount of time spent by the RTOS in a single scheduling interruption action as a function of the percentage of hard load, the number of hard tasks and the percentage of soft load in the system.

The results show that the percentage of hard or soft load do not have any important influence over the average time of a scheduling interruption, while it is influenced directly by the number of hard tasks in the system, specially for slack-stealing-based scheduling policies. This is because the main overhead of these policies comes from the calculation of the available slack for each hard task in the system. This slack has to be calculated for all hard tasks whenever any task ends its execution; thus, more hard tasks imply more scheduling time.

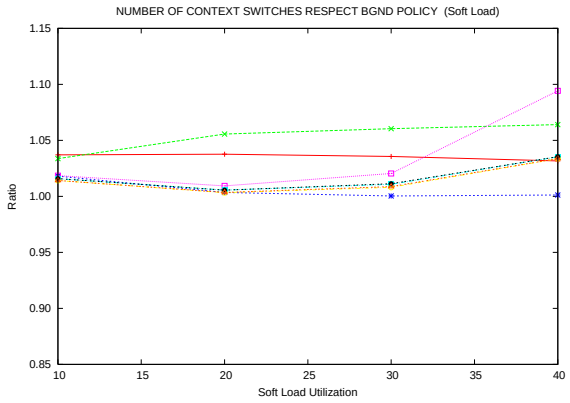


Figure 7: **Number of context switches as a function of the soft load in the system. Relative values.**

The average time spent by the scheduler for the two slack-stealing-based policies is much higher than for the rest of them, which explains differences between the response time results presented in this work and previous studies like [12].

Differences between the two server-based policies are not too important either, although DS has lower values as expected.

The results do not show any significant increase in overhead when supporting *gain time* for DASS, or for DP.

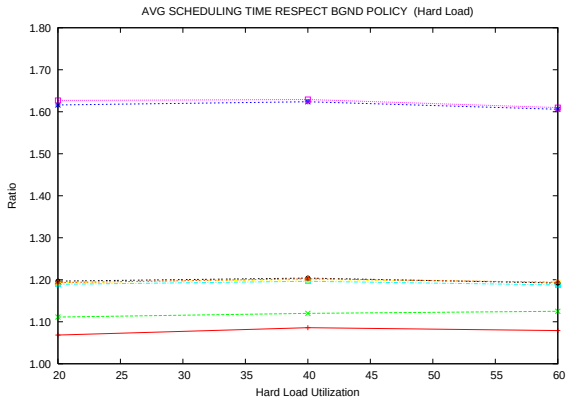


Figure 8: **Average time for a single scheduling interruption as a function of the hard load in the system. Relative values.**

Finally, Figures 11, 12 and 13 show ratio values for the total amount of time spent by the RTOS in scheduling

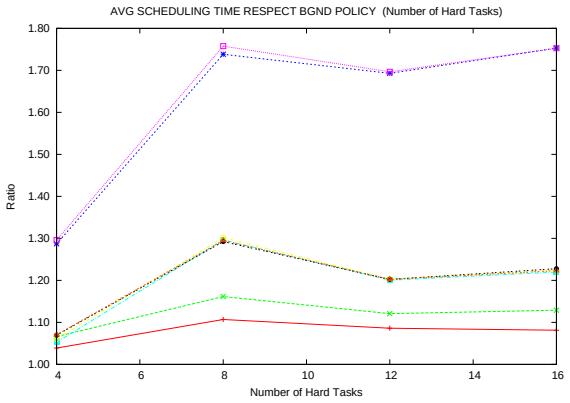


Figure 9: **Average time for a single scheduling interruption as a function of the number of hard tasks in the system. Relative values.**

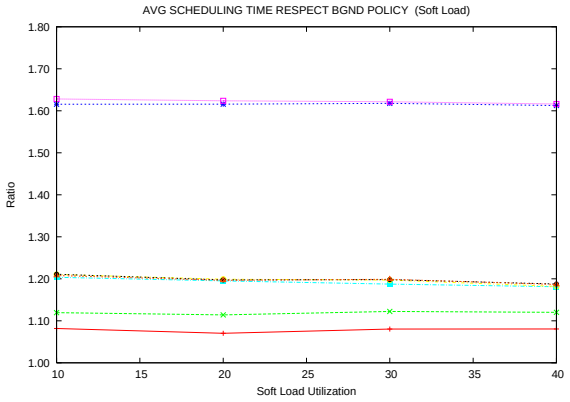


Figure 10: **Average time for a single scheduling interruption as a function of the soft load in the system. Relative values.**

during the execution of each *synthetic application*.

The resulting graphs are almost parallel horizontal lines, which means that the total amount of time spent by the scheduler during the execution has the same evolution for all the scheduling policies and for background scheduling. The only difference among them is the magnitude of the time spent by the scheduler with each policy.

Slack-stealing-based policies show much more overhead than the other policies. Differences between the two server-based scheduling policies are not important, although SS consumes more CPU. DP policies consume some more time than server-based, but not as much as DASS.

Differences between supporting or not supporting gain time are not as significant as differences among the different policies.

4.4 Design: Experiment 2

The parameters of this experiment are the following:

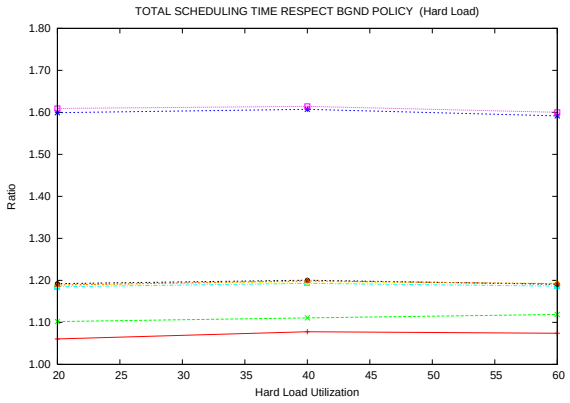


Figure 11: **Total amount of time spent by the scheduler as a function of the hard load in the system. Relative values.**

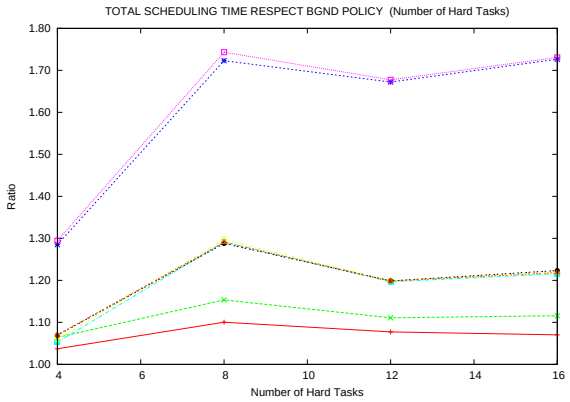


Figure 12: **Total amount of time spent by the scheduler as a function of the number of hard tasks in the system. Relative values.**

- Number of hard tasks: 4, 8, 12 and 16
- Number of soft tasks: 1
- Percentage of hard load: 60%, 70% and 80%
- Percentage of soft load: 10%, 20%, 30% and 40%

When using server-based policies, the soft task is always executed at the highest priority level until the server runs out of capacity.

The number of possible combinations of these values is 36. For each possible combination, 5 different task sets were generated, making a total of 180 different *synthetic applications* that were run once with each scheduling policy.

4.5 Results: Experiment 2

For this experiment, only response times of soft tasks were considered. Figures 14, 15, 16 and 17 show these response times as a function of the percentage of *gain time* in hard tasks, the percentage of hard load, the number of hard tasks and the percentage of soft load in the system.

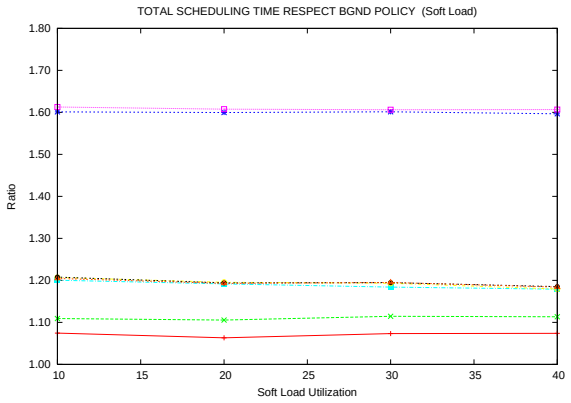


Figure 13: **Total amount of time spent by the scheduler as a function of the soft load in the system. Relative values.**

DASS obtains better results when supporting *gain time*. Differences between the two versions of DASS tend to increase when the percentage of hard load in the system is higher. For DP, differences between considering or not considering *gain time* are less important. The best results are obtained when considering *self gain time* rather than *propagated gain time*. Except for low percentages of hard or soft load or small task sets, all DP versions obtain lower response times than DASS without *gain time*.

These results differ from those in [9, 12], since here DP outperforms DASS when the number of hard tasks or the system load grow.

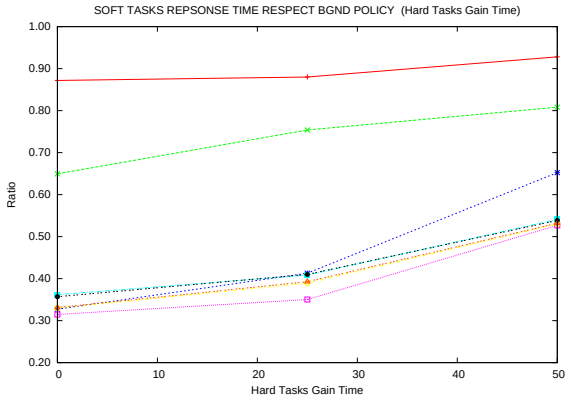


Figure 14: **Average response time of soft tasks as a function of hard tasks gain time in the system. Relative values.**

5. CONCLUSIONS AND FUTURE WORK

This study complements some previous work on the performance of scheduling policies for soft tasks in fixed-priority preemptive real-time systems. The main objective of this study was to determine whether or not the results of such previous work were maintained for real applications.

In order to do this, we selected the most representative policies and integrated them into our instrumented ver-

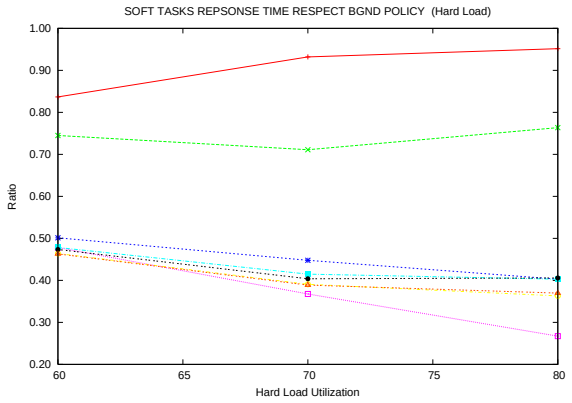


Figure 15: **Average response time of soft tasks as a function of the hard load in the system. Relative values.**

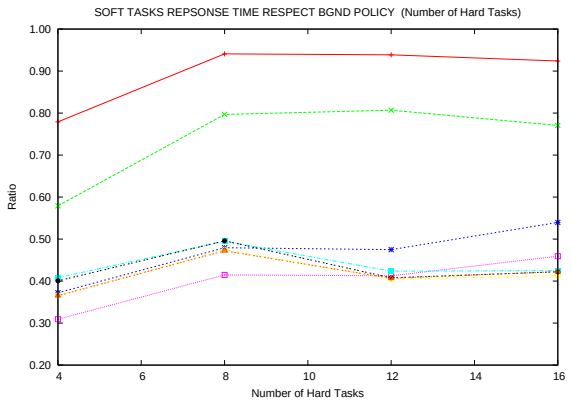


Figure 16: **Average response time of soft tasks as a function of the number of hard tasks in the system. Relative values.**

sion of RT-Linux. Then, we ran a large number of task sets for each policy and measured several aspects of each execution, including the response time of soft tasks, the execution time of each scheduler execution and the number of context switches. The overall goals of the experiment are to measure the different policies in equal terms and to determine how the overhead affects the performance in each case.

The results are presented in such a way that they show the average values of each aspect (response time, scheduler overhead and number of context switches) as a function of each experiment parameter (number of hard tasks, soft load utilization and hard load utilization). In this way, we can characterize the effects of each parameter independently.

The main conclusions of the experiment results are the following:

- All the scheduling policies improve the results of the background policy, even including the effects of the overhead in the execution.
- Due to the overhead, the performance of the DASS

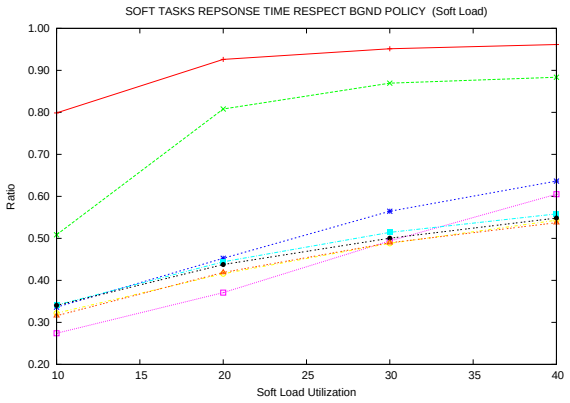


Figure 17: Average response time of soft tasks as a function of the soft load in the system. Relative values.

policy is worse than expected (in the simulation studies) and it is outperformed by the Dual Priorities policy. This confirms the conclusions of [9]. However, the DASS still performs better than any of the server-based policies.

- Server-based policies produce higher response times for soft tasks than the DASS and Dual Priorities policies, in spite of the fact that they also produce lower overhead. The two policies (DS and SS) do not differ very much in their performance when the system load is not too high. DS has a more restrictive feasibility test which allows less capacity than SS for high values of system hard load. This makes DS performance worse than SS. However, SS is more difficult to implement and produces more overhead.
- DASS provides lower response times when *gain time* is considered; however, this difference is less important for the DP policy.

Future lines of work will include an exhaustive testing of the scheduling policies in different scenarios in order to determine the best policy for each case. The testing framework will be furtherdeveloped to extend and combine such policies and to propose new ones; for example, to find ways to combine multiple servers in the same application (at the same priority, or at different priorities) in order to improve the overall performance.

Acknowledgements

This work is partially supported by the TIN2005-03395 project, which is co-funded by the Spanish government and FEDER funds.

6. REFERENCES

[1] http://rtportal.upv.es/.

[2] http://www.rtlinuxfree.com/.

[3] *1003.1, 2004 EDITION IEEE Standard for Information Technology Portable Operating System Interface (POSIX)*. IEEE, 2004.

[4] L. Sha B. Sprunt and J. Lehozky. Aperiodic task scheduling for hard real-time systems. *The Journal of Real-Time Systems*, (1):27–60, 1989.

[5] M. Barabanov. A linux-based real-time operating system. Master's thesis, Institute of Mining and Technology, New Mexico, Jun 1997.

[6] G. Bernat and A. Burns. New results on fixed priority aperiodic servers. In *IEEE Real-Time Systems Symposium*, pages 68–78, 1999.

[7] Luis Antonio Búrdalo Rapa, Agustín Espinosa Minguet, Ana Mª. García-Fornes, and Andrés Terrasa Barrena. Framework for the development and evaluation of new scheduling policies in rt-linux. In *OSPERT 2006*, pages 42–51, 2006.

[8] R. Davis. Dual priority scheduling: A means of providing flexibility in hard real-time systems. Technical Report YCS230, University of York, UK, May 1994.

[9] R. Davis and A. Wellings. Dual priority scheduling. In *RTSS '95: Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS '95)*, page 100, Washington, DC, USA, 1995. IEEE Computer Society.

[10] R. I. Davis. Approximate slack stealing algorithms for fixed priority preemptive systems. Technical Report YCS 217, Department of Computer Science, University of York, Nov 1993.

[11] R. I. Davis. Scheduling slack time in fixed priority preemptive systems. Technical Report YCS93217, Department of Computer Science, University of York, 1993.

[12] A. García-Fornés, A. Terrasa, and V. Botti. Planificación de tareas aperiódicas en sistemas de tiempo real estricto. *NOVATICA*, Septiembre:22–30, 1997.

[13] B. Gaujal, N. Navet, and J. Migge. Dual-priority versus background scheduling: A path-wise comparison. *Real-Time Systems*, (25):39–66, 2003.

[14] T. M. Ghazalie and T. P. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Syst.*, 9(1):31–67, 1995.

[15] J. Goossens and C. Macq. Performance analysis of various scheduling algorithms for real-time systems composed of aperiodic and periodic tasks. In *CISSAS'99*, 1999.

[16] J. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks fixed-priority preemptive systems. In IEEE Society Press, editor, *Proc. IEEE Real-Time Systems Symposium*, pages 110–123, December 1992.

[17] J. Lehoczky, L. Sha, and Jay K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In IEEE Society Press, editor, *Proc. IEEE Real-Time Systems Symposium*, pages 261–270, December 1987.

[18] V. Lorente, A. Espinosa, A. García-Fornes, and A. Crespo. Measuring execution time of code by means of posix tracing. In *27th IFAC/IFIP/IEEE Workshop on Real-Time Programming, WRTP'03*,

2003.

[19] J. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 1(44):73–91, 1995.

[20] A. Terrasa, A. García-Fornes, and I. Pachés. An evaluation of the posix trace standard implemented in *RT-Linux*. In *IEEE Intl Symposium on Performance Analysis of Systems and Software*, 2001.

Feather-Trace: A Light-Weight Event Tracing Toolkit*

Björn B. Brandenburg and James H. Anderson
The University of North Carolina at Chapel Hill

Abstract

We present a light-weight event tracing toolkit for real-time operating systems on the Intel x86 platform. Our approach is wait-free, multiprocessor-safe, and introduces very low overhead. Only a single unconditional jump instruction is required to distinguish between enabled and disabled events. As a case study, we traced the locking behavior of the Linux kernel and several soft real-time multimedia applications. Our results provide strong support for the wide-spread assumption that short non-nested critical sections are the common case in practice.

1 Introduction

When developing operating systems and embedded systems, *event tracing facilities* are an essential tool. Such facilities allow developers to trace the behavior of the system being developed by collecting performance and state data while the system in question executes for later offline analysis. The ability to better understand observed behaviors and to obtain high-resolution timing information greatly helps to both debug failures and improve performance. Thus, it is not surprising that there has been considerable recent interest in tracing frameworks [5, 7, 11, 19, 20].

Prior work. For general-purpose operating systems, powerful and flexible solutions have been developed and integrated into commercially-available products. For example, the DTrace facility of the Solaris 10 operating system, developed by Sun Microsystems [5], offers flexible dynamic instrumentation support. By embedding a virtual machine inside the kernel, it allows event data to be safely gathered and processed at arbitrary locations inside the kernel by compiled trace scripts. Such flexibility comes at a price, however. The DTrace implementation is complex and requires many operating-system services such as run-time symbol information, which may not be present in (space-constrained) embedded systems. Further, interrupts are disabled while executing trace scripts, which makes it unfit for use in real-time systems. Other dynamic instrumentation approaches based on binary re-writing such as kerninst [12] also require substantial in-kernel infrastructure. The K42 kernel [8] provides a lock-free, unified performance monitoring facility. While it provides a high-performance event tracing facility, its implementation is closely tied to the memory-management implementation of the K42 operating system, and thus cannot be easily ported to other operating systems. Disabled events incur an overhead of four instructions [19], some of which access main memory and affect branch prediction. Also, the use of potentially unbounded lock-free retry loops in K42's buffer implementation may restrict its applicability in hard real-time environments. The Ferret framework has been designed specifically for the Dresden Real-Time Operating System Project (DROPS) [10] and is based on a rather heavy-weight architecture. It is designed to allow tracing of real-time and best-effort tasks, system services, and the microkernel. However, the reliance on an event-structure description language and a custom tool chain restricts the portability of the framework. While tools that capture instruction-level execution traces such as Nirvana [1] provide a wealth of information for offline analysis, their use for obtaining real-world timing information is limited due to high overheads.

Motivation and contributions. In this paper, we present a light-weight, multiprocessor-safe tracing toolkit called *Feather-Trace*. Our motivations in producing this toolkit were two-fold. First, our research group has been engaged in an ongoing development effort involving a system called LITMUS$^{\mathrm{RT}}$ [3, 4, 17], which extends the base Linux kernel so that different scheduling and synchronization methods can be loaded as plug-in components. The primary focus of our LITMUS$^{\mathrm{RT}}$-related research has been scheduling and synchronization support for multiprocessor real-time systems. Our

*Work supported by a grant from Intel Corp., by NSF grants CNS 0408996, CCF 0541056, and CNS 0615197 and by ARO grant W911NF-06-1-0425. The first author was also supported by a Fulbright fellowship.

current development platform for LITMUS$^{\mathrm{RT}}$ is a four-processor machine. In order to debug scheduling and synchronization code in LITMUS$^{\mathrm{RT}}$, we needed a tracing mechanism that could be used on a multiprocessor with very low overhead, and that could be invoked anywhere in the kernel. We found that existing tracing mechanisms were ill-suited for our purposes. Second, in devising and evaluating synchronization mechanisms implemented in LITMUS$^{\mathrm{RT}}$ [2, 3], we desired to have a better understanding of locking patterns that are typical of "real-world" systems, so that we could optimize these mechanisms for common-case scenarios. Linux itself is certainly a real-world system, so we desired to trace its behavior to assess the frequency, duration, and degree of nesting in lock accesses. To validate our trace data, we also instrumented several soft real-time multimedia applications. Again, we found existing tracing facilities to be unsuitable for our purposes. Rather than providing a complete tracing framework, we found that our needs were best met by a highly-portable toolkit that can be easily integrated into existing operating systems with some "glue code." In Feather-Trace, trace events are checked via a single unconditional jump instruction, and trace data is collected in wait-free buffers that can be efficiently accessed on different processors. Although we were motivated by the specific concerns just noted in producing Feather-Trace, because it is very light-weight, can be used anywhere in the kernel, and it is portable, it should be of use to others engaged in kernel-related research. To the best of our knowledge, Feather-Trace is the first static tracing toolkit that achieves a single-instruction overhead in the case of both enabled and disabled tracing events.

The rest of this paper is organized as follows. In Section 2, we present Feather-Trace. In Section 3, as a case study, we present some measurements of the locking behavior of the Linux kernel and several soft real-time applications. Finally, in Section 4, we conclude.

2 Feather-Trace

To trace the execution of an operating system, a toolkit must provide methods to embed "triggers" in the program text and to collect data for offline analysis. The purpose of a *trigger* is to redirect the flow of execution to a user-provided *callback function* that can take appropriate actions such as collecting performance data or checking invariants for debugging purposes.

To be of practical use, several requirements must be met. First, it should be possible to selectively *enable* and *disable* triggers, since it is likely that only a specific aspect of an operating system is being inspected at any time. Second, no assumption concerning the execution context and preemptivity should be made so that triggers can be placed anywhere in the kernel, including interrupt handlers. Third, the framework should be multiprocessor-safe and it should not introduce additional mutual-exclusion requirements—by "multiprocessor-safe," we mean that tracing actions on one processor should not adversely affect other processors. Further, to increase portability and suitability for embedded platforms, only very little support should be required from the operating system. Of course, any overheads introduced by the tracing framework must be kept at a minimum. This implies that the trigger code should be short and affect neither cache performance nor the processor's branch prediction accuracy negatively. Ideally, a disabled trigger should incur no costs.

2.1 Event Trigger

In Feather-Trace, event triggers are realized as C preprocessor macros (`ft_eventX()`, where `X` is the number of arguments) that insert trigger code realized with inline assembly instructions. Thus, as is always the case with static instrumentation, events can only be added at compile time. While this may be an unacceptable limitation in the case of general-purpose operating systems such as Solaris, dynamic instrumentation has the disadvantage that enabled events incur the (considerable) costs of a CPU exception [10]. Also, if a custom exception handler were to be required, then adding tracing to an existing operating system would require intrusive modifications of its exception-handling code, thereby drastically increasing development effort.

The trigger code must accomplish three tasks. First, it must determine whether the event is enabled. If it is enabled, it must collect the necessary context information and invoke the callback function associated with the event. Finally, it must restore the processor context so that the original code surrounding the trigger can proceed correctly.

To achieve the goal of negligible overhead, the decision whether to invoke the callback function must be made as quickly as possible. Therefore, we chose the following approach (illustrated in Fig. 1): the `ft_eventX()` macro precedes the invocation of the callback function with an unconditional jump instruction (`jmp`) that skips over the rest of the trigger code. Thus, events are initially disabled. To enable an event, the offset parameter of the jump instruction is set to zero, which effectively disables the jump. As a result, the required context information is pushed on the stack and control is transferred to the callback function.

Since the trigger code is less than 128 bytes long, in

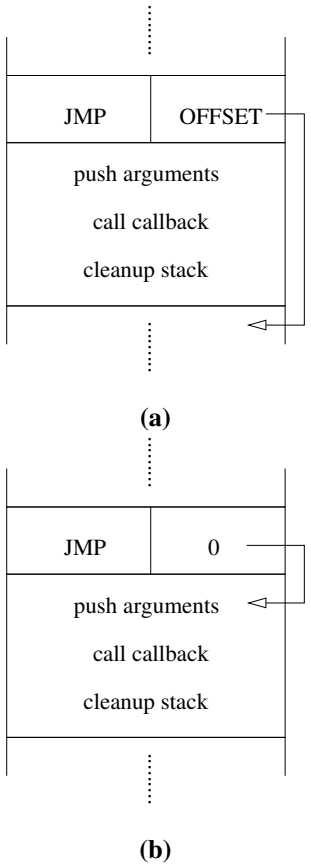


Figure 1: An illustration of the trigger assembly code. **(a)** In the disabled state, the jump instruction will skip the invocation of the callback. **(b)** In the enabled state, the jump instruction's offset is zero.

the Intel x86 instruction set, the unconditional jump including the offset can be encoded in two bytes. The jump instruction code in the first byte (`0xeb`) is followed by a signed eight-bit integer, which is the offset of the desired destination. To enable or disable an event, only the offset must be changed. Since eight-bit write operations to arbitrary byte-aligned addresses are guaranteed to be atomic on the Intel x86 platform, enabling and disabling events is multiprocessor-safe.

To summarize: events can be safely enabled and disabled on multiprocessors. No operating-system support is necessary and no locking/mutual-exclusion support is required. If an event is disabled, then only one additional instruction is executed compared to the case if there were no trigger code present. On the other hand, if an event is enabled, then only one additional instruction is executed compared to a normal function call. Determining whether a given event is enabled with only a single instruction that does not access memory (and which also has no effects on either branch prediction or pipelining)

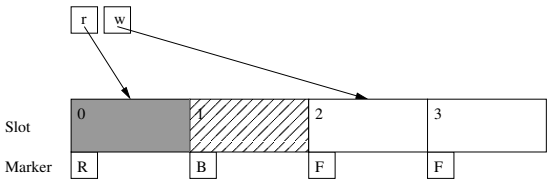


Figure 2: An illustration of a wait-free bufer for $n = 4$ and $f = 2$. Slot 0 is ready, Slot 1 is busy and being written, Slots 2 and 3 are free. The current read index r points to the next ready slot (Slot 0), and the current write index w points to the next free slot (Slot 2).

in both the enabled and the disabled case is arguably optimal. An additional small overhead can be incurred because the compiler may be forced to (re-)load some registers before and after the trigger code. This effect can be reduced by placing triggers mainly at the start and end of functions.

2.2 Data Collection

A tracing framework is of no utility if it does not offer a method to collect data. To keep the overhead of enabled events low, any trace data should be temporarily accumulated in an in-memory buffer and be transferred to stable storage after a certain number of samples have been obtained. To support multiprocessors, such a buffer must allow for multiple concurrent writers and, for similar reasons as is the case for triggers, should not rely on mutual exclusion to achieve correctness. While read operations should be possible in parallel with write operations, there is usually no great need for multiple readers, since typically a single reader is tasked with flushing the buffer to stable storage.

To attain the stated goals, Feather-Trace provides a *wait-free* FIFO-buffer implementation to store event data. The buffer is said to be wait-free since no locks are required and each read and write operation completes in a bounded number of steps (such is not the case when lock-free retry loops are used). Our implementation supports arbitrarily many concurrent writers. To simplify the data structure and to improve performance, we allow only one concurrent reader.

As illustrated in Fig. 2, a buffer consists of n *slots*. Slots may be of arbitrary but uniform size s. Each slot is associated with a *slot marker* that indicates the current state of the slot. A slot may be either *free*, *busy*, or *ready*. For each buffer, the number of free slots f (a signed 32-bit integer), the current write index w, and the current read index r (both unsigned 32-bit integers) are maintained. We require that n divides the maximum value that an unsigned 32-bit integer can store, *i.e.*,

```
unsigned int r = 0, w = 0, e = 0;
int f = n;

start_write(void **ptr) {
  unsigned int idx;
  if (fetch_and_dec(f) <= 0) {
    /* buffer full */
    atomic_inc(f);
    atomic_inc(e);
    *ptr = NULL;
    return 0;
  } else {
    /* slot reserved */
    idx  = fetch_and_inc(w) % n;
    marker[idx] = SLOT_BUSY;
    *ptr = &slot[idx];
    return 1;
}

finish_write(void *ptr) {
  unsigned int idx;
  idx = (ptr - &slot[0]) / s;
  marker[idx]  = SLOT_READY;
}

read(void* dest) {
  unsigned int idx;
  if (f == n)
    /* nothing available */
    return 0;
  idx = r % n;
  if (marker[idx] == SLOT_READY) {
    memcpy(dest, &slot[idx], s);
    marker[idx] = SLOT_FREE;
    r++;
    atomic_inc(f);
    return 1;
  } else
    return 0;
}
```

Figure 3: Pseudo-code for the methods used to access the wait-free buffers provided by Feather-Trace.

$2^{32} \bmod n = 0$. This allows us to ignore integer overflows, which is a minor performance improvement.

To detect missed samples, the number of failed writes is stored in the error count e. (A write fails if the buffer is full.) Pseudo-code for the buffer access methods is given in Fig. 3. The implementation relies on the atomic `XADD` ("exchange and add") instruction, which is used to realize `fetch_and_dec/inc()` and `atomic_inc()`. Writers access the buffer by first invoking `start_write()` to obtain a pointer to a free slot. A slot is reserved in two steps. First, the number of free slots f is read and decremented atomically to reserve a slot. If a reservation can be made ($f > 0$ holds), then the next free slot is obtained by atomically reading and incrementing w. Since n divides 2^{32}, a potential overflow of w does not need to be handled. The slot is marked as busy to prevent a concurrent reader from observing incomplete data. If no slot is available, then the reservation is canceled by atomically incrementing f and the error count e. The single reader accesses the buffer by first checking whether there exist non-free slots by comparing f and n. If there exists such a slot, then the reader checks the slot's state, and if the slot is ready, copies the slot's contents to a reader-provided location such as I/O buffers.

The multi-writer, single-reader, wait-free FIFO buffer provided by Feather-Trace offers a low-overhead method to store uniformly-sized data items. The limitation of uniformly-sized items can be easily dealt with by providing several buffers of different sizes. As both the provided event triggers and FIFO buffers are designed to minimize overheads, Feather-Trace can be used to trace highly performance-critical code sections. For example, as explained in more detail in the next section, we have used the toolkit to measure critical section lengths in the Linux kernel. This was made possible in part because the code used to obtain and store time stamps, including the event trigger, consists of only 61 instructions, which is a negligible overhead in most cases.

Since the toolkit is minimally intrusive and makes no assumptions on the availability of operating-system services, it can be easily integrated into existing code bases. For example, we have used Feather-Trace to obtain event traces in both the Linux kernel and the FreeBSD kernel by implementing a custom device driver that exports the accumulated event data to user space. Further, by pre-loading Feather-Trace (packaged in a shared library) into dynamically linked user space applications, we were able to record the locking behavior of various soft real-time multimedia applications.

3 Case Study: Locking in Linux

One motivation for the development of Feather-Trace was to allow us to obtain empirical results on the frequency, degree of nesting, and duration of critical sections in "real-world" systems. In prior work, Devi *et al.* [6] measured the length of critical sections accessing common data structures in order to generate task sets for schedulability-analysis purposes [6]. The method employed by them, however, cannot give insight into the nesting depth and the distribution of lock requests, as it relies on measuring synthetic tasks. Other studies have assessed the impact of lock-free synchronization on large scientific applications [15, 16]. Unfortunately, these benchmarks are mostly concerned with overall performance and do not reveal the nature of individual critical sections. In this paper, we seek to provide additional data points on "real-world" locking behavior by measur-

ing critical sections in both the Linux kernel under various workloads and several soft real-time applications.

In the following subsections, we say that a lock request has a *nesting depth* of n if the processor already was holding n locks at the time of the request. Further, we define the *critical section length* of a lock to be the length of the time interval that starts when the lock is successfully acquired and ends just before it is released again, *i.e.*, the cost of acquiring the lock itself is not included.

3.1 Kernel Modifications

We modified the Linux kernel, version 2.6.20, to capture timing information on critical section lengths. The Linux kernel employs two different kinds of locks, *spinlocks* (contention is handled by busy-waiting) and *semaphores* (processes are suspended in case of contention). To trace spinlocks, we changed the locking primitives `spin_lock()`, `read_lock()`, `write_lock()`, and the corresponding unlock primitives (as well as special cases such as `spin_lock_irqsave()`) to include event triggers after a lock has been acquired and before a lock is released. To trace semaphores, we modified `mutex_lock()`, `down()`, and related primitives such as `down_read()` in a similar fashion. At each event, a time stamp was obtained by reading the `TSC` register (which can be read from both user and kernel space) and the sample consisting of the time stamp, the CPU on which the event occurred, the address of the lock involved, and the type of the event (enter critical section, or exit critical section) was stored in a Feather-Trace buffer. That buffer was made available to user space by means of a custom character device driver.

3.2 Setup of the Experiments

To obtain insight into the kernel's locking behavior, we executed various test workloads and captured locking events during several intervals of 60 seconds each. Our particular test platform is an SMP consisting of two 32-bit Intel(R) Xeon(TM) processors running at 2.70 GHz, with 8K instruction and data caches, and a unified 512K L2 cache per processor, and 2 GB of main memory.

The results of three workloads are presented in this paper. First, we obtained a trace from an otherwise idle system. Second, we traced the locking behavior of the Linux kernel while compiling a copy of the kernel itself. Last, we used the `stress` utility [18] to generate a test load of three processes that stressed the memory management and I/O subsystems of the kernel.

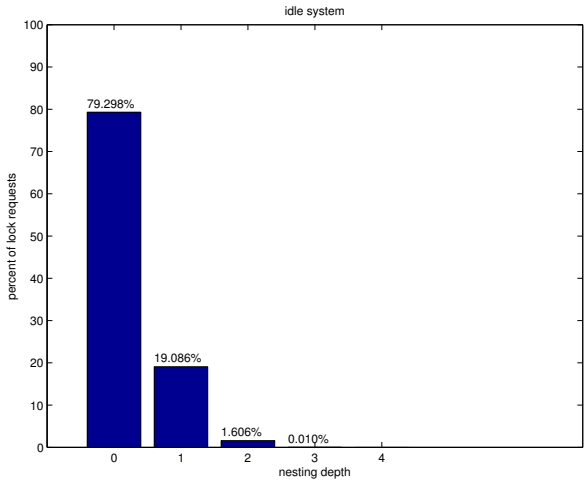
The captured event traces were analyzed offline as follows. After filtering incomplete event sequences (*i.e.*, lock accesses that were missing one of the two expected timestamps), the remaining lock requests were annotated with their respective nesting depth. Incomplete sequences may occur at both the beginning and the end of the trace interval and when there is insufficient buffer space available. The clock speed of the processors (2.70 GHz) was used to convert raw cycle-count timestamps to microseconds. Finally, histograms of the nesting level and the critical section length (with a bin size of 0.1μs), the cumulative distribution, and the average critical section length were computed for both spinlocks and semaphores.

3.3 Results

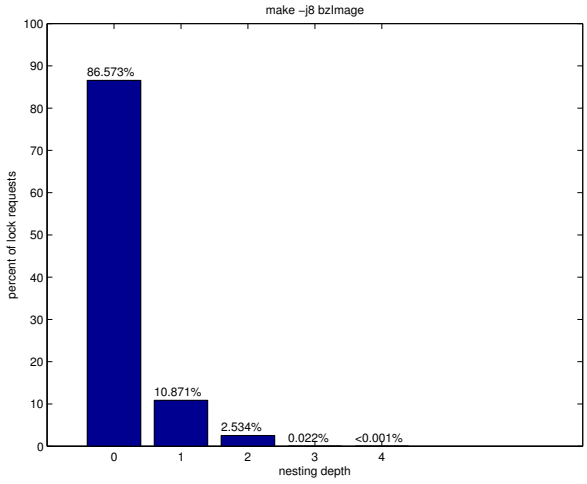
After filtering, the traces contained valid events for a total of 2,366,122 (idle system), 25,002,249 (compile test), and 70,807,495 (stress test) spinlock and 51,360 (idle system), 4,880,570 (compile test), and 18,998,386 (stress test) semaphore acquisitions.

The distribution of the spinlock nesting depth is shown in Fig. 4. The maximum nesting depth in the presented data is four under load and three on an idle system. One can clearly see that the vast majority of lock acquisitions are non-nested, *e.g.*, under load, more than 85 percent of all lock requests have a nesting depth of zero. When comparing the nesting depth distribution of an idle system to the distribution observed during the stress test, one can see two trends. First, maximum nesting depth increases from three to four (nesting depths as deep as six have been observed in traces not presented here, but occur so rarely that they are hard to reproduce), which reveals that there exist deeply nested lock requests that are only required very seldomly. Second, the percentage of non-nested lock requests increases under load, which can be attributed to the fact that the number of shared objects that are mostly accessed in a non-nested fashion increases under load. As can be seen in Fig. 6, critical sections protected by semaphores are less frequently subject to nesting as those protected by spinlocks. Semaphore requests exceeding two levels of nesting were not observed in any of the traces.

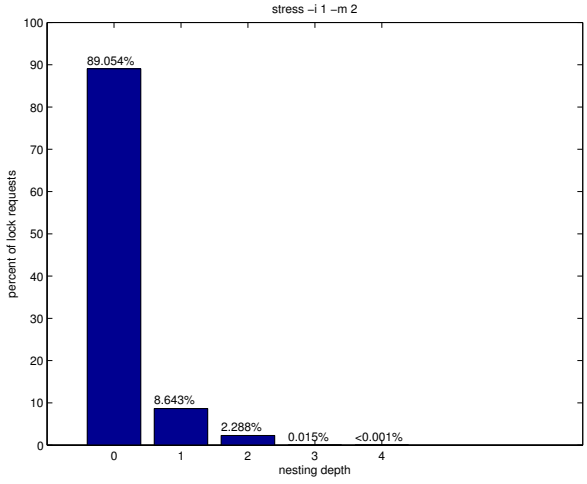
In Fig. 5, the distribution of spinlock critical section lengths is depicted. While the distributions do have a long tail, more than 96 percent of all observed critical sections are shorter than 5μs. Inset (a) depicts the distribution of an idle system. Since system-call activity is low, most lock requests are issued by interrupt handlers. One can clearly see two distinctive spikes as a result, because the critical sections encountered in periodic activities such as the timer-interrupt service routine contribute the majority of observed critical sections. The average critical section length observed in an idle



(a)

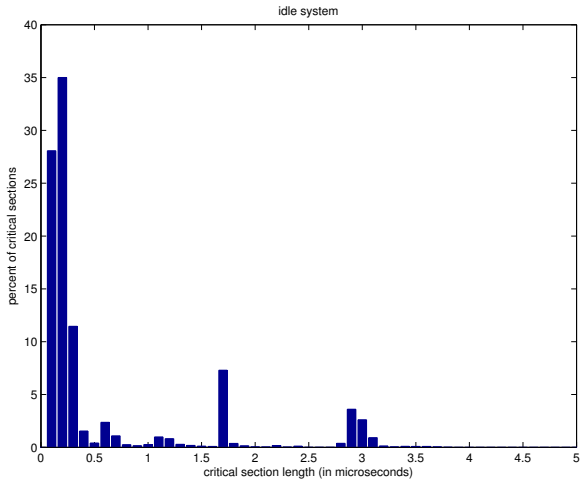


(b)

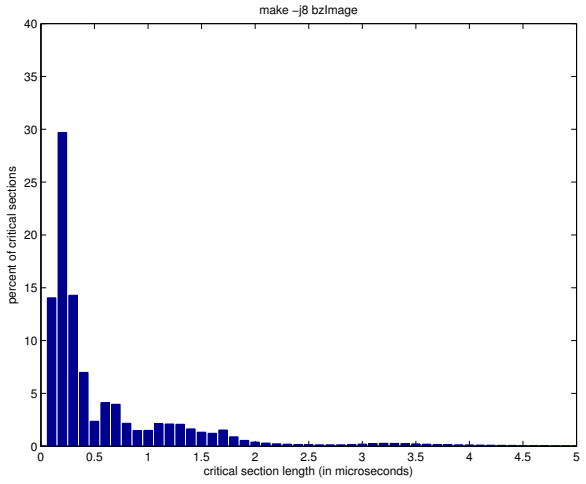


(c)

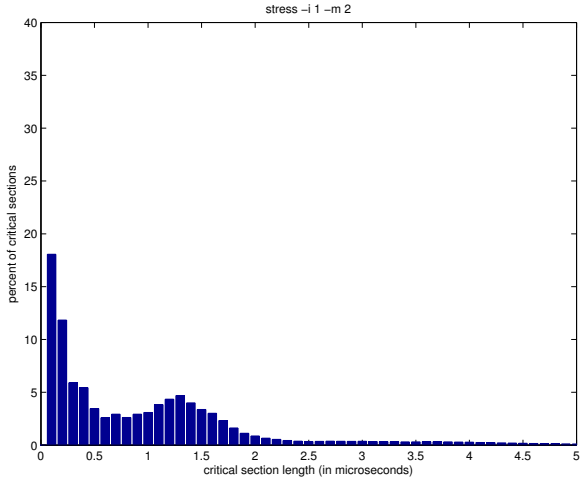
Figure 4: Distribution of nested spinlock accesses in the Linux kernel under various work loads.



(a)

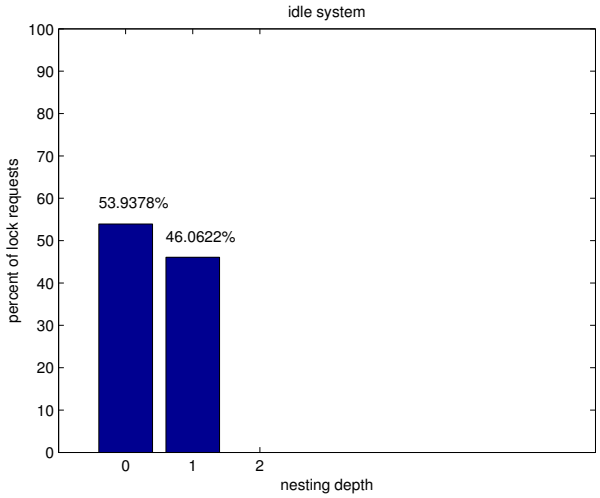


(b)

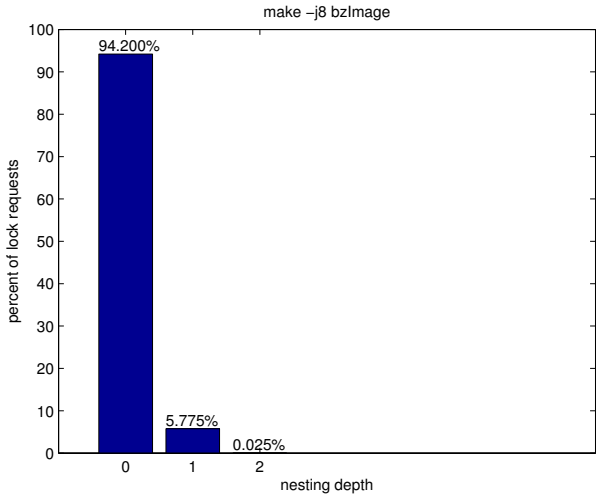


(c)

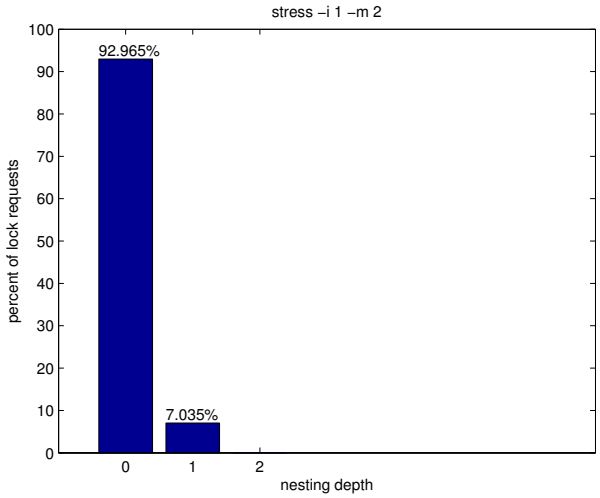
Figure 5: Distribution of spinlock critical section length in the Linux kernel. More than 96 percent of all observed critical sections were shorter than $5\mu s$.



(a)

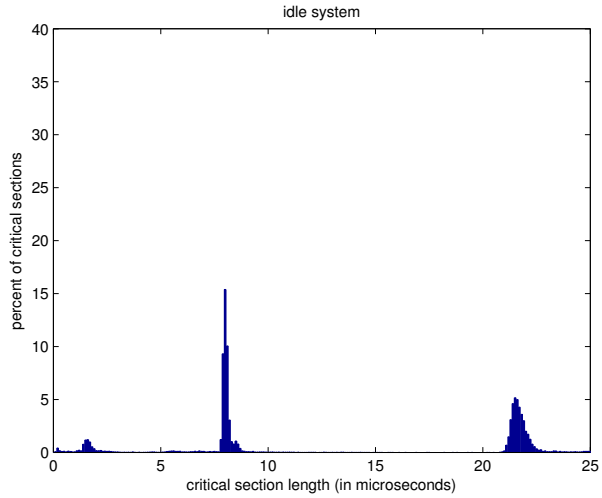


(b)

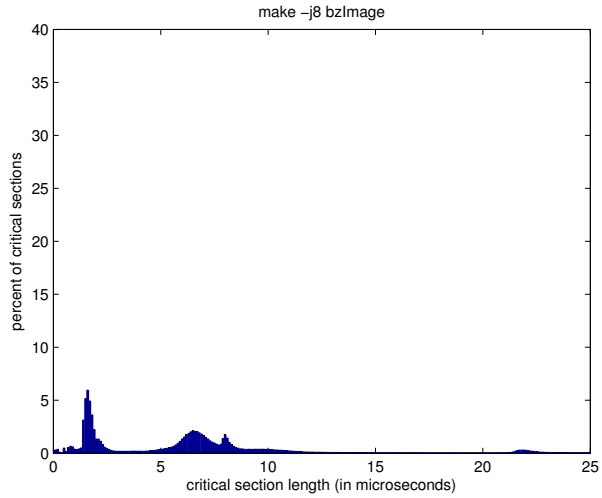


(c)

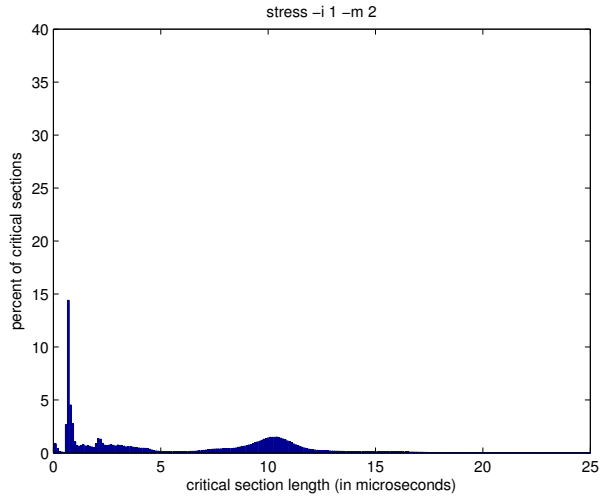
Figure 6: Distribution of nested semaphore accesses in the Linux kernel under various work loads.



(a)



(b)



(c)

Figure 7: Distribution of semaphore critical section length. Under load, more than 93 percent of all observed critical sections were shorter than $13\mu s$.

system was 0.67μs. The distribution of critical section lengths observed in the compile benchmark is shown in inset (b). The impact of the periodic activities decreases noticeably compared to an idle system. As a result of the kernel actually doing "real" work on behalf of user-space processes, the critical section lengths are spread out over a wider range. The average critical section length observed in this benchmark increased to 0.81μs. The trend continues in inset (c), which depicts the distribution observed under the stress test. In this case, the kernel has to service many "expensive" system calls, so that the center of the distribution is shifted noticeably to the right. The average observed critical section length was 1.24μs. In Fig. 7, the distribution of semaphore critical section lengths is shown. Critical sections protected by semaphores are typically significantly longer than those protected by spinlocks. The average observed critical section lengths were 6.3μs (stress test), 6.4μs (compile test), and 14.9 μs (idle system).

3.4 Soft Real-Time Applications

Since the Linux kernel may not be representative of real-time applications, we conducted similar experiments with several multimedia applications running on top of Linux 2.6.23-rc3 to ensure the validity of our conclusions. The results for three of the benchmarks are shown in Figs. 8 and 9. (Because the distributions did not contain characteristic spikes, we chose to present them as cumulative distributions instead.) With each tested application, we used Feather-Trace to instrument the acquisition and release of user space binary semaphores as provided by the POSIX thread (`pthread`) library. As was the case with the kernel, cycle-count timestamps were used to determine the beginning and end of a critical section. The data depicted in Insets (a) and (b) was obtained by instrumenting two popular open source video players (Video Lan Client (VLC) [13] and Xine [14]) over a period of one hour. Inset (c) shows the behavior of Tux Racer [9], an interactive 3D video game, over a period of about one minute. Since these applications need to ensure that both visual and audio content is presented to the user in a timely manner they can be considered to be soft real-time applications.

Fig. 8 clearly shows the nesting characteristics of the three applications. While nesting almost never occurs in Tux Racer, and only very rarely in Xine, it is used more commonly in the VLC video player. However, non-nested accesses, which make up more than 70 percent of the critical sections, are still the common case. A nesting level greater than three was never observed in the tested multimedia applications.

Distributions of critical section lengths are depicted in Fig. 9. As opposed to the nesting levels, the cumulative critical section length distributions of the instrumented applications are somewhat similar. In all cases, more than 95 (99) percent of the critical sections are shorter than 5μs (10μs). This indicates that critical sections in multimedia applications are typically even shorter than those observed in the kernel. This observations is also supported by a significantly lower average critical section length (compared to the average length of in-kernel semaphore-protected critical sections).

Our results strongly support the wide-spread assumption that the vast majority of critical sections in many settings are short and non-nested. While deep nesting does occur in practice, nesting depths of three or more occur only rarely. Critical sections longer than 5μs (13μs) are rare in the case of spinlocks (semaphores, under load) in the kernel. In multimedia applications, they tend to be even shorter. Thus, our data supports the claim that the common case in practice is short, non-nested lock requests.

4 Conclusion

This paper presented Feather-Trace, a new light-weight static tracing toolkit that is both highly portable and can be used for performance data collection as well as debugging purposes. Because Feather-Trace uses neither locks nor retry loops, it is suitable for hard real-time environments. Further, since disabled events incur only the negligible overhead of one additional instruction per event, there is no need to remove Feather-Trace in production releases.

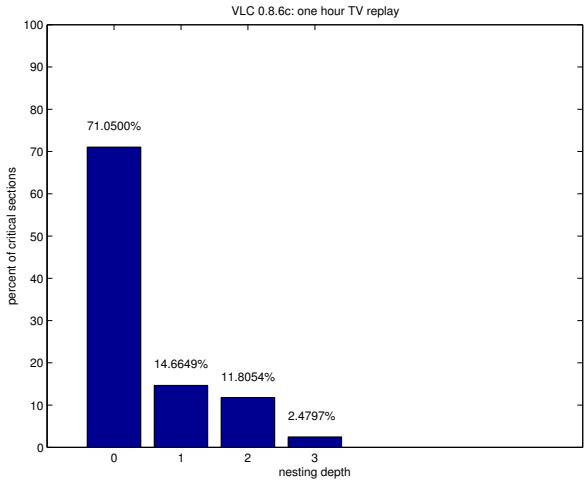
As a case study and to support our ongoing work on multiprocessor real-time synchronization, we used Feather-Trace to obtain the frequency, duration, and degree of nesting in lock accesses in both the Linux kernel under various workloads and soft real-time applications. Our measurements strongly support the wide-spread assumption that short, non-nested critical sections are by far the common case in practice.

As mentioned in the introduction, we believe that Feather-Trace may be of interest to a wider audience of embedded and real-time systems developers. Our implementation is available under a permissive open source license at the first author's home page[1].

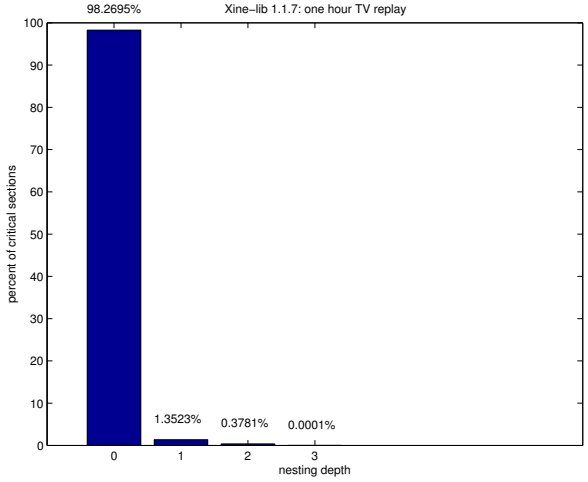
References

[1] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihoĉka, and J. Chau. Framework for instruction-level tracing and analysis of pro-

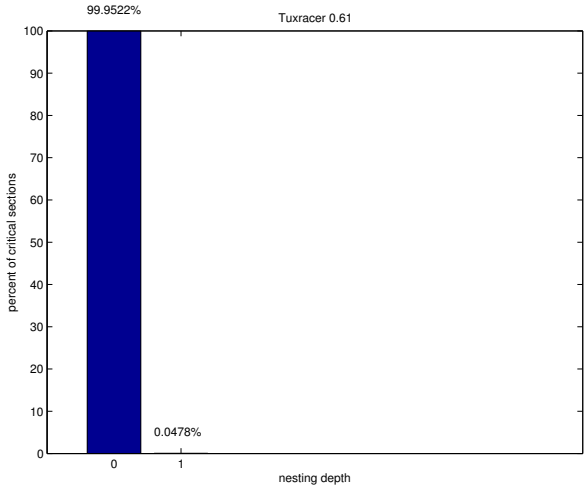
[1] http://www.cs.unc.edu/~bbb/feathertrace/ .



(a)

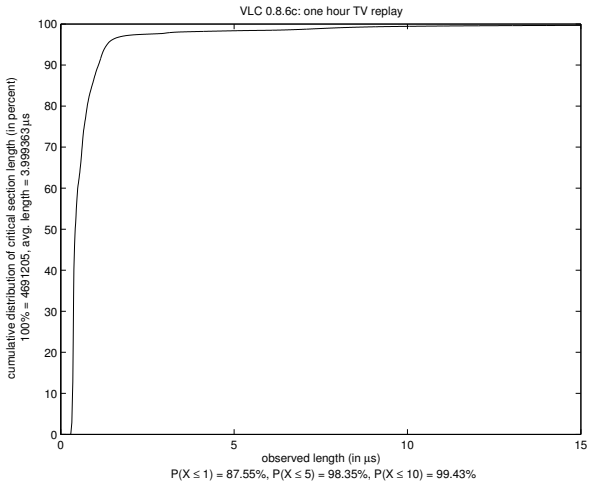


(b)

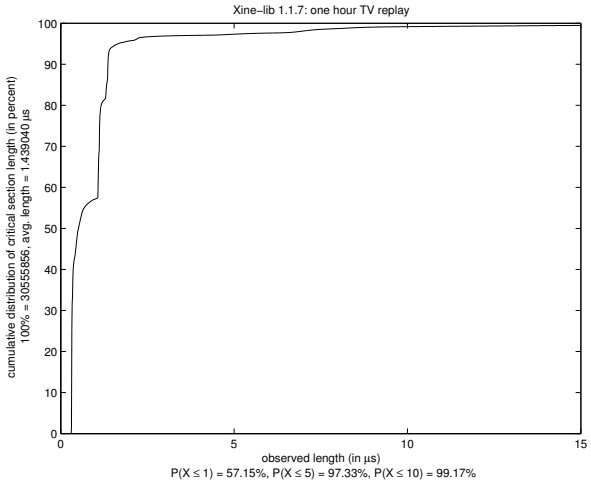


(c)

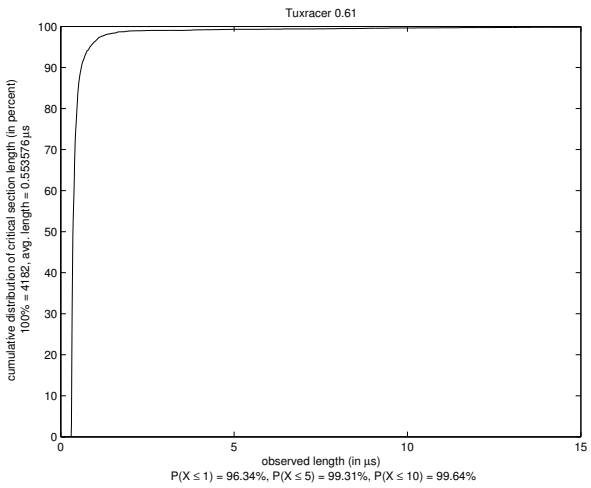
Figure 8: Distribution of nested mutex accesses in multimedia applications.



(a)



(b)



(c)

Figure 9: Distribution of mutex critical section length in multimedia applications. Note, that 99% of the critical sections were shorter than $10\mu s$.

gram executions. In *VEE '06: Proceedings of the second international conference on Virtual execution environments*, 2006.

[2] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, August 2007. To appear.

[3] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson. Synchronization on real-time multiprocessors: To block or not to block, to suspend or spin? In submission, 2007.

[4] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, 2006.

[5] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of USENIX '04*, 2004.

[6] U. Devi, H. Leontyev, and J. Anderson. Efficient synchronization under global EDF scheduling on multiprocessors. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, 2006.

[7] IBM Linux Technology Center. Dynamic probes. Homepage. http://dprobes.sourceforge.net/.

[8] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. D. Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: Building a complete operating system. In *Proceedings of EuroSys 2006*, 2006.

[9] J. Patry. Tux Racer. Homepage. http://tuxracer.sourceforge.net/.

[10] M. Pohlack, B. Döbel, and A. Lackorzynski. Towards runtime monitoring in real-time systems. In *Proceedings of the Eighth Real-Time Linux Workshop*, 2006.

[11] Red Hat, IBM, Intel, and Hitachi. System tap. Homepage. http://sourceware.org/systemtap/.

[12] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, 1999.

[13] The VideoLan Project. VideoLan Client. Homepage. http://www.videolan.org/.

[14] The Xine Project. Xine Libraries and UI. Homepage. http://xinehq.de/.

[15] P. Tsigas and Y. Zhang. Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the 2001 ACM SIGMETRICS Int'l Conf. on Measurement and Modeling of Computer Systems*, 2001.

[16] P. Tsigas and Y. Zhang. Integrating non-blocking synchronisation in parallel applications: performance advantages and methodologies. In *Proceedings of the the Third Int'l Workshop on Software and Performance*, 2002.

[17] UNC Real-Time Group. LITMUS$^{\mathrm{RT}}$ project. Homepage. http://www.cs.unc.edu/~anderson/litmus-rt/.

[18] A. Waterland. stress. Homepage. http:// weather.ou.edu/ ~apw/projects/stress/.

[19] R. W. Wisniewski and B. Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *Proceedings of SC 2003*, 2003.

[20] K. Yaghmour. Linux trace toolkit. Homepage. http://www.opersys.com/LTT/.

A Deterministic Infrastructure for Real-Time Distributed Systems

Claudiu Farcas
Calit2, University of California
San Diego, USA
cfarcas{at}soe.ucsd.edu

Wolfgang Pree
C. Doppler Lab Embedded Software Systems
University of Salzburg, Austria
wolfgang.pree{at}cs.uni-salzburg.at

ABSTRACT

The development of reliable software for real-time systems is currently a challenge. Moreover, changing the underlying platform for simple purposes such as a processor upgrade may severely affect the behavior of the real-time software. Working with distributed systems is even more difficult, and transitioning from one system to another is typically impossible.

We address these problems through a development framework for deterministic and portable real-time software using the Timing Definition Language (TDL). It enables transparent, yet deterministic distribution of real-time components regardless of the target platform and its deployment architecture. In this paper, we introduce the algorithms and internal mechanisms for transparent real-time distribution, and analyze the interactions between the user-functionality, the virtual machine of the language, the communication subsystem, and the underlying platform.

1. INTRODUCTION

The advances in computational hardware and the corresponding promises for real-time process control make "embedding" computers into many systems a common practice. Complex applications typically require distributed systems for reasons of dependability (fault-tolerance), scalability, localization. A distributed system may be more reliable than a single node system as through replication faults on a node may be corrected by the other nodes; thus, maintaining a high degree of dependability of the overall system. It can also be extended by adding more processing nodes to solve a computational-intensive job, in comparison with the case of a single node system where a more powerful processor may be too costly, require too much power, or simply be unavailable. On the other hand, the complexity of the distributed systems is several orders of magnitude more significant and harder to deal with than a single-node system. Migrating from a single-node solution to a distributed system is hardly possible with the traditional development methodologies for real-time systems. Even simple changes in the topology of a distributed system or addition of new nodes become a challenge in most applications.

To address these problems our approach for real-time distribution relies on a high-level component-oriented language that makes the timing an explicit part of the real-time software design and decouples the timing from the implementation of the computational tasks of an application. The Timing Definition Language (TDL) [17, 5] is a high-level description language for specifying the explicit timing requirements of a time-triggered [11] application, which may be constructed out of several independently developed components. The actual functionality can be implemented in any imperative language available for the target platform, e.g. C, C++, Java, and later linked with the compiled TDL source. TDL relies on the *Logical Execution Time* (LET) abstraction introduced in the Giotto language [7], but goes beyond with a component model, improved syntax and semantics, and full support for distribution.

LET means that the observable temporal behavior of a computational task is independent from its physical execution. The LET of a TDL computation is always equal with its invocation period and we only assume that its physical execution is fast enough to fit somewhere within the logical start and end points. Thus, it is always defined which value is in use at which time instant and there are no race conditions or priority inversions involved. LET introduces a *unit-delay* behavior [7], which may appear as a disadvantage. However, it provides determinism, composition [9], and platform abstraction, which are more relevant for safety-critical systems.

This paper focuses on the algorithms and mechanisms for transparent hard real-time distribution and the interplay of the components of the run-time system. We present the distribution from the logical point of view of the developer and then from the underlying run-time system of TDL. We introduce an algorithm to bridge the gap between the task and network communication scheduling, and detail the interactions between the user-functionality, the virtual machine of the language, the communication subsystem, and the underlying platform. We introduce an abstraction layer for distribution and present the algorithms for data encapsulation and state synchronization across the network.

We begin with an overview of the TDL component model and its capabilities for structuring complex real-time applications. Section 3 describes the notion of transparent distribution available with TDL and briefly presents the development tool-chain. In Section 4, we analyze the internals of the TDL run-time system, and introduce the algorithms that govern the interactions and synchronization of its constituents, namely the TDL scheduler, the virtual-machine (E-Machine) for logical timing, and the communication layer TDL-Comm. An evaluation in Section 5 illustrates our approach through a running example. We complete the article with a section of related work and our conclusions.

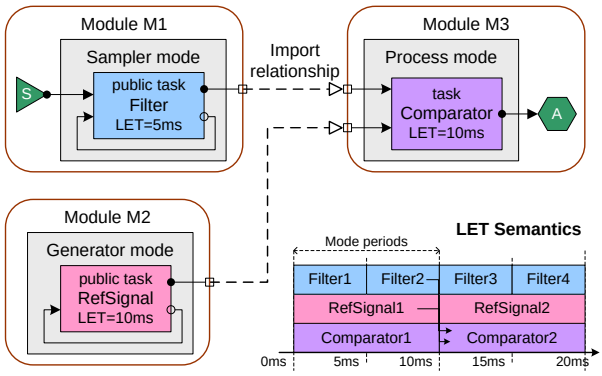


Figure 1: TDL Modules and LET semantics

2. TDL COMPONENT MODEL

The TDL component model relies on the concept of a *module*, which may encapsulate an entire application or parts of a complex application.The TDL modules may work independently or may collaborate to implement a complex behavior. The component model allows for decomposing existing complex applications into smaller, more manageable parts, each with specific timing and functionality, and provides the means for deterministic component interaction. Developers can reuse existing modules to extend the functionality of an application or create new applications.

The TDL module acts as the unit of composition and distribution, and may *import* one or more other modules as depicted in Figure 1. Computation results or environment state can be exported from a module to any other module that imports it. Typically, a developer may import the values of some task output ports by declaring the corresponding task as *public*. This feature introduces data dependencies between modules as one module provides data services to other client modules. It is important to note that LET is always preserved, i.e., adding a new module to an application will never affect the observable temporal behavior of other modules. The TDL compiler performs schedulability analysis using the worst-case execution time for tasks on the target platform and issues an error if LET cannot be maintained (e.g., wcet is too large).

Modes.. The *mode* of a module is a set of periodically executed activities such as task invocations, actuator updates, and mode-changes. The period of an activity within a mode is equal with the mode period divided by the invocation frequency of that activity within the mode. A module has an unique *start* mode.

The mode-change protocol of TDL is different from Giotto, requiring new schedulability analysis [4]. A TDL module changes at run-time its mode independently of other modules. Within a module, TDL enforces *harmonic* mode switches – the mode switch must not break the LET of any task invocation within the current mode of the module. This restriction enables deterministic mode switches in distributed applications. Furthermore, mode switches in a module may break the LET of tasks from other modules, which are not affected by the mode switch. TDL mode changes are regarded as instantaneous.

Tasks are computational units in TDL. A task has a set of input, state, and output ports, along with an external implementation referred through symbolic linking. A *task invocation* within a mode represents the execution of a task instance within the period of that mode. TDL regards the tasks as *scheduled* elements with logical execution time [9]. From the logical point of view, a task reads its inputs at the *release* time, then it runs continuously until its *termination* time, when its computation results are available to the environment and other tasks; whereas, from the platform point of view, the task starts at some point in time after it was released, may be preempted by some other tasks or the RTOS, and completes before the end of its LET. The underlying assumption, which we have to verify [4], is that the run-time system and the scheduling mechanisms used for the physical execution allow each task to complete before its deadline.

Sensors and actuators exchange information between the environment and the tasks of a module. TDL assumes that the external functional implementation of the sensor getters and actuator setters executes in *logical zero time*, i.e., orders of magnitude faster than the smallest task computation. Practical implementations may simply read or write to dedicated memory locations or I/O ports.

Guards are lightweight Boolean functions operating on sensor or task output values. Depending on their result at run time, they condition the execution of corresponding tasks, actuator updates, or mode switches.

Ports interface TDL entities within a module and between modules. There can be input, output, or state ports, each with a distinct type (int, byte, float, etc.). Only tasks can have state ports. To implement the LET mechanism, the TDL tasks have two copies of the output ports: internal and visible. The internal output ports are updated directly by the task functionality code, whereas the visible ports are updated through drivers by the TDL runtime environment at the end of the LET of the task.

The *drivers* as introduced by Giotto are no longer syntactically explicit in TDL. Nevertheless, a TDL driver still performs the port-value copying operations under the LET semantics. The improved syntax of TDL allows the TDL compiler to automatically perform the type checking between ports and then generate the drivers which transport the port values between the interconnected TDL entities.

3. TRANSPARENT DISTRIBUTION

We use the term *transparent distribution* [5] in the context of hard real-time application with respect to two aspects. Firstly, at runtime a TDL application behaves exactly the same, no matter if all modules (i.e., components) are executed on a single node or if they are distributed across multiple nodes. The logical timing is always preserved, only the physical timing, which is not observable from the outside, may be changed. Secondly, for the developer of a TDL module, it does not matter where the module itself and any imported modules are executed. The TDL tool-chain and runtime system frees the developer from the burden of explicitly specifying the communication requirements of modules. It should be noted that in both aspects transparency applies not only to the functional but also to the temporal behavior of an application.

The advantage of transparent distribution for a developer is that the TDL modules can be specified without having the execution on a potentially distributed platform in mind. The only place where distribution is visible is for the system integrator, who must specify the module-to-node assignment.

The development process for TDL relies on the tool-chain from Figure 2. It consists of the following functional components: a TDL compiler, a visual editor fully integrated with the Matlab/Simulink environment, and a corresponding run-time environment. The TDL compiler has a plug-in architecture, which allows its extension with other tools such as automatic glue-code and bus schedule generators for a target platform. Worst-case execution analysis [20] can be plugged into the visual editor to enable schedulability analysis within the TDL compiler. In this paper, we focus on the TDL runtime environment and briefly mention the relevant aspects of the other tools.

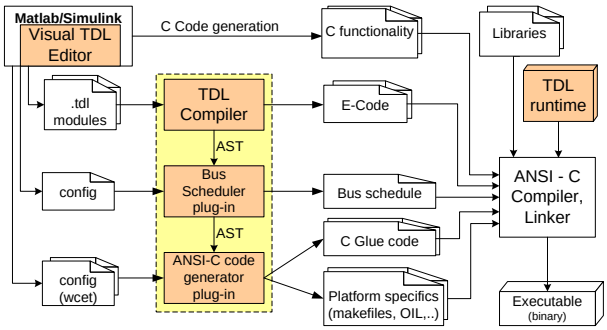


Figure 2: TDL tool-chain for distribution

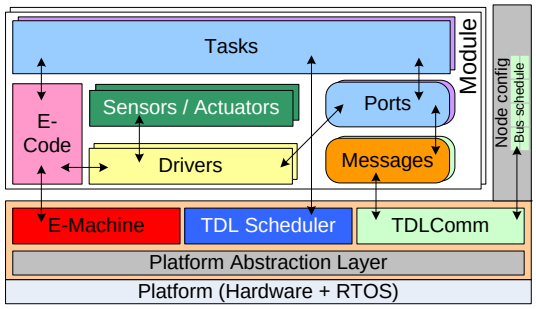


Figure 3: Run-Time Environment

For distributed systems, the Bus-Schedule Generator tool [6] is a compiler plugin that automatically compiles offline the communication schedule. Its configuration file contains the list of computing nodes, the assignment of TDL modules to nodes, and the properties of the communication channels (e.g., bus rate, minimum and maximum packet sizes). The tool analyzes the import relationships in the TDL modules to identify their remote dependencies and the set of messages required for exchanging the information between producer tasks and consumer entities such as tasks, actuators, or guards. It then tries to generate a TDMA communication schedule that satisfies the requirements of the TDL modules and their mode changes. The schedule specifies which node sends information at which time; the structure of the information depends on the current modes at run-time [6].

As a first step towards fault-tolerance TDL supports module replication. The replicas are identified from the module-to-node assignment in the configuration file of the tool. We send the messages produced in all service-provider modules and we process them in all their stubs through majority voting. By scheduling the replicated messages as any other message, the tool also achieves temporal isolation between replicas that improves the recovery chances from transient failures.

4. RUN-TIME MECHANISMS

We introduce in Figure 3 the TDL runtime environment consisting of three logical components deployed on each node: virtual machine, scheduler, and communication layer. The virtual machine supervises the logical behavior of the application and its interaction with the environment. The TDL Scheduler performs the mapping of platform time to logical time, the invocation of the virtual machine and the communication layer, and the preemption and dispatching operations of the user tasks. The communication layer handles the distribution aspects.

4.1 E-Machine

For portability reasons, TDL reuses the approach of a virtual machine, the E-Machine introduced in Giotto [8], to handle the logical aspects of its runtime environment. In addition to Giotto, the TDL E-Machine handles parallel and distributed modules. It executes a small set of instructions (TDL E-Code [17]) related only with the logical aspects of a module: when and which tasks to release, and which drivers to execute for the correct information flow between ports. The functionality of the module runs in the native code of the platform for maximum performance.

In distributed systems, a *service-provider* module and its *client* modules (there can be more than one module importing a service provider module) may be placed on different nodes. The TDL compiler generates a *stub* of the service-provider module on each node that does not contain it but contains one of its clients. A stub module is a logical image of a service-provider module. It does not contain any functionality for tasks, but only their output ports. This concept enables the seamless distribution of modules in the system and improves the performance on the nodes containing the client modules as they do not have to locally execute the service module. Nevertheless, the communication layer must synchronize the state (mode, port values, timing) of a service provider module with all its stubs (see Section 4.3).

From a logical point of view, the E-Machine executes the E-Code in logical zero time, for each module individually, regardless of its type, i.e. "normal" module or stub module. From the platform point of view, the TDL Scheduler that invokes the E-Machine accounts the time spent interpreting E-Code and adjusts its decisions accordingly.

The TDL compiler generates an E-Code file per module as a compact representation of the activities defined in that module and their timing information. It also includes the dependency information related with the import relationships between modules. The Code-Generator Plugin converts this information into the corresponding drivers and associated glue-code. Thus, the E-Machine can simply execute the E-Code instructions and call the appropriate drivers to ensure the correct intra- and inter-modules data flow.

4.2 TDL Scheduler

The TDL Scheduler is the actual bridge between the TDL semantics and the underlying platform (real-time operating system, computing hardware, and distributed system architecture). Its purpose is to run the E-Machine for each module at the right time as defined by the TDL semantics and then to execute the released tasks according to a specific scheduling policy. In a distributed setup, it furthermore has to coordinate the exchange of data between the nodes via the TDLComm layer.

To support the parallel composition of modules and to execute them on the same node, we have to allocate a fraction of the CPU time to each module of the node. Traditionally, we would solve this problem with clock-driven scheduling [13] via a static time-sharing mechanism, or CPU partitioning. Within a scheduling cycle of the GCD of all activity periods from all modules, we would allocate for each module a time quantum proportional with the maximum load the module generates on the CPU. However, this approach introduces a high context-switch overhead on the running system, because of the infinitesimal time quantum required to implement this mechanism.

As alternative, we could use a on-line scheduling algorithm, such as Rate Monotonic (RM) or Earliest Deadline First (EDF) [12]. In the following, we present a scheduling approach for TDL using EDF on the global set of tasks from all modules. Conceptually, the modules are simply logical constructs; hence, from a scheduling

perspective we can treat equally all tasks from all modules. The only property that matters for EDF scheduling is the deadline of the task, regardless of its parent module. In addition, as the application is strictly time-triggered and we know at compile time all task and mode periods, we can benefit from this apriori information to reduce the run-time scheduling overhead by using precompiled tables. The TDL Scheduler avoids unnecessary context switches by running only when it has to invoke the E-Machine/TDLComm or when a task completes. Consequently, it allows a better CPU utilization than the partitioning approach. Note that its sleeping interval is not constant as it changes with every scheduling decision.

Using the EDF algorithm, we build for every module a set of dispatch tables $DT[M]$, one for each of its modes, which captures at compile-time the dispatching order of the tasks within a mode. The dispatch table $DT[M]_m$ of a mode m contains a set of entries, each entry consisting of a task and its relative deadline since the beginning of the mode. On single node systems, these deadlines are simply multiples of the task periods; whereas for distributed systems, the bus schedule reduces the available time for the producer task executions by moving their deadlines sooner, at the start time of the corresponding messages. Note that also other scheduling policies (e.g., power-aware) can be used to build the dispatch tables.

For each module M, we have an associated dispatch table index i_M, which points to a task entry in the dispatch table of the current mode $DT[M]_m$ that has a deadline closest to the current logical time. Each time the E-Machine releases a task, it adds the task to the set of active tasks $\mathsf{Tasks}_a[M]$ of that module. It also resets the index i_M when it performs a mode switch in a module or starts a new cycle of a mode. The set of active tasks in a module is always correlated with the dispatch table of the current mode of that module.

We introduce Algorithm 1 for parallel execution of multiple modules using a lightweight EDF scheduling with precompiled dispatch tables. The complexity of the algorithm is $O(\|\mathsf{Modules}[N]\|)$, where $\|\mathsf{Modules}[N]\|$ represents the number of modules on a node N, because we have to perform EDF scheduling only among a single task per module. Note that it is not possible to create a dispatch table for all tasks from all modules because they can switch their modes independently. Also, creating dispatch tables for all combinations of modes from all modules is highly unpractical.

We note with t the current absolute logical time and with t_m the absolute logical time when the mode m started its current period. We use t_m to convert from the deadlines relative to the beginning of the mode to the absolute deadlines required by the EDF algorithm.

The algorithm proceeds through five steps: update the state of the system (value and time), perform the network communication, perform the logical actions according to the LET semantics, schedule the active user tasks, dispatch one task, and sleep until the task completes or a precomputed timeout expires.

Algorithm 1: Task/Bus scheduling

```
// Step 1 − UPDATE STATE
t_begin ← Get_Current_Time()
Save_Task_Context_and_Preempt(τ_old)

// account for elapsed time, where t is the logical time
// and δ is the waiting interval from the previous invocation
if (t − t_begin < δ) // a task completed sooner
  // update δ to reflect elapsed time
  δ ← t − t_begin
end if
Update_Time(t, δ)
Update_Time(EMachineWait, δ)

// Step 2 − COMMUNICATION
NetWait ← NextPacket.time − t mod NetworkPeriod
if (NetWait = 0)
  Invoke(TDLComm) // data exchange required
end if

// Step 3 − LOGICAL ACTIONS
if (EMachineWait = 0)
  // at least one module has to perform logical activities
  Invoke(E−Machine) // returns Time_to_Next_Activity
  EMachineWait ← Time_to_Next_Activity
end if

// Step 4 − USER−TASKS SCHEDULING
δ ← ∞  // retains closest task deadline from all modules
foreach M ∈ Modules // all modules of this node
  // skip past entries
  increment(i_M) while(t − t_m >= DT[M]_m[i_M].dln)
  i ← i_M // seek the first active task
  while(i < ||DT[M]_m||)
    if (δ > DT[M]_m[i].dln − t_m and DT_m[i].τ ∈ Tasks_a[M])
      δ ← DT[M]_m[i].dln − t_m
      τ_new ← DT[M]_m[i].τ
      break
    else
      i ← i + 1
    end if
  end while
end foreach // τ_new has the closest deadline from all modules

// Step 5 − DISPATCHING & WAITING
δ ← minimum(δ, EMachineWait, NetWait)
t_overhead ← Get_Current_Time() − t_begin
δ ← δ − t_overhead // account for elapsed time
Set_Alarm_for_Sleep(δ) // will sleep after dispatching the task
Dispatch_Task(τ_new) // start/resume the execution of task τ_new
```

In the step 1, we first preempt a previously running task and save its current state. The time interval δ represents the sleeping interval of the TDL Scheduler from its previous invocation. We first verify that the Scheduler slept for the required interval or that a task completed sooner and thus the Scheduler was invoked to dispatch another task. We then update the current logical time t and the time interval until the next logical activity from a module, i.e. the moment when the E-Machine has to execute the E-Code of a module.

The bus scheduler tool provides the communication schedule for each node in the form of a table, which lists the packets and their timing. Thus, at step 2, we lookup in this table the time of the following packet and compare it with the current time correlated with the network cycle time. If they match, we invoke the TDLComm layer to send or receive that packet.

Afterward, at step 3, we evaluate the existence of an immediate logical activity to perform. In the case, when any module has such upcoming logical activity (e.g. beginning or end of a task's LET, actuator updates, mode switches), we invoke the E-Machine to execute the E-Code of the appropriate modules.

Reaching the step 4, we proceed to the actual scheduling of the active tasks from all modules. We process all modules and skip the entries in the dispatch table of their currently executing modes that have the deadlines less than the current logical time relative to the beginning of the corresponding mode. We then select as the next dispatch-able task the first task that is active in the dispatch table, as it would have the closest deadline. We cannot increment the dispatch index at this point as it could be that more tasks have the same deadline but have not been released yet (the alternative of keeping track of both released and running tasks requires twice the amount of memory than the simple set of active tasks $\mathsf{Tasks}_a[M]$).

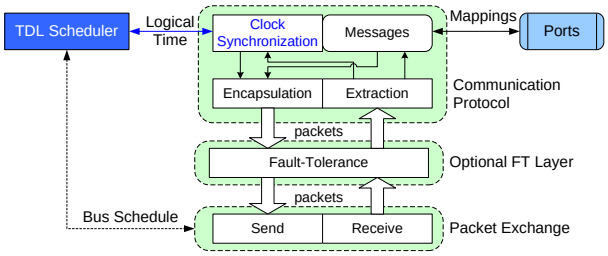


Figure 4: TDLComm services

We update δ with the current closest deadline and the supposed next task to dispatch τ_{new}. After we iterate through all modules, we obtain the smallest sleep/dispatch interval until we have to invoke the scheduler again.

At step 5, we compute the waiting interval as the minimum between the time until the next dispatching, logical, and communication actions. We then subtract from the computed δ the overhead of communication, scheduling, and E-Code execution, to obtain a more accurate estimation of the time allocated to the next dispatching task.

With the information about the next task to dispatch and an estimation of its running time, the Scheduler can prepare its next invocation, dispatch the task and sleep. From the calculation of δ, we can see that the algorithm is invoked repeatedly, but without a fixed period.

We implemented the algorithm on top of existing real-time operating systems such as OSEK [16] or RT-Linux by enforcing the dispatching operations through task/thread priorities. In contrast with classical implementations of EDF on top of fixed-priority operating systems requiring a large number of priority levels [3], our algorithm can work with just three priority levels: high - for the task/thread implementing the TDL runtime (scheduler, E-Machine, TDLComm), medium - for the next/current task to dispatch, and low - all other active tasks. In this way, we still have linear complexity $O(\|\mathsf{Modules}[N]\|)$ because the RTOS scheduler becomes a dispatcher of the first active job (TDL scheduler or dispatched task).

4.3 TDLComm

The TDLComm layer abstracts the physical exchange of information between the nodes of a distributed system. From a logical point of view, using the precompiled scheduling information, the TDLComm layer performs at run-time three steps: the *encapsulation* of port values from service provider modules into packets, the *transmission* of the packets over the communication medium, and the *extraction* of stub-port values from the packets received on the client node (see Figure 4). Each node contains the subset of the global communication schedule relevant for its activities. This subset contains also the failure-management information about the replicated messages how to consolidate them.

For its time-triggered transmission and reception of packets, the TDLComm layer relies on the TDL Scheduler to invoke its functionality at moments of time defined by the communication schedule. On the other hand, it provides the clock-synchronization service on each node of the system, and introduces constraints on the scheduling of the tasks that exchange values over the network. Thus, the close cooperation between the TDLComm layer and the TDL Scheduler is crucial for a successful implementation of the transparent distribution concept of TDL.

The TDLComm layer uses an innovative TDMA protocol [6] that dynamically multiplexes the messages over a static schedule. To handle the independent mode switches of TDL modules, this protocol considers the communication period as the smallest interval where mode switches cannot occur, that is the GCD of the mode-switch periods of all producer modules. The resulting communication period equally divides the period of any mode of a producer module into a fixed number of *phases*. The phases of a mode are mutually exclusive, and any producer module may change its mode only at phase boundaries.

According to this TDMA protocol, any node is allowed to send messages in statically defined slots only. The run-time environment implements a mechanism for global clock synchronization over the network [14]. The data exchange model implemented by the scheduling tool adheres to the Producer-Consumer model. The nodes that generate information (the producers), trigger the sending of information over the network. Contrary to the classical Client-Server model, in the Producer-Consumer model the consumers (the nodes that need the information) do not send any requests to the producers.

Messages, Datagrams, and Packets.

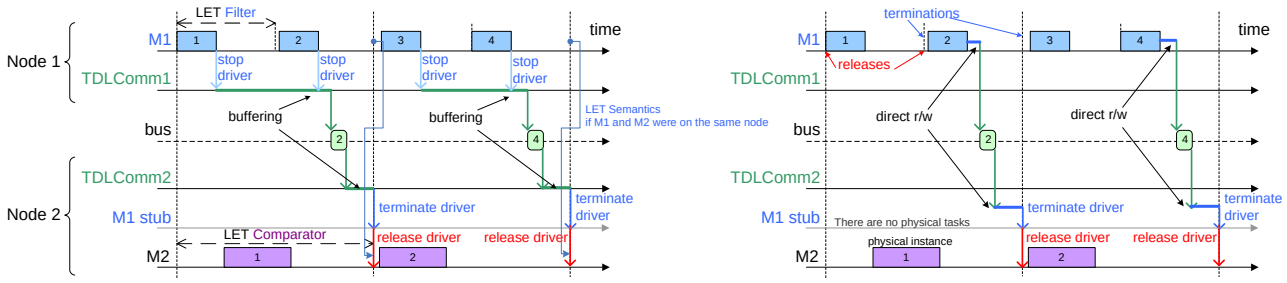
A *message* represents a data exchange between the ports of a pair of TDL entities from two modules located on different nodes in a distributed system. It corresponds to a value exchange operation between sets of ports, discarding the output ports of the producer entity that are not used by a consumer entity. It has a fixed size equal with the sum of the sizes of the producer port types, and two time constraints derived from the availability of the corresponding port values and the latest allowable receive moment (i.e., the end of the LET of the producer task).

The algorithm of the bus scheduler tool identifies the messages from producer tasks per each phase of a mode, and then it associates a message with the phase at the end of the producer's task LET. As the phase and the mode in which a message is produced change at run-time, we also associate to each message a *tag* that encapsulates this information.

A message belongs to a particular task instance; thus, it is not periodic. The number of messages depends on the periods of the producer tasks, the number and period of mode switches, and possibly on the number and periods of the consumer entities [4].

A *datagram* represents a collection of messages exchanged at the same time instant. It contains one or more messages; thus, it refers indirectly to one or more task entities that provide output values from the same or different modules of a node. A datagram has a fixed size equal with the sum of the sizes of each of its message constituents. During the generation of the communication schedule, the scheduling algorithm may grow or shrink the size of a datagram by adding or removing messages; however, once a feasible communication schedule is found and generated, the allocation of messages to datagrams remains fixed.

We refer to a *packet* as the unit of information to send on the communication channel. A packet has a minimum and maximum size derived from the physical properties of the communication channel and the low-level data-exchange protocol. Any packet may contain one datagram only, but more packets may refer to the same datagram. Thus, we can consider a packet as a physical instance of a datagram on the communication channel. Note that in contrast with a datagram, a packet contains actual port-value information, whereas a datagram acts just as a logical container. The order and timing of the packets is fixed within a communication round and expressed statically in the communication schedule. In addition to the actual message data, a packet may contain control information such as tags, timestamps, or other communication-protocol related information.



a) Stop drivers initiated encapsulation

b) TDLComm initiated encapsulation

Figure 5: Information exchange between nodes

A packets has two important attributes: direction and time. The direction specifies the type of operation the TDLComm layer has to perform with the packet. The time attribute for a sending operation reflects the logical time when the TDLComm layer of a node containing a producer module has to send the packet; whereas, for the receiving operation it reflects the logical time when the packet was already received by the network processor of the node containing the client module, and stored in the processor's local buffers or the main memory. In addition, a packet has an index and a datagram reference.

Sending information

We identified two means for capturing the information from a producer module intended for its stubs. The original approach of TDL relies on the stop drivers of the producer tasks. In this case, after the completion of the tasks, their stop-drivers copy the relevant internal port values to the TDLComm layer as presented in Figure 5 (a). On the client-node side, the terminate driver of the stub of the service-provider module performs the port-value copy operation from the TDLComm layer into its visible ports at the end of the LET of the producer task. The release driver of the client task reads the value from the visible output ports of the stub as if the producer task was running on the same node. With this approach, there are multiple drawbacks from the additional meta-information required at the producer module within the task stop-drivers and at the stub module within the task-termination drivers about the TDLComm data-structures, operational mode of the module, and logical time. This requires different glue-code (containing the drivers) for the provider module in the case of single-node versus distributed systems.

We take a better approach that gives the TDLComm layer direct access to the relevant internal ports at the moments defined by the TDL Scheduler and the communication schedule as depicted in Figure 5 (b). Thus, TDLComm reads directly the internal output ports of the producer task, encapsulates them into a packet and sends the packet to the other nodes. On the client-module side, the corresponding TDLComm layer invoked by the TDL Scheduler at the receiving time extracts the port values from the received packet and stores them directly into the internal output ports of the stub module. Hence, the stop-drivers are no longer needed, the stub task-termination drivers are simply identical with its service-provider module, and the glue-code of the service provider module remains the same regardless of the system architecture. Note that under any circumstances the internal output port values are available only to the TDLComm layer before the end of the LET of the corresponding task. All the user tasks in the system can access only the visible output port values, which retain their previous values until the termination event of the task (when the termination driver updates them from the values of internal output ports).

The overhead of copying the data from the main memory to the network processor memory may be negligible, but on slow systems it has to be evaluated and considered as networking overhead when performing the time-safety checking of the distributed system and at runtime within the TDL Scheduler.

The transmission of the computation results of a producer module to its stubs requires a data encapsulation phase. For this purpose, we introduce the Algorithm 2 that computes first the phase of the current mode of a module and creates a corresponding tag from the module, mode, and phase. It then starts building a new packet by matching the set of possible messages with the previously identified tag. As the tag captures the dynamic state of the module and there may be more than one message with the same tag, it packs the tag and the content of the corresponding ports into the packet. For the case where multiple messages from different modules and phases are merged into a larger frame, we have to repeat the algorithm for each module.

Algorithm 2: Dynamic packet encapsulation

// t is the current time
// $m \in$ Modes$[M]$ is the current mode of M
// t_m is the time when the current mode period started
// d is the datagram corresponding to the $NextPacket$ to send

$NextPacket$.data $\leftarrow \emptyset$ *// first construct a new packet*
foreach $M \in$ Modules$[N]$ *// all modules on the node N*
  $\phi_m = (t - t_m)$ **mod** $NetworkPeriod$ *// compute phase*
  $tag \leftarrow (M, m, \phi_m)$ *// tuple expressing module dynamics*
  Store_Tag($NextPacket$.tags, tag) *// add tag*
  foreach $msg \in d$ *// process all messages*
    if (msg.tag = tag)
      $v \leftarrow$ InternalPortValue(msg.PortReference)
      $NextPacket$.data $\leftarrow$ $NextPacket$.data $\cup \{v\}$
    end if
  end foreach
end foreach
// Send $NextPacket$ and advance its index

Receiving information

The extraction of port values from the packets relies on the tag information from a received packet. Hence, we introduce the Algorithm 3 that first identifies the tag of the received packet and then decodes the state of the producer module, its mode, and current phase.

When the node containing a client module and a stub of the service provider module receives the packet, the producer module may have changed into a new mode at the beginning of the communication cycle. In this case, we change the mode of the stub module and update its mode start time t'_m and phase ϕ'_m. We proceed to the extraction of the packet's content and store the received values

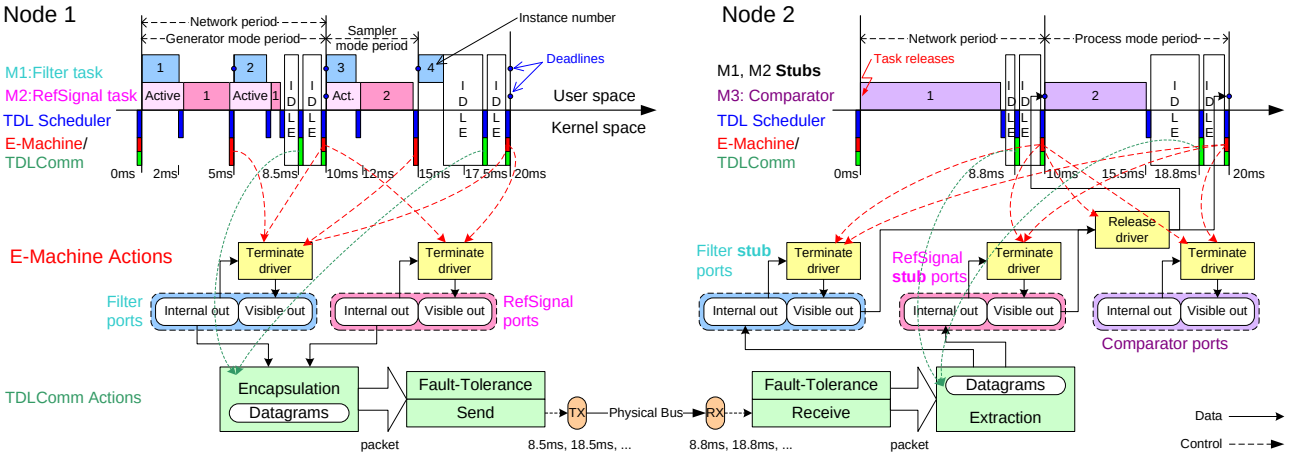


Figure 6: Data flow between modules through TDLComm

into the corresponding internal ports of the stub module.

The protocol implemented by TDLComm brings flexibility in static communication scheduling, by allowing dynamic mapping of messages over the same frames. Nevertheless, it still provides deterministic communication patterns and a solid foundation for transparent distribution.

Algorithm 3: Stub synchronization

// t is the current time
// M_{stub} is a stub module for M
// $m \in$ Modes$[M]$ is the current mode of M
// $m' \in$ Modes$[M_{stub}]$ is the current mode of M_{stub}
// t_m is the time when the current mode period started
// t'_m is the time when the current stub mode period started
// d is the datagram corresponding to the $NextPacket$ to receive

foreach $tag \in NextPacket$.tags
  $(M, m, \phi_m) \leftarrow tag$ *// decode tag*
  $\phi'_m = (t - t'_m)$ **mod** $NetworkPeriod$ *// compute stub phase*
  // synchronize timing of producer and stub
  if $(m' \neq m)$ *// producer has changed mode during this phase*
    $t'_m \leftarrow \lfloor t/NetworkPeriod \rfloor \cdot NetworkPeriod$
    $m' \leftarrow m$ *// update mode of stub and its start time*
    $\phi'_m \leftarrow \phi_m$ *// update mode phase*
  end if
  foreach $msg \in d$ *// process all messages*
    if (msg.tag = tag)
      $v \leftarrow$ FetchPortValue($NextPacket$.data)
      SetInternalValue(msg.PortReference, v)
    end if
  end foreach
end foreach *// internal ports & mode of stub are synchronized*

5. EVALUATION

We take as example the application containing the three modules from Figure 1. In this simple application, each module has only one operating mode containing one task. The first module has a sensor connected to the input of the Filter task, which is invoked with a period of 5ms. The RefSignal task of the second module has a period of 10ms and only state ports to retain its state between consecutive invocations. The third module has no sensor inputs but imports the other two modules to access the outputs of their tasks. The Comparator task is invoked every 10ms to perform some computations and provide their result to the environment via an actuator.

From a logical point of view, the three module run in parallel regardless of the underlying platform and its capabilities. The LET semantics dictate that the first output of the RefSignal goes to the second invocation of the Comparator (in the second cycle of the Process mode). Similarly, the output of the second instance of the Filter reaches the second instance of the Comparator, then the fourth output of Filter reaches the third output of the Comparator and so on. This is the so-called unit-delay behavior (see Figure 1).

We consider now the case of a distributed system, where the first two modules are located on the node 1 and the third module on the node 2. Figure 6 presents on the upper part the CPU scheduling, and in the lower part the mechanism for transparent distribution (without the clock synchronization service).

On the first node, the TDL Scheduler invokes the TDLComm layer to perform the clock synchronization of the two nodes, and then it invokes the E-Machine to initialize the ports of the tasks and release the tasks of both modules. The deadline of the Filter task is sooner than the deadline of the RefSignal task; therefore, the Scheduler dispatches the Filter task and keeps the RefSignal task into the active state. The run-time environment of node 2 performs the same steps and dispatches the Comparator task.

The two nodes run in parallel, each executing one task until the moment of 2ms, when the Filter task completes its execution. The scheduler of the first node is invoked and dispatches the RefSignal task, whereas the second node continues its computations undisturbed. At 5ms, the scheduler of the first node preempts the RefSignal task to invoke the E-Machine that first executes the termination drivers of the Filter task and then releases a new instance of this task. At this point both tasks on the first node have the same deadline of 10ms. We assume that the scheduler dispatches the Filter task that completes its computations sooner than its first instance (around 6.5ms). The scheduler dispatches the remaining RefSignal task that shortly completes its computations.

Meanwhile, the second node completed the execution of the Comparator task around 7ms and idles. We assume that the communication schedule specifies the sending of a packet from the first node to the second at 8.5ms. As there is no other active task, the node 1 remains idle until this moment. The scheduler on node 1 invokes its TDLComm layer to capture the internal output port values of the two tasks. The results are encapsulated and transmitted over the network in 300us, reaching the second node at the time of 8.8ms. The Scheduler of the first node returns to the idle state, whereas the Scheduler of the second node wakes up and executes its TDLComm

layer to extract the data from the received packet and update the internal output ports of the two stub modules. Afterward, it reenters the idle state until the beginning of a new mode cycle.

When the logical time reaches 10ms, on both nodes, the TDL Scheduler invokes the TDLComm layer for clock synchronization and then invokes the E-Machine. The virtual machine executes on the first node the termination drivers of both tasks, thus making their outputs available to the environment. Afterward, it begins a new mode cycle in each module and releases new instances of the Filter and RefSignal tasks. On the second node, the E-Machine executes the stub termination drivers and makes the computation results of the two modules from the first node available to the local environment. Thus, the release drivers of the Comparator task read the correct values as if all modules were executed on the second node. The new cycle of the third module begins with another release and dispatch of the Comparator task.

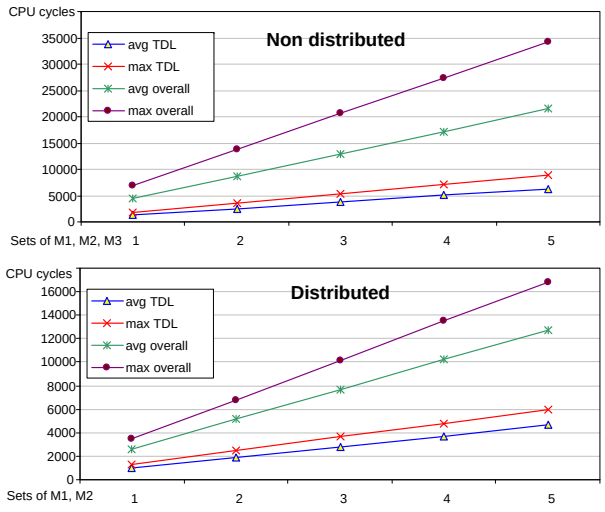


Figure 7: CPU utilization

In the second network cycle, assuming that the sensor values require less filtering, and the comparison takes less time, all tasks from all modules complete their execution sooner than in the first network cycle. Nevertheless, the behavior is still the same and the communication pattern remains unchanged. It also matches the behavior of the modules on a faster single-node system, thus, illustrating the benefits of our approach for distribution.

In Figure 7, we measured using Avrora [18] the number of CPU cycles required for the execution of the TDL runtime environment and the overall utilization. The system under evaluation is an Atmel AVR microcontroller with TDL running on the bare hardware (microkernel design). We start with one set of the three modules on a single node. We continue by adding sets of (M1, M2, M3) up to 15 modules per node and observe that the CPU utilization grows linearly with the number of modules to execute. In correlation with previous research for Giotto [10], our measurements indicate aprox. 1.5% CPU utilization on a similar platform, proving that executing parallel modules increases the development flexibility at no performance penalty. In the second graph, we distribute the module M3 on a second node connected via CAN (with our TDMA protocol on top), and use sets of (M1, M2) for testing. The CPU utilization on the first node drops significantly and the system maintains its observable behavior unchanged. Thus, we can make better use of the available resources of the target platform in a distributed system.

6. RELATED WORK

Giotto [7], the precursor of TDL, is primarily an abstract mathematical concept and there exist only simple prototype implementations, which show some of the potential of LET: static time-safety checks and platform-independent embedded code (E-Code) executed by virtual machines. Its main focus are the single-node systems with limited support for task-level distribution. The developer has to annotate the Giotto source code with network specific parameters such as the hostname and ports; thus, mixing the platform-independent and platform-dependent code. The actual implementation of the communication has to be coded manually with so-called scheduling-code instructions. In other words, no tools automatically generate the message schedules for the bus communication.

Our approach is more realistic and takes into account the complexity of the real-world applications, where entire parts of an application are distributed - modules. The key elements for the transparent distribution in this case are the stub module concept and the TDLComm layer that decouples the distribution aspects from the rest of the run-time environment and frees the developer from the burden of developing individual modules towards a distributed solution. The integrated communication and task scheduling make our approach a feasible solution for distribution, with support for parallel module execution with data dependencies between modules regardless of their placement on the network.

TTP [19] protocol and its related tools provided by the TTTech company approach the problem of building real-time distributed system with proprietary, expensive, high-confidence hardware that features membership services, time-triggered transmission of messages and distributed clock synchronization. In addition, in its current state the developer has to consider in advance the platform topology, the number and type of messages exchanged between the nodes. The schedule itself is then generated with the TTPplan tool, but it cannot support multiple application modes. The possibility of independent mode switches on each node, by arbitrary modules is simply out of the modeling possibility of the current tools.

Our approach abstracts from the communication infrastructure through transparent distribution, which shields the developer from ever defining each message individually or even designing for a particular distributed platform. Modules can be developed independently and later integrated without affecting their timing behavior.

FlexRay [1, 2] is an emerging high-speed fault-tolerant protocol for control applications. A group of companies including BMW, Volkswagen, DaimlerChrysler, Bosch, Philips, and Freescale is actively developing it, with its specifications in the final phase. Apart from time-triggered operations, its focus is flexibility. Thus, it can operate in both active star and passive bus topologies, and can accommodate both static and dynamic parts in its communication rounds. Node may be connected through one or two communication channels (for redundancy), and may transmit the same data on both channels, or different data on both channels, in a given current time slot. However, given its age it is mostly implemented using proprietary ASIC chips and its adoption in the automotive industry has still to gain momentum. The development tools for it have also limited capabilities, comparable with the ones for the TTP protocol. Thus, the potential of the protocol remains still hindered by inappropriate software.

The presented approach is currently under evaluation on this platform and preliminary results show that it is possible to gain the benefits of using this protocol and accompanying hardware in addition to the transparent distribution of the TDL components. A description of the model-driven development process for a FlexRay platform using TDL is available in [15].

7. CONCLUSIONS

The presented development methodology goes significantly beyond Giotto and relives the developer from the burden of explicitly designing the application for a distributed system and allows the integration of an application out of individually developed modules. It also reduces the dependence on proprietary networking technologies for real-time systems, while still benefiting from their capabilities when such networking options are available. Future research, implementation, and testing efforts are required to show the scalability of transparent distribution in complex scenarios. Remaining challenges are better heuristics for generating communication and task execution schedules to account for power/bandwidth usage or other dynamic environments, and strategies for avoiding the re-generation of schedules when components are added or modified.

8. REFERENCES

[1] J. Berwanger, C. Ebner, and et al. FlexRay - The Communication System for Advanced Automotive Control Systems. In *SAE World Congress*, Detroit, MI, Apr. 2001. Society of Automotive Engineers Press. 2001-01-0676.

[2] F. Bogenberger, B. Müller, and T. Führer. Protocol overview. *FlexRay International Workshop: The Communication System for Advanced Automotive Control Applications*, Apr. 2002.

[3] G. C. Buttazzo. Rate monotonic vs. EDF: Judgement day. *Embedded Systems*, pages 67–83, Sept. 2003.

[4] E. Farcas. *Scheduling Multi-Mode Real-Time Distributed Components*. PhD thesis, Department of Computer Science, Univ. of Salzburg, Austria, 2006.

[5] E. Farcas, C. Farcas, W. Pree, and J. Templ. Transparent distribution of real-time components based on logical execution time. In *Proc. of LCTES*. ACM Press, 2005.

[6] E. Farcas, W. Pree, and J. Templ. Bus Scheduling for TDL Components. *LNCS - Dagstuhl Conference on Architecting Systems with Trustworthy Components*, May 2006.

[7] T. Henzinger, B. Horowitz, and C. Kirsch. Giotto: A time-triggered language for embedded programming. In *Proc. of EMSOFT*, LNCS 2211, pages 166–184. Springer, 2001.

[8] T. Henzinger and C. Kirsch. The Embedded Machine: Predictable, portable real-time code. In *Proc. of the PLDI*, pages 315–326. ACM Press, 2002.

[9] C. Kirsch. Principles of real-time programming. In *Proc. International Workshop on Embedded Software (EMSOFT)*, LNCS 2491, pages 61–75. Springer, 2002.

[10] C. Kirsch, M. Sanvido, and T. A. Henzinger. A programmable microkernel for real-time systems. *Proc. of VEE*, 2005.

[11] H. Kopetz. The Time-Triggered Model of Computation. In *Proc. of the IEEE Real-Time Systems Symposium*, 1998.

[12] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in hard real-time environments. *Journal of the ACM*, 20(1):46–61, Jan. 1973.

[13] J. W. S. Liu. *Real-time Systems*. Prentice Hall, 2000.

[14] G. Menkhaus, M. Holzmann, and S. Fischmeister. Time-triggered Communication for Distributed Control Applications in a Timed Computation Model. In *23rd Intl. Digital Avionics Systems Conference*. IEEE Press, 2004.

[15] A. Naderlinger, J. Pletzer, W. Pree, and J. Templ. Model-Driven Development of FlexRay-Based Systems with the Timing Definition Language. Mineapolis, May 2007.

[16] OSEK Group. *OSEK/VDX Operating System v2.2.3, and OSEKtime - Time-Triggered OS v1.0*, 2005.

[17] J. Templ. TDL Specification and Report. Technical report, University of Salzburg, Austria, http://www.softwareresearch.net/site/publications/C059.pdf, 2004.

[18] B. L. Titzer, D. K. Lee, and J. Palsberg. *Avrora: scalable sensor network simulation with precise timing*. IEEE Press, Los Angeles, CA, 2005.

[19] C. G. TTTech. *TTP/C Protocol Specification, Version 1.0*. Wien, Austria, June 2002.

[20] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Muller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. *The Worst-Case Execution Time Problem – Overview of Methods and Survey of Tools*. Tech. Report. Mälardalen University, Sweden, Mar. 2007.

An OSEK/VDX Implementation of Synchronous Reactive Semantics Preserving Communication Protocols*

Guoqiang Wang[1] geraldw@eecs.berkeley.edu
Marco Di Natale[2] marco.dinatale@gm.com
Alberto Sangiovanni-Vincentelli[1] alberto@eecs.berkeley.edu
[1] University of California at Berkeley, CA, USA
[2] General Motors R&D, Warren, MI, USA

Abstract

Synchronous Reactive (SR) models are increasingly used in model-based design flows for the development of embedded control applications. In this paper, we present semantics preserving implementations of SR communication for multi-rate tasks. The implemented protocols define the assignment of indexes of shared buffers to writer and reader tasks at activation time, rather than at execution time, and therefore require kernel-level support. We provide the details of two constant-time solutions, developed in the C language, and using the automotive OSEK OS standard for portability and reusability. Run-time complexity and memory requirements are discussed for the two protocol implementations and tradeoffs are analyzed.

1 Introduction

Model-based development of embedded real-time software aims at improving quality by enabling design-time verification and simulation and fostering reuse. Synchronous reactive (SR) models have been traditionally used in the design of hardware logic and more recently for modeling control-dominated embedded applications. SR zero-time semantics is very popular because of the availability of tools for simulation and formal verification of the system properties. When implementing a high-level model into code, it is important to preserve its semantics, so to retain the results of the simulation and verification stages. However, this requires that *the run-time behavior when the functions are implemented by a program with a finite execution time and possibly subject to preemption is provably equivalent to the SR model with zero execution time and no preemption.* In general, defining such a provably correct implementation is non-trivial.

*This work was supported by the MARCO/DARPA Gigascale Systems Research Center (http://www.gigascale.org). Their support is gratefully acknowledged.

2 SR Model and Definitions

In our synchronous reactive model, a system is a network of tasks communicating using ports. All tasks in the model react at the same time (synchronously) and periodically, at the *base rate* of the system. A reaction may be the computation of a function that updates the values of the output ports as function of the task state and possibly of the values at the input ports. The function also updates the state as a function of the state and the inputs. Alternatively, a reaction may be a *stutter* meaning that the output values and the state are left unchanged. According to the SR model, the reaction occurs in *zero time*, meaning that the outputs and the state depend instantaneously on the input values. The implication of the zero execution time semantics is that the input values *must be determined* at the time the task is activated for a non stutter reaction. According to the possible existence of ports of either input or output type, a task can consequently be a reader, a writer, or both. By using a proper definition of stuttering reactions, it is possible to define tasks that update their outputs and state only at multiples of the base period. In the following, we will assume that these tasks perform their reaction at such multiple periods.

Each SR task is then implemented by a run-time task, executed under the control of the operating system. The task execution time is finite and it can be preempted according to its priority. A run-time task can implement one or more SR tasks. Each implementation task τ_i is characterized by a set of parameters: priority π_i, period T_i, computation time C_i, worst case response time R_i, and relative deadline d_i. Tasks are scheduled by priority with preemption. Schedulability of tasks requires that $R_i \leq d_i$.

Our implementation assumes that $R_i \leq T_i$, which implies that only one active instance for each task exists at any time. Let $a_i(j)$ be the activation time of the j^{th} instance of τ_i. Under the SR semantics, given that the execution time is zero, the activation time $a_i(j)$ captures also the start time and the finish time of the

same instance of τ_i.

Let w and r_i denote a writer and one of its readers. Let $o_w(j)$ be the output associated with the j^{th} instance of w and $i_{r_i}(j)$ be the input associated with the j^{th} instance of r_i. We define $\zeta_i(t)$ to be the number of times that task i has occurred up to time t, i.e.

$$\zeta_i(t) = \sup\{m | a_i(m) \leq t\},$$

where the sup of an empty set is defined to be zero.

In the general case, communication is between one writer and its NR = NHPR + NLPR readers, among which NHPR have higher priority, and NLPR have lower priority than the writer. Let delay[i] $= \{0, 1\}$ be the communication link delay for reader i. The SR communication semantics can be formulated as follows:

$$i_{r_i}(j) = o_w(k),$$

where $k = \max\{0, \zeta_w(a_i(j)) - delay[i]\}$.

The top of Figure 1 illustrates the execution of a pair of tasks communicating with the SR zero-time semantics. The horizontal axis represents time. The vertical arrows capture the time instants when the tasks are activated and compute their output from the input values. Please note that, in the middle of the figure, it is $i_{r_i}(j) = o_w(k)$ during simulation. The bottom of Figure 1 shows the possible problems with data transfers in a multi-task implementation when buffers are addressed at execution time. A fast writer, implemented by a high priority task, communicates with a slow reader. The writer finishes its execution producing output $o_w(k)$ and the reader is executed right after. If the reader performs its read operation before the preemption by the next writer instance, then $i_{r_i}(j) = o_w(k)$. Otherwise, it is preempted and a new instance of the writer produces $o_w(k+1)$. In case the read operation had not been performed before, the task reads $o_w(k+1)$, in general different from the value $o_w(k)$. Even worse, in case the signal value is not read atomically, there is a finite probability that $w(k)$ preempts the reader task r_i while a read is in progress, resulting in an inconsistent value and a data integrity problem.

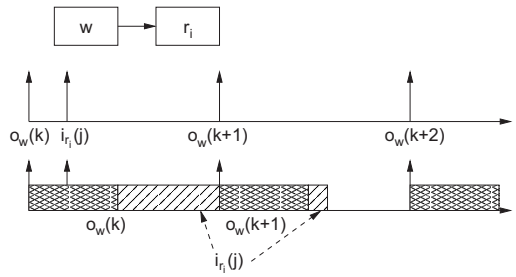


Figure 1. How Preemption Changes the Values Read by a Reader

We further introduce the offset, denoted by $o_{wi}(k)$, between the k^{th} instance of the reader $r_i(k)$ and the writer instance $w(j)$ that produced the data consumed by the reader, i.e. $o_{wi}(k) = a_i(k) - a_w(j)$, where $j = \sup\{m | a_w(m) \leq a_i(k)\}$. The largest value of o_{wi}, O_{wi}, is always smaller than the period of the writer, i.e. $O_{wi} < T_w$.

Review of previous work There are two options for a correct implementation of an SR multirate model on single-processor execution platforms. In a single task implementation, all the SR tasks are implemented by a single run-time task (or *executive*), running at the base rate of the system. Such an implementation is easier to construct, but often characterized by poor resource utilization.

A multi-task implementation typically uses one task for each execution rate, and possibly more. Multi-task implementations allow for a much better schedulability, but because of the possible preemption, communication may have integrity or non-determinism problems and the implementation raises issues with respect to the preservation of the zero-time execution behavior.

Any (real-time) data communication between concurrent tasks that cannot be made atomic at the hardware level must be implemented using a concurrency control mechanism. Wait-free schemes [1][2] can be used to protect a writer and its readers against concurrent access to the communication data by replicating the communication buffers and possibly by leveraging knowledge of access times and scheduling constraints such as task priorities and periods.

Wait-free schemes are the preferred choice for the implementation of semantics-preserving communication protocols due to their simplicity and efficiency. A one-to-one communication mechanism that preserves the SR semantics has been presented in [3]. A two-place buffer, two buffer indexes and a reader execution flag are required. In the case of single processor systems, given that the code that updates the index variables is executed inside the kernel, at task activation time, there is no need for a Compare-And-Swap (CAS, or another equivalent) instruction, or any other mechanism that ensures atomicity when swapping buffer pointers or comparing state variables.

In the general case of multiple reader tasks, wait-free mechanisms can be sized and constructed by leveraging two properties of the relationship between the writer and its readers. The first method consists in computing an *upper bound for the maximum number of buffers that can be used at any given time by reader tasks*. In [4] the bound is defined for the case of communication links with a unit delay, under the assumption that each task instance terminates before its next activation event. The protocol is called DBP (Dynamic Buffering Protocol). When unit delays are allowed on links, NLPR + 2 buffers are demonstrably sufficient, where one buffer guarantees that the writer can safely update the

latest data and one is for higher priority reader tasks that need access to the previously written data.

The other method provides buffer sizing and access procedures by guaranteeing that writer and reader tasks never access the same data item at the same time. The size of the buffer can be computed by upper bounding the number of times the writer can produce new values while a given data item is considered valid by at least one reader. This concept has been first introduced (together with a lock-free protocol implementation) in [5] and [6] (under the name of Non-Blocking Write, or NBW), assuming as the validity time of the data the worst case execution time of a reader. The temporal concurrency control concept is also used in [7] for buffer sizing while preserving the SR semantics.

In an SR semantics-preserving implementation, we need to ensure that the reader accesses the value produced by the correct instance of a writer task. In particular, the buffer slot that contains the item produced by the writer has to be defined at the writer's activation time. Similarly, the buffer item read by a reader is defined at the reader's activation time.

Later, at execution time, the writer and the reader will use the buffer positions defined at their activation time. Both writer and reader tasks, however, are not guaranteed to start their execution at their release time because of the scheduling delays. Therefore, in general, the selection of the data buffer entry that will be written into or read from must be delegated to the operating system (or to a hook procedure that is guaranteed to be executed at the task activation time). Of course, there may be cases in which the writer produces multiple outputs before the reader completes its execution. In this case, the implementation must necessarily consist of an array of buffer entries in which pointers (indexes) are assigned to the writers and the readers to find the right entry.

We have implemented different versions of SR communication protocols under the OSEK/VDX in [8]. OSEK/VDX is a standard for automotive applications and includes operating system (OS), communication (Com), network management (NM), and debugging (ORTI). In addition, we analyze the tradeoffs between different versions of the protocols in terms of time, space, and implementation complexity. Due to space limitation, we only present two implementations of SR communication protocols in this paper.

3 Buffer Sizing Mechanisms

Any communication scheme consists of two parts: a buffer sizing mechanism and a buffer indexing procedure. In this section, we first present two mechanisms used to size communication buffers and in Section 4 we define the corresponding procedures.

The first mechanism is the DBP protocol [4], based on the active number of reader instances. In DBP, the writer writes data into a buffer (message array) in a *spatially-out-of-order* manner. Given the possibility of unit delay links, we need to keep one copy of the current and the previous buffer indices. Some readers may share the same buffer slot, but in the worst case, all of them may require unique entries. If NLPR is the number of the readers with a lower priority than the writer, then the size of the buffer is $\mathtt{NB} = \mathtt{NLPR} + 2$,where NLPR slots are reserved for lower-priority readers, one entry stores the writer output with a unit delay, and another entry is for the writer to write into a new data item. All higher-priority readers share the same copy of the buffer item written by the previous writer instance.

The other mechanism used for communication buffer sizing allows the writer to write data into a buffer in a *spatially sequential order*. This mechanism is termed Temporal Concurrency Control (TCC), because it relies on the temporal properties of the tasks.

Assume that some writer instance k happens at time $\mathtt{a_w(k)}$ and it updates a buffer position of index n (Figure 2). The item in position n is used by the readers with no delay that are activated during the interval $[\mathtt{a_w(k)}, \mathtt{a_w(k+1)})$ and by the unit delay readers activated in $[\mathtt{a_w(k+1)}, \mathtt{a_w(k+2)})$.

The buffer slot with index n must remain valid until any of those reader instances has finished its execution. Future instances of the writer use buffer slots with indexes $\mathtt{n}+1, \mathtt{n}+2$, and so on, until, eventually, the index wraps around the circular buffer and goes back to position $\mathtt{n}-1$. A correct buffer sizing must ensure that all the reader instances that used the item with index n finished using it, when some future writer instance goes back to position n and overwrites it. The

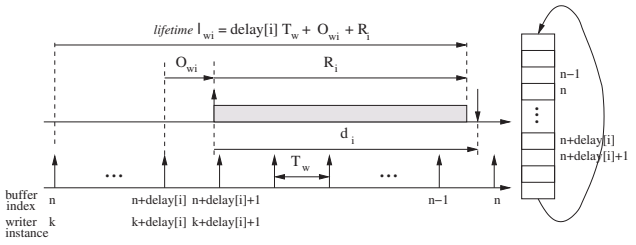


Figure 2. Buffering Sizing Mechanism Based on Spatially-In-Order Writes

maximum lifetime of the data produced by the writer for the reader $\mathtt{r_i}$, denoted by $\mathtt{l_i}$ is:

$$\mathtt{l_i} = \mathtt{delay[i]} \times \mathtt{T_w} + \mathtt{O_{wi}} + \mathtt{R_i}.$$

If NR is the number of readers of a generic writer, the buffer size required by the writer is:

$$\mathtt{NB} = \max_{1 \le i \le \mathtt{NR}} \left\lceil \frac{\mathtt{l_i}}{\mathtt{T_w}} \right\rceil. \qquad (1)$$

4 Protocol Implementation

In a fixed-priority scheduled multi-task implementation, the link delay must be equal to one for readers with a priority higher than the writer, while for readers with a lower priority, it can be either zero or one. A pair of variables (`cur`, `prev`) refers to the indexes of the latest written item and the previous one.

For convenience, Table 1 shows the notations. Variables `cur`, `prev`, `NLPR`, and `WrtInit` defined for each output port in the system. All readers and writers share array `Buf[SysNB]` for communication. The total buffer size required by the system, `SysNB`, is simply the sum of buffer sizes of all writers and $\texttt{NB}_{\texttt{w}_\texttt{o}}$ is computed as specified by the DBP or TCCP protocols, respectively.

$$\texttt{SysNB} = \sum_{1 \le o \le \texttt{SysNOP}} \texttt{NB}_{\texttt{w}_\texttt{o}} \qquad (2)$$

NR	number of readers	NT	number of tasks
delay	link delay	IsHPR	relative priority
WrtInit	initial output value	pri	task priority
Buf[]	shared comm. buffer	SysNB	total buffer size
SysNIP	number of input ports	SysNOP	number of output ports
cur	buffer slot with latest data		
prev	buffer slot with immediate previous data		
NLPR	number of lower-priority readers		
Read[i]	buffer slot currently used by reader i		

Table 1. Notations Used to Describe a System

4.1 The Dynamic Buffering Protocol

The high-level pseudo-code of the DBP protocol is shown in Figure 3, as defined in [4].

Data Structures	
`char cur, prev;`	`char Read[NLPR];`
`message Buf[NB];`	`char HPR[NHPR];`
Writer	
activation time	*execution time*
`prev = cur;`	···
`cur = FindFree();`	`Buf[cur] = ···`
	···
`FindFree() {`	
`  return j∈[1,NLPR+2]if prev≠j∧∀i∈[1,NLPR] Read[i]≠j;`	
`}`	
Lower Priority Reader	
activation time	*execution time*
`if (delay[i])`	···
`   Read[i] = prev;`	`··· = Buf[Read[i]];`
`else`	···
`   Read[i] = cur;`	`Read[i]=FREE;`
Higher priority Reader	
activation time	*execution time*
`HPR[i] = prev;`	`··· = Buf[HPR[i]];`

Figure 3. Code for Writer/Readers in [4]

There are different ways to implement the `FindFree()` procedure that is used to find a free buffer slot by the writer at its activation time. We present here an implementation with a constant execution time.

`FindFree()` must be executed at the activation time of the writer, by the kernel, or at the highest priority level. A long execution time is therefore highly undesirable, and we are interested in trading off some memory space for an implementation with a constant execution time using a use free list as shown in Figure 4.

An array implementation of the list contains two fields: the use count (`use`) and the next free slot index (`NextFree`). The start of the free list is indicated by `FreeHd` and the free list is terminated by a value of -1. Since the values of the `use` fields along the free list are all zeros, to save memory, the two columns can be compacted into one, containing the value of the next free slot index or the use count. The free entry can be obtained by getting the `FreeHd` value, and list updates can be performed in constant time. Figure 5 shows the data structures of the implementation. The array `TaskL` has an entry for each task specifying its priority and a reference to its input and output port information. The input port descriptor contains the communication source port, the link delay, and the relative priority of the reader with respect to its writer. Similarly, the output port descriptor specifies the properties of each output port, including the reference to the list of the free blocks, `FreeHd`, the `cur` and `prev` variables and `BufHd` and `NB`, which specify a contiguous segment in the buffer array `Buf[]` for the output ports.

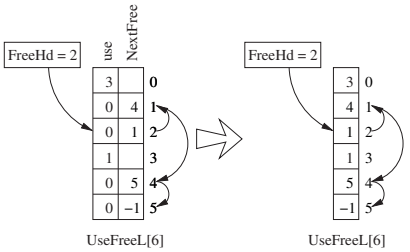


Figure 4. A Use Free List Data Structure

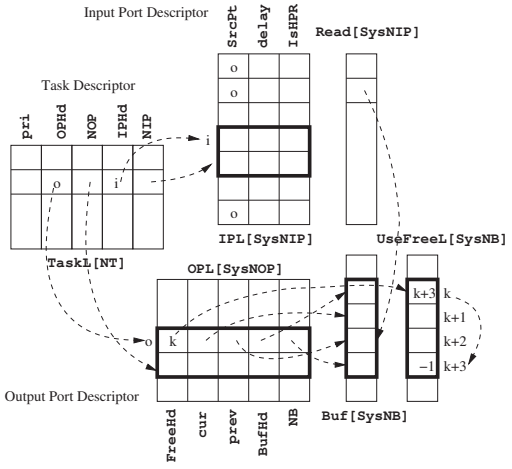


Figure 5. Data Structure for DBP

All the port descriptors that refer to the same task are stored in contiguous locations, as specified by (`OPHd, NOP`) and (`IPHd, NIP`). Figure 6 shows the data structure declarations and their initialization. As shown in

Data Structure

```
struct TaskEntry {
  char pri;
  char OPHd;
  char NOP;
  char IPHd;
  char NIP;
} TaskL[NT];

struct IPEntry
  char SrcPt
  char delay
  char IsHPR
} IPL[SysNIP];

struct OPEntry {
  char FreeHd;
  char cur;
  char prev;
  char BufHd;
  char NB;
} OPL[SysNOP];

char Read[SysNIP];
message Buf[SysNB];

char UseFreeL[SysNB];
message WrtInit[SysNOP];
```

Initialization

```
OPL[0].BufHd = 0; /* var and buffer init */
OPL[0].cur = OPL[0].prev = OPL[0].BufHd;
Buf[OPL[0].BufHd] = WrtInit[0];
for (i = 1; i < SysNOP; i++) {
  OPL[i].BufHd = OPL[i-1].BufHd + OPL[i-1].NB;
  OPL[i].cur = OPL[i].prev = OPL[i].BufHd;
  Buf[OPL[i].BufHd] = WrtInit[i];
}
for (i = 0; i < SysNOP; i++) { /* init of free list*/
  UseFreeL[OPL[i].BufHd] = 1; /* not free */
  OPL[i].FreeHd = OPL[i].BufHd + 1;
  for (j = 2; j < (OPL[i].NB-1); j++) {
    k = j + OPL[i].BufHd;
    UseFreeL[k] = k + 1;
  }
  UseFreeL[OPL[i].NB-1+OPL[i].BufHd] = -1;
}
```

Figure 6. DS Initialization for DBP

```
/* activation time */
/* each writer i */
UseDec(i, OPL[i].prev);
OPL[i].prev=OPL[i].cur;
OPL[i].cur=FindFree(i);
UseFreeL[OPL[i].cur]=1;

/* each reader i */
j = IPL[i].SrcPt;
if (IPL[i].delay)
  Read[i]=OPL[j].prev;
else
  Read[i] = OPL[j].cur;
if (IPL[i].IsHPR == 0)
  UseFreeL[Read[i]]++;
```

```
/* execution time */
...
/* each writer k */
Buf[OPL[k].cur] = ...
...
/* each reader k */
... = Buf[Read[k]];
...
```

```
/*termination time:CS*/
if (IPL[k].IsHPR==0) {
  t1 = Read[k];
  t2 = IPL[k].SrcPt;
  UseDecM(t2,t1);
}
```

```
Def of UseDec(char i,char j)
void UseDec(char i, char j){
  UseFreeL[j]--;
  if(UseFreeL[j] == 0) {
    UseFreeL[j]=OPL[i].FreeHd;
    OPL[i].FreeHd = j;
} }
```

```
Def of FindFree(char i)
char FindFree(char i) {
  t = OPL[i].FreeHd;
  OPL[i].FreeHd=UseFreeL[t];
  return t;
} /* O(1) */
```

Figure 7. Application Tasks for DBP

Figure 7, the communication protocol executes at two levels: on task activation at the kernel level, and during execution by the application task code. At activation time, if the task is a writer, for all its output ports, the `use` count of the buffer item referred by `prev` needs to be decremented and if the new count drops to zero, this buffer slot is freed by updating the corresponding `FreeHd`. Also, the `cur` index and its corresponding `use` count are updated. For lower-priority readers, their corresponding use counts need to be incremented.

When a task terminates, the task decrements the `use` count of the buffer slot for all the input ports that receive data from a high priority writer. Moreover, when the `use` count of a buffer item becomes zero, the item is returned to the list and the corresponding writer's `FreeHd` may need to be updated using operations that are not atomic. Since `FreeHd` and `UseFreeL[]` are shared by each writer with its lower-priority readers, atomicity of the critical section at termination time must be guaranteed by any correct implementation. The constant time `FindFree()` shown in Figure 7 takes the writer's index as input and returns the current `FreeHd` after assigning the index of the second entry on the free list as the new `FreeHd`. Under the DBP buffer sizing, it is guaranteed that `FreeHd` always refers to a valid entry. The memory requirements of the implementation is shown in Table 2.

variable	char	message
count	5×(NT+SysOP)+4×SysNIP+SysNB	SysNB+SysNOP

Table 2. DBP Memory Requirement

4.2 Temporal Concurrency Control Protocol

The mechanism based on spatially-in-order writes is used for the buffer sizing in the Temporal Concurrency Control Protocol (TCCP). Figure 8 shows the data structures for TCCP. The size of the shared buffer and the buffer size of each writer are computed using Equations 2 and 1. Similar to DBP, there are three descriptors to characterize each task, with its input and output ports. The data structure declaration and the initialization code is shown in Figure 9. Each writer is assigned a continuous segment of `Buf[]`, identified by the pair (`BufHd, NB`), as shown in Figure 8. As shown in the buffer indexing procedure in Figure 10, all the

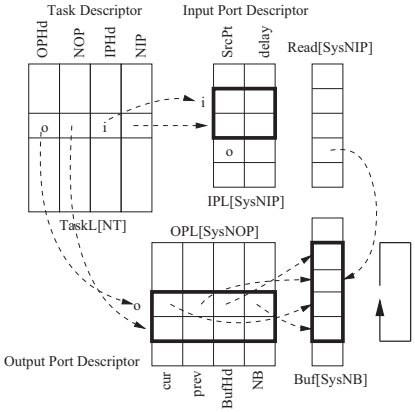


Figure 8. Data Structure for TCCP

Data Structure

```
struct TaskEntry {
  char OPHd;
  char NOP;
  char IPHd;
  char NIP;
} TaskL[NT];
```

```
struct OPEntry {
  char cur;
  char prev;
  char BufHd;
  char NB;
} OPL[SysNOP];
```

```
struct IPEntry {
  char SrcPt;
  char delay;
} IPL[SysNIP];
```

```
message WrtInit[SysNOP],Buf[SysNB];
char Read[SysNIP];
```

Initialization

```
OPL[0].cur = OPL[0].prev = OPL[0].BufHd = 0;
Buf[OPL[0].BufHd] = WrtInit[0];
for (i = 1; i < SysNOP; i++) {
  OPL[i].BufHd = OPL[i-1].BufHd + OPL[i-1].NB;
  OPL[i].cur = OPL[i].prev = OPL[i].BufHd;
  Buf[OPL[i].BufHd] = WrtInit[i];
}
```

Figure 9. DS Initialization for TCCP

task ports need to be processed.

Unlike DBP, no bookkeeping operation is required for readers at termination time. Since the writer writes data into a circular buffer, the `FindFree()` simply increments `cur` modulo the buffer size, and returns the remainder as the new `cur`. Because there may be multiple writers in the system, the `FindFree()` takes as argument the index of the writer and returns its queue reference `FreeHd`. The memory requirements of the

```
/* activation time */
/* each writer i */
OPL[i].prev = OPL[i].cur;
OPL[i].cur = FindFree(i);

/* each reader i */
i2 = IPL[i].SrcPt;
if (IPL[i].delay)
  Read[i] = OPL[i2].prev;
else
  Read[i] = OPL[i2].cur;
```

```
/* execution time */
...
/* each writer k */
Buf[OPL[k].cur] = ...
...
/* each reader k */
... = Buf[Read[k]];
...
char FindFree(char idx) {
  return (OPL[idx].cur+1)\
              % OPL[idx].NB;
} /* O(1) */
```

Figure 10. Application Tasks for TCCP

implementation are shown in Table 3.

variable	char	message
count	4×(NT+SysOP)+3×SysNIP	SysNB+SysNOP

Table 3. TCCP Memory Requirement

4.3 Comparison of DBP and TCCP

From the above discussion, it is clear that more operations are needed at kernel level for the writer and lower-priority readers in the DBP case. Furthermore, lower-priority readers must update the shared use free list before their termination and any correct implementation must guarantee mutual exclusion. Its constant execution time `FindFree()` requires more memory and has additional time overhead. Compared with DBP, TCCP requires a smaller amount of memory for auxiliary data structures and it is simpler since the TCCP can achieve a constant time `FindFree()` without introducing extra data structures. However, the buffer size based on the temporal concurrency control is highly dependent upon the temporal properties of the writer and reader tasks, and it may be much larger than the DBP case. The tradeoffs among time, memory, and implementation complexity need to be analyzed to select the best implementation.

5 OSEK/VDX

SR semantics-preserving protocols need kernel-level support to assign reading and writing buffer indexes. To support portability of real-time application software, RTOS API standards such as OSEK/VDX, POSIX [9], and μITRON [10] have been developed. In this paper, we choose OSEK/VDX as the OS platform for our implementation. The OSEK/VDX standard originated from France and Germany and is widely used in the automotive industry.

Three processing levels are defined in OSEK. From higher to lower priority, they are interrupt level, logical scheduler level, and task level. To support design reuse and to ease upgrade, four conformance classes are defined according to the number of active activations of a task, the task type, and the number of tasks per priority level. Based on whether they can enter a wait state by calling the `WaitEvent` kernel service, tasks are categorized as either basic or extended. A basic task is not allowed to wait on an event. Minimum requirements are defined for the four conformance classes as shown in the columns of Table 4.

	Basic		Extended	
	BCC1	BCC2	ECC1	ECC2
Multiple Active Task Instances	No	Yes	BT: No ET: No	BT: Yes ET: No
# of Tasks not in Suspend State	8		16 (Any Comb. of BT/ET)	
> 1 Task/Priority	No	Yes	No	Yes
# of Events/Task	-		8	
# of Priority Levels	8		16	
Resources	RES_SCHEDULER		8(including RES_SCHEDULER)	
Internal Resources	2			
Alarm	1			
Application Mode	1			

Table 4. Minimum Requirements for OSEK CC

In OSEK, the kernel functionality includes the task management, interrupt management, synchronization, alarm, intra-processor message handling, and error treatment. A task can be activated by either `ActivateTask` or `ChainTask` and must call `TerminateTask` before its termination.

An Interrupt Service Routine (ISR) has a statically assigned priority level higher than those of the tasks. The OSEK OS standard specifies two categories of ISRs. An ISR of category 1 is not allowed to use any kernel services and cannot be preempted. ISRs of category 2 allow calling kernel primitives and, at the end of their execution, rescheduling occurs if there is no other pending interrupt.

Synchronization can be achieved by using events or semaphores. An event is owned by an extended task and it can be set by either a basic task, an extended task, or even a category 2 ISR, but the kernel primitive `WaitEvent` can only be invoked by extended tasks.

Alarms are managed in a layered manner. At least one counter is generated from a hardware or software timer. The counter can be used as a time reference for alarm generation. An alarm, associated with a counter, can be used to activate a task, set an event, or call a callback routine. OSEK supports absolute and relative alarms, single instance or cyclic.

Finally, the hook routine mechanism is used for error handling, tracing, and debugging purposes. This mechanism allows application specific functionality to be processed by the OSEK OS when kernel events occur. A hook routine has a priority that is higher than all application tasks and cannot be preempted by ISRs in category 2.

5.1 OSEK Development Process

Figure 11 illustrates the OSEK development process. The OIL (OSEK Implementation Language) [11] declarations are used to configure an OSEK application. The OIL description of an OSEK application consists of a set of OIL objects, characterized by attributes and references. Refer to Table 5 for all OSEK OIL objects and their properties. An OIL con-

Object	Mandatory	Standard Attribute	Std Reference
CPU	yes	-	-
OS	yes $(=1)$	STATUS; USERESSCHEDULE; USEGETSERVICEID; Hooks; USEPARAMETERACCESS	-
APPMODE	yes (≥ 1)	-	-
TASK	yes (≥ 1)	PRIORITY; SCHEDULE; ACTIVATION; AUTOSTART	MESSAGE; EVENT; RESOURCE
COUNTER	no	MAXALLOWEDVALUE; TICKSPERBASE; MINCYCLE	-
RESOURCE	no	RESOURCEPROPERTY	-
EVENT	no	MASK	-
ISR	no	CATEGORY	MESSAGE;RESOURCE
MESSAGE	no	NOTIFICATION; etc.	-
NWMESSAGE	no	SIZEINBITS; etc.	IPDU
COM	no $(=1)$	COMTIMEBASE; etc.	-
IPDU	no	SIZEINBITS; etc.	-
NM	no $(=1)$	-	-

Table 5. OIL Objects and Their Properties

figuration is composed of two parts: the implementation definition and the application definition. The former defines all standard and application-specific attributes and their properties for a particular OS implementation while the latter defines the set of objects and their corresponding attribute values for an OSEK application.

An OIL configuration file, coded manually or generated automatically, is fed to System Generator (SG), which automatically configures a kernel by choosing the required modules and customizing the data structure attributes. The source code of the application tasks, the selected module files from the OSEK OS kernel library, and the additional application file produced by SG are compiled and linked together to produce an executable file for the application.

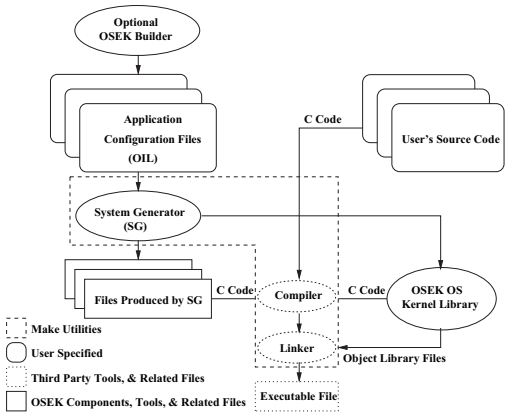


Figure 11. Application Development Process

6 OSEK Implementations of SR Semantics Preserving Protocols

After the descriptions of SR semantics preserving protocols and of the OSEK basics, we describe their implementation in a portable BCC1 conformance class. Please note, *only standard features of OSEK are used, and no modification to the kernel is required.* The implementation that was previously referred as *kernel-level* will often be performed by a higher priority task or Os hook routines that cannot be interrupted and can therefore guarantee atomicity. In BCC1 and BCC2, events are not available and the alarm mechanism is the only way to activate periodic tasks. Since the minimum requirements allow one alarm only, we use it to periodically activate a `dispatcher` task that, in turn, activates the application tasks at their rates. `dispatcher` is periodically activated by an alarm, statically configured as cyclic, every `GCDR` time units, which denote the Greatest Common Divider of the Rates of application tasks.

The data structures for the task `dispatcher` are declared in Figure 12. The array `TickL[]` has dimension `LCMR`, the Least Common Multiple of the Rates of application tasks. Each `TickL[i]` entry has two fields: `DispHd` and `size`. `DispHd` points to the first task on the dispatch table `DTab[]` and `size` indicates the number of tasks that need to be activated at this specific `tick` value. The array `DTab[]` contains the tasks that need to be activated from `tick = 0` to `tick = LCMR − 1`. The entries of `DTab[]` are used to index the tasks in the task descriptor array presented earlier.

The top right column in Figure 12 is the initialization of the data structures used by `dispatcher`. The bottom part of the figure shows the `dispatcher` im-

```
Declaration
struct TickEntry {
  char DispHd;
  char size;
} TickL[LCMR];

char tick;
char DTab[TSize];

Compute TSize
char TSize = 0;
for (i=0; i<NT; i++) {
  TSize += \
    LCMR/TaskL[i].rate;
}

Init w/o Phase Shift
tick = -1; i1 = 0;
for (j = 0; j < LCMR; j++) {
  TickL[j].DispHd = -1;
  TickL[j].size = 0;
  for (i = 0; i < NT; i++) {
    if (j%TaskL[i].rate==0) {
      i2 = TickL[j].size + i1;
      DTab[i2] = i;
      TickL[j].size++;
  } }
  if (TickL[j].size != 0) {
    TickL[j].DispHd = i1;
    i1 += TickL[j].size;
} }

Implementation of Task dispatcher
Task (dispatcher) {
  tick = (tick+1) % LCMR;
  if (TickL[tick].DispHd != -1) {
    for (k = 0; k < TickL[tick].size; k++) {
      idx = DTab[k+TickL[tick].DispHd]; /* task id */
      for (i=0; i<TaskL[idx].NOP; i++) { /* writers */
        idx2 = TaskL[idx].OPHd + i;
        ··· /* kernel level writer code */
      }
      for (i=0; i<TaskL[idx].NIP; i++) { /* readers */
        idx2 = TaskL[idx].IPHd + i;
        ··· /* kernel level reader code */
      }
      ActivateTask(idx);
  } }
  TerminateTask();
}
```

Figure 12. Task Dispatcher

plementation. During its execution, the counter `tick` is incremented modulo `LCMR`. Then, the value of the field `DispHd` of `TickL[tick]` is checked. If it is "-1", no task needs to be activated. Otherwise, the tasks in `DTab[]`, as specified by the `(DispHd, size)` are processed. For each of them, the dispatcher processes its input and output ports, performing the read or write procedures specified by the earlier protocols. Specifically, it calls `FindFree()` to find a safe buffer slot for each output port. For each reader, it defines the buffer slot that the task will be using during its execution. Then, `dispatcher` activates the task by calling `ActivateTask` and, at the end, calls `TerminateTask` to terminate.

```
TASK (init) {
  ··· /* init implementation specific auxiliary DS */
  ··· /* init DS required by protocol */
  ··· /* init dispatcher as in Fig 12 */
  SetRelAlarm(dispAlarm, 0, GCDR);
  TerminateTask();
}
```

Figure 13. General Structure of Task Init

The data structures of the communication protocols discussed in Section 4 need to be initialized to obtain a correct execution. In addition, the data structures of the task `dispatcher` are initialized as shown in Figure 12 by the OSEK task `init`, (Figure 13), executed at system startup. The data structures storing static information, such as `isHPR`, are also initialized at system startup. Besides initializing data structures, task `init` also sets the cyclic alarm, `dispAlarm`, associated with task `dispatcher`.

6.1 OSEK Communication Implementation in C

In this section, we show the data structures, the declarations, and the definitions for `dispatcher`, the application tasks, and `init` for each implementation. The declaration of the data structures that are common to the two implementations is in Figure 14.

```
#DEFINE NT X          #DEFINE SysNIP X    #DEFINE TSize X
#DEFINE LCMR X        #DEFINE SysNOP X    #DEFINE SysNB X
#DEFINE GCDR X

message Buf[SysNB];                       char Read[SysNIP];
message WrtInit[SysNOP] = {X, ···};
```

Figure 14. Common DS Declaration

6.1.1 DBP

Figure 15 shows the data structures, which are a combination of those used for `dispatcher` in Figure 12 and for DBP in Figure 5 with additional fields in the task descriptor. In addition to `pri` and `rate`, a field called `done` is added for the purpose of detecting when a context switch is executed upon the termination of a lower priority reader. A field called `owner` is added to the port descriptors. The corresponding declarations are shown in Figures 16, 14, and 12. The `dispatcher` task has the same structure as shown in Figure 12 and the kernel-level code for the DBP from Figure 7 is executed. Similarly, the `init` task has the same structure of Figure 13. Figure 17 shows the application-level code required by DBP. We propose to use the hook mechanism provided by OSEK to let lower-priority readers atomically update the buffer free list at termination time. Specifically, we use the `PostTaskHook` to execute

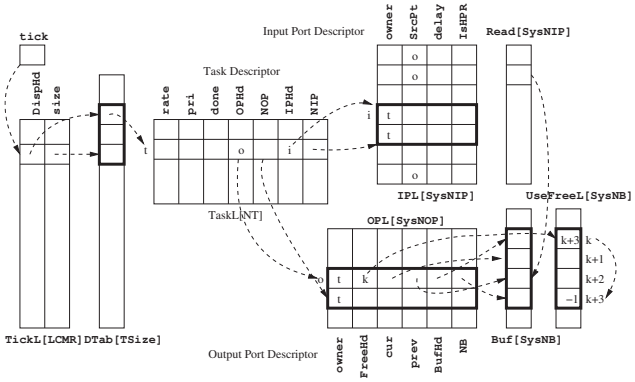


Figure 15. CTDBP Implementation DS

```
struct TaskEntry {
  char rate;
  char pri;
  char done;
  char OPHd;
  char NOP;
  char IPHd;
  char NIP;
} TaskL[NT] = {
  {X,X,0,X,X,X,X},
  ···};
```

```
struct OPEntry {
  char owner;
  char FreeHd;
  char cur;
  char prev;
  char BufHd;
  char NB;
} OPL[SysNOP] = {
  {X,0,0,0,0,X},
  ···};
```

```
struct IPEntry {
  char owner;
  char SrcPt;
  char delay;
  char IsHPR;
} IPL[SysNIP] = {
  {X,X,X,X},
  ···
};
```

```
char UseFreeL[SysNB];
```

Figure 16. DS Declaration for DBP

```
TASK (AppTask_i) {
  TaskL[i].done = false;
  ···
  Buf[OPL[k].cur] = ···
  ···
  ··· = Buf[Read[k]];
  ···
  TaskL[i].done = true;
  /* atomic hook code */
  TerminateTask();
}
```

```
void PostTaskHook(void) {
  char id, j, k, nip, t1, t2;
  GetTaskID(id);
  if (TaskL[id].done) {
    nip = TaskL[id].NIP;
    for (j=0; j<nip; j++) {
      k = j + TaskL[id].IPHd;
      ··· /* CS in Fig 7 */
    }
  }
}
```

Figure 17. OSEK Implementation of Application Task for DBP

a critical section upon the termination of these tasks. Since the `PostTaskHook` routine executes at each context switch and for all the tasks in the system, a flag `done` is added to the task descriptor. The flag indicates for which tasks the `PostTaskHook` needs to be executed and also ensures that the operations in the `PostTaskHook` are only executed at task termination time. The `done` flag of each task is set to false at the beginning of the task, and changed to true with the last task instruction.

`PostTaskHook` (Figure 17), first obtains the identifier of the active task by calling the OSEK API `GetTaskID`. Then, it checks whether its `done` flag is set to true. If so, the updates required by the communication protocol are performed.

6.1.2 TCCP

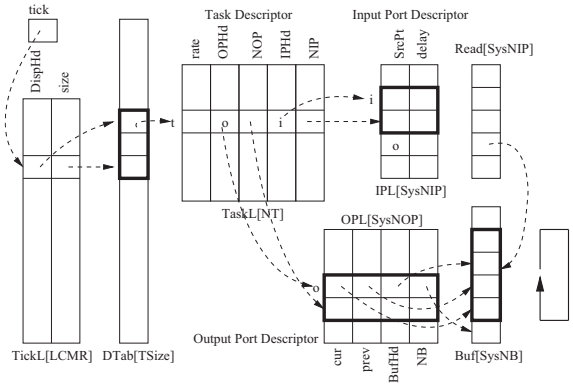


Figure 18. TCCP Implementation DS (MPT)

Figure 18 shows the data structures used for the OSEK implementation of TCCP as a combination of those in Figures 12 and 8 with the additional `rate` field. Compared with Figure 15, the data structures for the TCCP implementation are simpler, since no bookkeeping is required by the `FindFree()`. The declarations are shown in Figure 19, 14, and 12.

```
struct TaskEntry {
  char rate;
  char OPHd;
  char NOP;
  char IPHd;
  char NIP;
} TaskL[NT] = {
  {X,X,X,X,X}, ···};
```

```
struct OPEntry {
  char cur;
  char prev;
  char BufHd;
  char NB;
}OPL[SysNOP] = {
  {0, 0, 0, X},
  ···}
```

```
struct IPEntry {
  char SrcPt;
  char delay;
}IPL[SysNIP] = {
  {X, X},
  ···
};
```

Figure 19. DS Declaration for TCCP

Similar to the implementation of the DBP discussed in Section 6.1.1, the `dispatcher` task executes the kernel-level code, and the `init` task executes the initialization code. The definition of application tasks shares the same structure with its counterpart in Figure 17, but is simpler because there is no termination code for the input ports and no hook routine.

6.1.3 Implementation Comparison

In this section, we compare the two implementations. Table 6 shows the memory requirements. DBP requires more auxiliary data structures than TCCP, including the counter/free-list structure. The buffer sizes for the DBP and the TCCP are not comparable since they are based on different buffer sizing mechanisms, but DBP can be used to have smaller memory requirements [12]. The DBP implementation is more complex because of

protocol	char	message
DBP	7×NT+6×SysNOP+5×SysNIP+ 2×LCMR+TSize+SysNB+1	SysNB+ SysNOP
TCCP	5×NT+4×SysNOP+3×SysNIP+ 2×LCMR+TSize+1	$SysNB_T$+ SysNOP

Table 6. Memory Requirement Comparison

the code required for finding the free buffer and for accounting for the buffer usage, and the necessity to update the shared use free list. Furthermore, since the hook mechanism is mainly designed for debugging and error management, the use of `PostTaskHook` introduces a time overhead at each context switch.

6.2 OIL Configuration File

In this section, we define the OIL configuration file for the implementation of the communication protocols. Figure 20 shows the basic structure of an OIL configuration file. Inside the container `CPU` declaration, objects are statically specified. The application tasks are defined by the generic declaration of `AppTask_j`. We set the `SCHEDULE` attribute as `FULL`, indicating a fully preemptive scheduling policy. Under the assumption that the deadlines of application tasks are not

```
OIL_VERSION = "2.5";
/* Implementation Def */
IMPLEMENTATION myOSEKOS {
  ...
}; // End of myOSEKOS
/* Application Def */
CPU myCPU { // container
  /* OS Object */
  OS myOS {
    STATUS = STANDARD;
    STARTUPHOOK = FALSE;
    ERRORHOOK = FALSE;
    SHUTDOWNHOOK = FALSE;
    PRETASKHOOK = FALSE;
    POSTTASKHOOK = TRUE;
    USEGETSERVICEID = FALSE;
    USERESSCHEDULER = FALSE;
  };
  /* Task Object */
  TASK AppTask_j {
    PRIORITY = X_j;
    SCHEDULE = FULL;
    ACTIVATION = 1;
    AUTOSTART = FALSE;
  };
  ...
  TASK dispatcher {
    PRIORITY = X_d;
    SCHEDULE = NON;
    ACTIVATION = 1;
    AUTOSTART = FALSE;
  };
  TASK init {
     PRIORITY = X_i;
     SCHEDULE = NON;
     ACTIVATION = 1;
     AUTOSTART = TRUE {
       APPMODE = AppMode0;
     };
  };
  /* Alarm Object */
  ALARM dispAlarm {
    COUNTER = SysTimer;
    ACTION = ACTIVATETASK{
      TASK = dispatcher;
    };
    AUTOSTART = TRUE {
      ALARMTIME = 0;
      CYCLETIME = GCDR;
      APPMODE = AppMode0;
    };
  };
  /* Counter Object */
  COUNTER SysTimer {
    MINCYCLE = x;
    MAXALLOWEDVALUE = x;
    TICKSPERBASE = x;
  };
  /* Appl Mode Object */
  APPMODE AppMode0 {
    VALUE = AUTO;
  };
}; // End of myCPU
```

Figure 20. OIL Configuration File

greater than their respective periods, the `ACTIVATION` attribute is set to one (as required in BCC1). Application tasks are periodic and are activated by task `dispatcher`, therefore the attribute `AUTOSTART` is set to `FALSE`. For task `init`, the configuration is similarly specified, with the `AUTOSTART` attribute turned to on and a single application mode assigned to the `APPMODE` attribute. Task `dispatcher` activates the application tasks and performs part of the communication protocol operations on behalf of the kernel. Therefore, its priority should be higher than those of all application tasks, and its `SCHEDULE` attribute is set to `NON`, indicating a non-preemptive scheduling. Task `dispatcher` is activated by an alarm, `dispAlarm`, so its `AUTOSTART` attribute is set to `FALSE` and an alarm object is specified accordingly. The alarm is associated with a counter, which is an object defined in the OIL file. The alarm is configured to activate task `dispatcher` through setting its attribute `ACTION` as `ACTIVATETASK`. Finally the alarm's `AUTOSTART` attribute is set to `TRUE` and the period of `dispAlarm` is set to `GCDR`.

When the constant time `FindFree()` is used, the atomicity of the termination code that updates the shared use free list is guaranteed by the `PostTaskHook` mechanism, which is turned on by setting the corresponding attribute `POSTTASKHOOK` as TRUE in the OS object as shown in Figure 20.

7 Conclusions and Further Work

We presented two portable OSEK implementations for synchronous reactive semantics preserving communication protocols. We showed detailed data structures and imperative code for the dynamic buffering protocol with constant time complexity. Our OSEK implementation meets the minimum requirements of BCC1 for portability. Comparison of different versions of the implementation shows that TCCP uses a smaller amount of auxiliary data structures and it has a lower implementation complexity, with respect to DBP. As ongoing work, we are conducting experiments with different versions of the implementations of the presented protocols to study and verify the temporal, spatial, and implementation complexity tradeoffs.

References

[1] J. Chen and A. Burns, "A fully asynchronous reader/write mechanism for multiprocessor real-time systems," Tech. Rep. YCS 288, University of York, May 1997.

[2] J. Chen and A. Burns, "A three-slot asynchronous reader/writer mechanism for multiprocessor real-time systems," Tech. Rep. YCS 286, University of York, January 1997.

[3] N. Scaife and P. Caspi, "Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems," in *6th ECRTS Conference*, July 2004.

[4] C. Sofronis, S. Tripakis, and P. Caspi, "A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling," *Proceedings of the 6th ACM EMSOFT conference*, October 2006.

[5] J. Chen and A. Burns, "Loop-free asynchronous data sharing in multiprocessor real-time systems based on timing properties," in *Proc. of the 6th RTCSA Conference*, 1999.

[6] H. Kopetz and J. Reisinger, "The non-blocking write protocol nbw: A solution to a real-time synchronization problem," in *Proceedings of the 14th IEEE RTSS*, December 1993.

[7] M. Baleani, A. Ferrari, L. Mangeruca, and A. S. Vincentelli, "Efficient embedded software design with synchronous models," in *Proceedings of the 5th ACM EMSOFT conference*, 2005.

[8] G. Wang, M. D. Natale, and A. Sangiovanni-Vincentelli, "An osek/vdx implementation of synchronous reactive semantics preserving communication protococls," Tech. Rep. UCB/EECS-2007-81, EECS Dept., University of California, Berkeley, June 2007.

[9] POSIX, "POSIX standard." available at `http://www.posix.com`.

[10] microITRON, "microITRON standard, version 4.0." available at `www.sakamura-lab.org/TRON/ITRON/DOC/`.

[11] OSEK, "OSEK implementation language (OIL), version 2.5." available at `http://www.osek-vdx.org`.

[12] M. D. Natale, G. Wang, and A. Sangiovanni-Vincentelli, "Optimizing the implementation of communication in synchronous reactive models," in *Submitted to the EMSOFT Conference*, 2007.

Coordinated Allocation and Scheduling of Multiple Resources in Real-time Operating Systems

Kartik Gopalan and Kyoung-Don Kang
Computer Science, State University of New York at Binghamton
Binghamton, NY 13902-6000
{kartik,kang}@cs.binghamton.edu

ABSTRACT

Distributed real-time embedded (DRE) systems are key components of critical infrastructure including surveillance, target tracking, electric grid management, traffic control, avionics, and communications systems. They require (1) the coordinated management of multiple resources, such as the CPU, network, and disk, (2) end-to-end (E2E) real-time guarantees across the use of multiple resources, and (3) feedback control across multiple resources. None of these properties is supported as a first-class feature within the state-of-the-art real-time operating systems, but are left out as an inconvenient detail to be managed by DRE application programmers. In this paper, we shed light on this fundamental problem and make the case for greater research into the development of theory and a runtime systems for coordinated allocation and scheduling of multiple resources in real-time operating systems. We also present the outlines of our proposed solution approach, called the Multiple Resource Allocation and Scheduling (*MURALS*) framework, that aims to bridge this gap between the need for E2E timing requirements and the techniques to coordinate the use of multiple resources.

1. INTRODUCTION

Distributed real-time embedded (DRE) systems are key components of critical infrastructure including surveillance, target tracking, electric grid management, traffic control and safety, process control, robotics, avionics, communication systems, and even real-time networked games. DRE systems in these applications are required to use multiple heterogeneous resources, such the CPU, network bandwidth, main memory, and secondary storage, The heterogeneity of resources and their interactions calls for coordinated management across these resources to meet end-to-end (E2E) deadlines. While real-time scheduling for a single resource in isolation has been studied extensively, relatively little work has been done for *integrated allocation and scheduling of multiple heterogeneous resources* to meet E2E timing constraints. Absence of coordination across multiple resources can lead to failure in meeting E2E timing guarantees in critical DRE systems.

An example DRE application is a surveillance network [37, 49], which consists of a group of cameras (and other sensors) connected over an area of interest. Each camera periodically captures a video frame, which is compressed by an embedded processor and transmitted over a wired or wireless LAN to a command and control (C2) center. A C2 server receives compressed video frames from multiple cameras across the LAN and executes several concurrent activities, such as to monitor the battlefield or traffic status. The server decompresses the video frames, displays the video on monitors, processes each frame for surveillance purposes, triggers alarms for security or safety reasons if necessary, and logs the data to a storage device. In this example, an E2E real-time task is associated with an E2E deadline to support the required application QoS. It also consists of multiple distributed subtasks using different system resources. A subtask depends upon the successful and timely completion of the previous subtask(s) in the sequence, forming precedence constraints. To summarize, the following characteristics of DRE applications emerge: (1) Multiple heterogeneous resources are used; (2) The resource usages within an application are ordered forming a precedence graph; and (3) Execution of a repetitive sequence of correlated subtasks is bounded by an E2E deadline. Our focus is on the applications with the above characteristics.

In this paper, we make the case for greater research into the development of theory and runtime systems for coordinated allocation and scheduling of multiple resources. We discuss four key open research problems and possible solution strategies.

(1) Deadline Partitioning Techniques: Given a DRE task that requires the use of multiple resources to meet its E2E deadline, one needs a deadline partitioning algorithm during admission control and resource allocation that apportions the the E2E delay budget among the subtasks of the DRE task. Careful deadline partitioning is important because it determines the load on each individual resource. In particular, tighter delay budget at a resource can lead to higher resource load, resulting in fewer admissible E2E DRE tasks in future. We investigate algorithms to efficiently partition E2E deadlines among multiple underlying resources. The goal is to increase the success ratio, i.e., the fraction of submitted DRE tasks that are admitted and completed within their E2E deadlines. The key idea is to reduce the extent of load imbalance among different resources during deadline assignment to prevent formation of resource bottlenecks. Although deadline assignments in multiprocessor systems have previously been studied [51], most existing research considers only a single isolated resource.

(2) Coordinated Runtime Scheduling of Multiple Resources: While an effective deadline partitioning algorithm

is necessary to assign a delay budget to each subtask of an E2E task, it is not sufficient by itself to guarantee that the E2E deadline will be met. During task execution, one needs explicit coordination across runtime schedulers of different resources to ensure that each subtask is scheduled to complete before its assigned sub-deadline. This is not the case in traditional RTOSs where scheduling decisions at one resource are made oblivious of the scheduling decisions at other resources. It thus becomes the DRE application writer's responsibility to manage any cross resource timing dependencies among subtasks. We illustrate this problem and suggest possible approaches to address such scheduling dependencies at runtime.

(3) Statistical Performance Guarantees: Reserving resources for worst-case load requirements may lead to resource under-utilization in the common case of low offered load. Additionally, a number of DRE applications, such as visual tracking and traffic monitoring, can adapt to a small probability of violations in their E2E guarantees. In this light, the multiple resource allocation techniques could potentially exploit the statistical multiplexing nature of the resource usage among concurrent DRE tasks to improve the system's overall resource utilization efficiency. Thus one needs statistical multi-resource allocation algorithms, such as online measurement-based techniques, that can exploit the statistical multiplexing nature of the resource usage and distinct tolerance levels to QoS violations, to reduce overall resource requirements of DRE applications. We investigate the role of resource allocation algorithms that exploit statistical multiplexing effects across multiple resources not just along the traditional 'bandwidth' dimension, but also along an orthogonal 'delay' dimension. We outline algorithms that can support tasks with *distinct probabilistic delay guarantees*, i.e., if certain tasks can tolerate more delay violations, they can reserve less resources than the other tasks tolerating fewer violations.

(4) Feedback Control Across Multiple Resources. Workloads may dynamically vary in DRE applications. For example, the image processing frequency and compression ratio may change depending on the presence or absence of objects indicating security breaches or traffic jams. Further, resource could be overbooked due to statistical multiplexing. As a result, E2E deadlines can be missed. Thus one requires control theoretic techniques to support E2E timing guarantees across multiple resources in unpredictable environments. In general, statistical approaches can provide high-level resource usage monitoring and QoS management, while control theoretic approaches can support fine-grained QoS management to ensure that, for example, no more than 1% of E2E deadlines are missed in average, no more than 1.5% of deadlines are missed even when the system is in a transient state, and a transient miss ratio overshoot, if any, decays within the specified settling time. Statistical and control theoretic approaches for QoS management have been studied separately; however, interactions between them have rarely been investigated [66]. We outline possible approaches for applying a combination of statistical and control theoretic techniques in multiple resource environments.

We also present the outlines of our proposed solution approach, called the *Multiple Resource Allocation and Scheduling* (*MURALS*) framework and describe the ongoing development a proof-of-concept *MURALS* testbed on top of an commodity RTOS. The testbed includes not only kernel-level coordinated resource allocation mechanisms but also declarative APIs for E2E QoS specification. The API allows DRE application programmers to specify E2E tasks, their deadlines, periods, and precedence constraints across multiple resources. This information is utilized by kernel-level measurement-based QoS mapping scheme to automatically derive the low-level resource requirements from high-level performance requirements. Despite its importance, very little prior work has been conducted to provide declarative APIs and kernel support for deadline partitioning, coordinated scheduling, statistical guarantees, and feedback control. The goal of this paper is to make a case for greater research into this increasingly important subject.

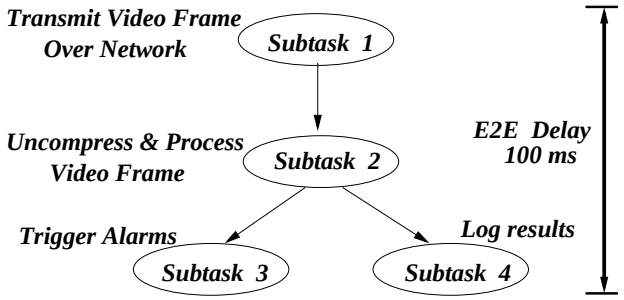


Figure 1: A task precedence graph for video surveillance with an E2E deadline.

2. DEADLINE PARTITIONING

A typical DRE task executes a set of subtasks related to each other via precedence constraints. For instance, Figure 1 shows the precedence graph of a video surveillance task executing four subtasks every period. Subtask 1 corresponds to the periodic compression and transmission of a video frame from a remote camera to a C2 server. At the C2 server, subtask 2 decompresses and processes the frame for target tracking, subtask 3 triggers alarms if necessary, and subtask 4 logs the results to a storage device in real-time. In this example, subtasks 3 and 4 can proceed concurrently once subtask 2 completes.

A precedence graph only describes the partial ordering but not the timing relationships among subtasks. For instance, the surveillance application may need to perform the E2E processing of each video frame within 100ms. This application-level performance requirement imposes a timing constraint for the E2E task and its subtasks in the precedence graph. Guaranteeing application-level QoS requires more than just local real-time scheduling for each individual resource, because it can only guarantee the subtask level QoS. For the DRE task shown in Figure 1, subtask 1 must be completed early enough to leave time for subtasks 2, 3, and 4 to complete before the E2E deadline.

A key problem illustrated in this example is *how to partition the E2E task deadline to meet the timing and precedence constraints, while improving the overall efficiency of resource utilization and E2E success ratio.* This requires specific algorithms that assign intermediate sub-deadlines for each subtask, while accounting for current and future load at each resource. Assigning a sub-deadline to a sub-

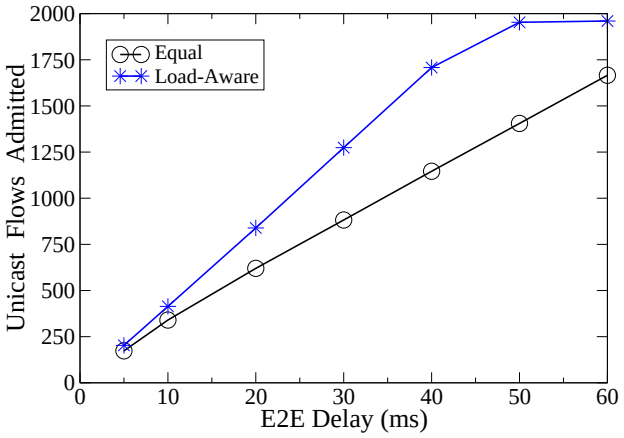


Figure 2: Flows admitted vs. E2E delay bound over Sprint IP Backbone. Hops=6. Flow data rate=100kbps. Link speeds:45–200Mbps.

task also entails a specification of the load on the subtask's corresponding resource; in general, a tighter sub-deadline implies a higher resource load. Therefore, partitioning the delay budget, i.e., the E2E deadline, opens up an opportunity for load balancing across multiple resources resulting in efficient resource utilization. Rate-based schedulers, such as Virtual Clock [65] or WFQ [43], permit explicit mapping between latency bound requirements and bandwidth reservations [64]. Thus, the queuing delay experienced by a subtask at any resource scheduler is inversely proportional to the bandwidth reserved on its behalf. If a subtask needs a smaller delay budget, the corresponding resource reservation has to be larger, which imposes a heavier load on the resource. It is possible that some resources in the system may be more heavily loaded than others. Thus, one could partition the E2E deadline in such a manner that the more loaded resources are assigned a larger proportion of the E2E delay budget. This ensures that critical resources do not deplete long before less critical resources, thus preventing system-wide bottlenecks.

In the previous surveillance example, suppose the E2E deadline is 120ms. Consider that a UAV (Unmanned Aerial Vehicle) or traffic sensing node can capture and compress a fixed size video frame in 20ms using the dedicated embedded processor, whereas the network transmission delay and processing delay at the C2 server are variable as these resources are shared with other surveillance nodes. Thus, the remaining delay budget of 100ms needs to be partitioned among the network link and the C2 server's CPU. Assume that the C2 server's CPU is already 20% utilized and network link is 80% utilized. Instead of equally partitioning the delay budget between the network link and the CPU, a more sensible partition can be to assign 20ms to the CPU and 80ms to the network in proportion to their respective loads, and still meet the E2E deadlines.

This approach has shown promising initial results [17, 18] in which our delay partitioning techniques significantly reduce the load imbalance across multiple resources. In [17], we proposed a load-aware delay partitioning approach for sequential precedence graphs. We showed that one can increase the number of tasks admitted into the system by assigning delay budget D_i to each subtask i as per Equation 1 below, such that the E2E delay bound $D \geq \sum_i D_i$.

$$D_i = M_i + \frac{k_i M_i}{\sum_{j=1}^{m} k_j M_j} S \ , \quad k_i = \sqrt{\frac{W_{avg_i}}{W_i}} \qquad (1)$$

Here M_i is the minimum delay budget required to complete the subtask i, W_i is the amount of work such as the number of CPU cycles or bytes read/written, S is the slack in delay budget given by $S = D - \sum_i M_i$, and W_{avg_i} is the average amount of work requested by a DRE task at resource i. In [18], we proposed an iterative algorithm for delay partitioning along multi-hop network paths where each network link corresponds to one resource. Figure 2 shows that our load-aware delay partitioning algorithm admits up to 39% more network flows compared to equal partitioning, under the same network setup and input workload.

There are several open research challenges in the deadline partitioning problem.

Resource Specific Overheads: In practice, an exact algorithm for deadline partitioning across different resources needs to consider hard-to-characterize overheads such as context switching, network transmission overheads, and disk seek/rotational latency. Unfortunately, a large body of real-time scheduling theory ignores these overheads. Thus, a set of tasks may not be perfectly schedulable across multiple resources even if the available capacity is theoretically sufficient. Thus there is a need for a global admission control mechanism, which first consults each local resource scheduler to determine the minimum delay it can support using its current residual capacity. The delay partitioning algorithm could then apportion any remaining slack in E2E delay budget in a stepwise manner to decrease the demand on each resource. In each incremental step, the algorithm again needs to consult the local resource schedulers to determine how much the local delay will be increased due to the increase in resource demand and whether the tasks would still be schedulable. Such resource-aware algorithms for apportioning slack depend upon accurate mapping functions between the required delay bound and resource reservation for multiple resources, possibly distributed across a LAN.

Revising Earlier Allocations: There also exists a scope for algorithms that can make better deadline partitioning decisions, given additional knowledge of workloads. To leave as much resources unreserved as possible for the future use, our previous online admission control policy only considers a single DRE task for admission upon its arrival without changing the current reservations. The virtue of this approach is its simplicity and low overhead. However, if the delay partitioning algorithm has the flexibility of revising the previously assigned delay budgets of all the DRE tasks admitted earlier, then it can allocate *all* system resources to the current set of DRE tasks without leaving any unreserved resources. Thus one can design better resource allocation algorithms that can revise earlier allocation decisions, if necessary, to improve the efficiency of multi-resource utilization.

Optimizing for Power Efficiency: Another interesting open research problem is to exploit E2E delay partitioning to improve the power usage efficiency, instead of load balancing. Note that a shorter delay budget can translate to higher load and hence greater power dissipation at each re-

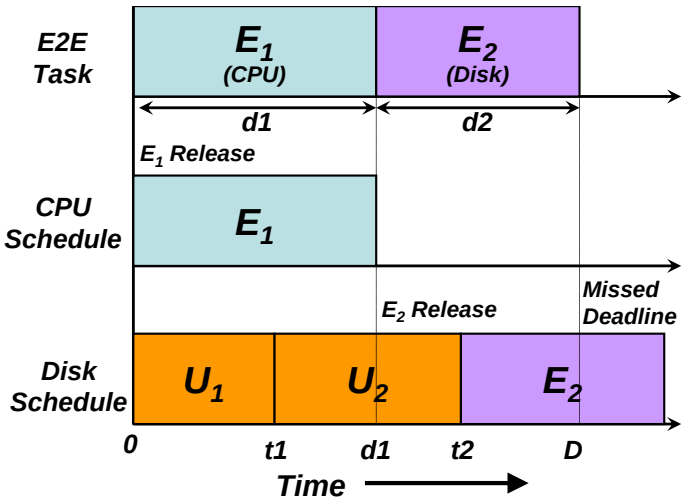


Figure 3: The inter-scheduler coordination problem at runtime for an E2E task with two subtasks, E_1 and E_2. The disk subtask E_2 misses the E2E deadline D after waiting for another non-preemptible I/O request U_2.

source. However, the mapping from delay budget to energy consumed for any given resource is likely to be a piecewise step rather than a continuous function. There is an opportunity to investigate different forms of such resource mapping functions and their impact on delay partitioning strategies that optimize the energy consumption at low-power resources. A useful metric to gauge both timeliness and power usage efficiency is the *effective power consumption* = *total_power_consumption*/*success_ratio*, lower value indicating greater effectiveness of delay partitioning algorithm. Existing work on Dynamic Slack Reclamation [34, 24] could be a starting point in this direction.

3. COORDINATED SCHEDULING

In the previous section, we discussed how effective partitioning of E2E delay budget among subtasks across multiple resources is essential to meet deadlines as well as to maintain high resource usage efficiency. However deadline partitioning constitutes only one component of the complete multi-resource allocation and scheduling framework. The other essential component is coordinated runtime scheduling of subtasks of an E2E task across multiple resources, in the absence of which even most judiciously partitioned deadlines might be missed.

To illustrate the problem, consider a simple example of an E2E real-time task E in Figure 3 that requires two resources – CPU and disk – to meet its E2E deadline of D. (We use disk I/O for illustration, though this example can easily be generalized for another I/O resource, such as network, or even for multiple I/O resources.) Assume that E2E delay budget D is partitioned as $d1$ and $d2$ across its CPU and disk subtasks, E_1 and E_2 respectively, where $d1 + d2 = D$. Both CPU and disk have their own independent real-time schedulers that, in the absence of any runtime coordination, maintain their own independent backlog queues of computation and I/O requests respectively. Assume that the subtask E_1 begins execution at CPU resource at time $t = 0$. In the meantime, the disk scheduler continues servicing other I/O requests U_1 and U_2 that are unrelated to the E2E real-time task E. At time $t1 < d1$, I/O request U_1 completes and U_2 is scheduled by the disk scheduler. At time $d1$, CPU subtask E1 completes and submits the real-time I/O task E_2 to the disk scheduler's runtime queue. However, even if E_2 is the most urgent I/O task at time $d1$, the disk scheduler needs to wait till time $t2$, when U_2 will complete service, before E_2 can be dispatched to the disk. This additional wait time could potentially cause E_2 to miss its I/O deadline, and consequently the E2E deadline, D.

Although the example above makes several simplifying assumptions about operating system behaviour, it serves to illustrate several fundamental problems when servicing a sequence of inter-dependent real-time subtasks across multiple resource schedulers.

Inter-Scheduler Coordination: Meeting the E2E deadline D depends upon the timely execution of component subtasks at *multiple* resource schedulers – here CPU and disk. In the absence of runtime coordination, each resource scheduler makes its own independent and locally optimal scheduling decisions because it is unaware of precedence constraints that control the release times of different real-time subtasks. For example, before time $d1$ in Figure 3, the disk scheduler was independently scheduling tasks U_1 and U_2, unaware of the fact that a real-time task E_2 was about to arrive with a tight deadline.

I/O Request Non-preemptibility: In general, I/O requests tend to be non-preemptible, once issued. Consequently, when dealing with multi-resource E2E tasks, a higher priority (static or dynamic) real-time I/O subtask may arrive at the I/O scheduling queue only to wait for the completion of a non-preemptible lower priority I/O request that's already in progress. Loosely speaking, this results in a short-term "priority inversion" that can potentially cause the higher priority real-time subtask to miss its E2E deadline.

Execution Time Prediction: While execution time of CPU subtasks can be predicted with reasonable accuracy, either through static code analysis or runtime profiling, the execution times for I/O subtasks are not as predictable. For example, disk response depends highly upon the seek and rotational latency whereas network response may depend upon switch congestion or channel access contention in broadcast media. In a multi-resource context, unpredictability in execution time might not only affect the timeliness the single I/O subtask in question, but also the timeliness of other dependent subtasks in the sequence of an E2E DRE task.

Advance Notification of Subtasks: None of the existing real-time operating systems address the above issues within a framework of coordinated multi-resource runtime scheduling, and it is undoubtedly a challenging research problem. To start making the problem manageable, our *MURALS* system incorporates an inter-scheduler coordination mechanism for multi-resource E2E real-time tasks that works as follows. Whenever a real-time E2E task begins execution, information about expected future arrival time and deadline of each of its subtasks are sent to the corresponding local resource

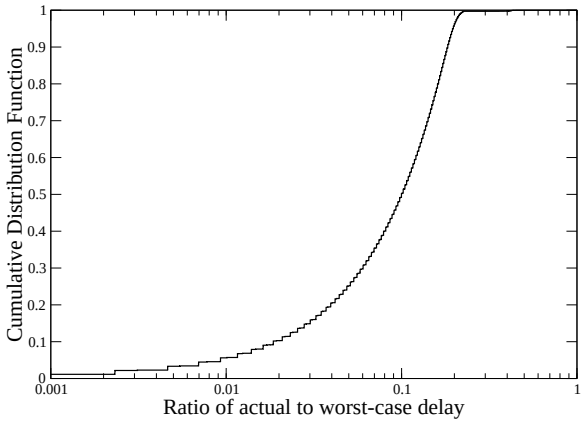


Figure 4: The CDF of the ratio of the actual delay experienced to the worst case delay expected by VoIP packets sharing a 10Mbps link.

schedulers. In the earlier example, at the start time 0, not only does the subtask E_1 begin execution at the CPU, but the disk scheduler is also given the information that subtask E_2, with delay budget of $d2$, will be ready at time $d1$. This enables the disk scheduler to insert an empty placeholder I/O request for task E_2 in its scheduling queue. At the scheduling time instant $t1$, the disk scheduler now has additional information to decide whether request U_2 can start and finish service early enough that the future real-time request E_2 can meet still its deadline. Admittedly, the advance notification to the disk scheduler may not provide a foolproof guarantee that E_2 will always meet its deadline (for instance a long non-preemptible I/O task that began before time instant 0 might yet delay E_2). Nor does advance notification address the inherent unpredictability of I/O completion times. However this additional inter-scheduler coordination does help the local resource schedulers make better informed scheduling decisions from timeliness standpoint at a larger number of scheduling instances than without the coordination. For example, one option that the disk scheduler can exercise at time $t1$ is to follow a non-work-conserving policy by not scheduling any request between $t1$ and $d1$, thus keeping the disk idle until the subtask E_2 becomes eligible for service. This enables the disk scheduler to sacrifice throughput for the sake of timeliness when necessary – an option it cannot exercise without the advance information about E_2.

4. STATISTICAL DELAY GUARANTEES

Another challenge faced in DRE systems using multiple resources is that of *resource under-utilization*. While reserving resources for the peak load ensures that individual DRE tasks always meet their performance targets under all conditions, it ignores the reality that individual resources do not encounter peak load situations in the common case. Additionally, a number of DRE applications, e.g., visual tracking and traffic monitoring, can adapt to a small probability of violations in their QoS guarantees. Thus, the multiple resource allocation techniques could exploit the statistical multiplexing nature of the resource usage among DRE tasks to improve the system's resource utilization efficiency.

Traditional approaches tend to exploit statistical multiplexing along the *bandwidth dimension*, where the offered load per resource is lower than the reserved bandwidth share,

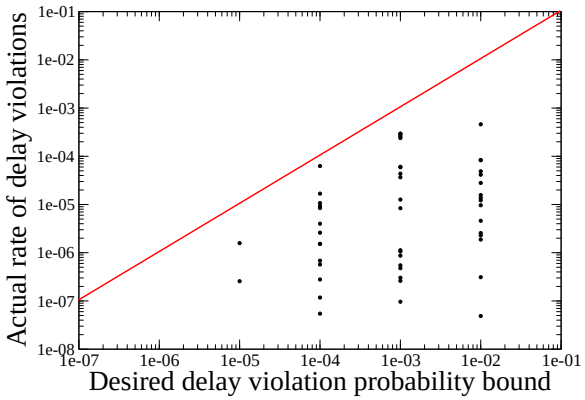


Figure 5: Effectiveness of DDM in differentiating between network flows with different tolerance to delay violations. Flows that have lower tolerance to violations experience fewer deadline misses. Each data point corresponds to one network flow. Delay bound=20ms. Link capacity = 10Mbps.

due to which one can oversubscribe the underlying resource. Additionally, one can also leverage statistical multiplexing along the **delay dimension**. The latter effect arises from the fact that not all real-time applications will generate their peak request bursts at the same time. For instance, among a set of ON-OFF Voice over IP (VoIP) flows traversing a shared network link, it is unlikely that all VoIP flows will be in their ON state simultaneously. The consequence of this multiplexing is that actual delays rarely approach worst-case delay bounds that are based on all real-time applications generating their peak burst simultaneously. Note that statistical multiplexing in the delay dimension is orthogonal to that along the bandwidth dimension.

Consider Figure 4 showing the cumulative distribution function (CDF) $Prob(r)$ of the ratio of the actual to worst case delay experienced by various network packets (subtasks) at a network access link (resource). The distribution $Prob(r)$ gives the probability P that the actual delay D encountered by a request is smaller than r times its worst-case delay D^{wc} where $0 \leq r \leq 1$. Figure 4 shows that most requests experience less than 1/4th of their worst-case delays. In general, given a subtask that requires a latency bound of D with a violation probability bound of P, the following relationship holds from the definition of CDF: $D = D^{wc} \times Prob^{-1}(1-P)$. The nature of the worst-case delay term D^{wc} depends upon the nature of the specific local resource scheduler and the amount of resource bandwidth ρ reserved by the subtask. For instance, if the resource is scheduled using the Weighted Fair Queuing scheduler [43] then $D^{wc} = \frac{\sigma + L_{max}}{\rho} + \frac{L_{max}}{C}$, where σ is the maximum request burst size, L_{max} is the maximum request size and C is the total resource capacity. Hence, given the measured CDF $Prob(r)$, we can derive the required resource reservation ρ that satisfies the application latency requirements (D, P).

Our earlier work [19, 20] has successfully demonstrated techniques to exploit statistical multiplexing in the context of individual network and storage resource allocations. Figure 5

shows one example of the effectiveness of our approach using *delay distribution measurements* (DDM) in differentiating between the QoS requirements of multiple network flows that share a link and have different tolerance levels to delay bound violations. DDM has been shown to improve the resource usage efficiency of network and storage resources by up to a factor of 3 in the presence of statistical QoS constraints.

While statistical multiplexing in the context of single resource in isolation has been studied extensively [28, 29, 47, 7, 27, 6, 26, 45, 8, 56, 59], there has been little work in the direction of exploiting **statistical multiplexing across multiple heterogeneous resources**. Consider a multi-resource DRE task, as in Figure 1, which requires an E2E delay bound of D with a violation probability bound of P. *How does one partition the E2E statistical QoS requirement (D,P) into individual subtasks requirements* (D_i, P_i) *such that resource loads can be balanced and the number of admitted DRE tasks can be increased as much as possible?* We outline a few possible solutions below.

Simple Partitioning Problem: A straightforward approach is to find a partition such that $\sum_i^n D_i \leq D$ and $\prod_i^n (1 - P_i) \geq (1 - P)$ for a set of subtasks consisting an E2E DRE task. We are currently investigating an iterative algorithm for this version of the partitioning problem, which is similar in structure to the basic delay partitioning algorithms discussed in Section 2. In every iteration, we can first estimate a delay partition assuming a fixed probability partition and then find a new probability partition using the fixed delay partition calculated in the previous step. This process continues till the delay and probability partitions obtained from two consecutive iterations are within a predefined threshold of each other.

General Partitioning Problem: Note that, in the above version of the partitioning problem, the condition on partitioning E2E violation probability is more conservative than necessary. In particular, it assumes that the entire DRE task can satisfy its E2E violation probability bound only if each subtask satisfies its local violation probability bounds. In contrast, while a subtask i could violate its local sub-deadline at one resource, its next subtask $i + 1$ in sequence could complete well ahead of its sub-deadline, thus making up for the lost time, and still meet the E2E deadline. Thus an open research problem is to model this general partitioning problem taking into account inter-dependencies among the current service loads at each resource, potentially yielding significant gains.

5. FEEDBACK CONTROL

Resource requirements of E2E tasks may dynamically vary in DRE applications, for example, due to varying image processing time or propagation delay. The relative importance of incoming data flows may change in time. For instance, certain data flows may actually capture enemy aircraft or traffic accidents, while the others do not deliver important data. In this case, the DRE application can increase the frequency of more important E2E tasks. Also, possible overbooking due to statistical multiplexing can overload the system. Given dynamic workloads, it is necessary to design *control theoretic techniques* to manage the miss ratio of E2E deadlines.

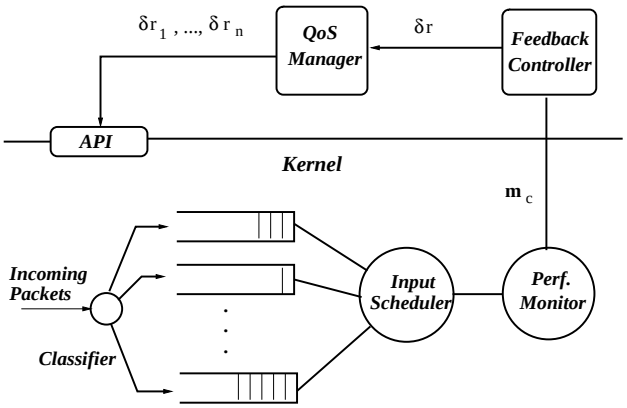


Figure 6: E2E Deadline Control Architecture

Figure 6 illustrates a possible architecture for feedback control at a C2 system. First, incoming packets are classified. Packet processing is scheduled at local resource schedulers via deadline partitioning and statistical multiplexing algorithms. Note that these algorithms may not be precise as workloads can vary dynamically, causing deadline misses under overload. The performance monitor measures the E2E miss ratio at every sampling instant, e.g., 1s, and informs the feedback controller of the current miss ratio m_c. At the k^{th} sampling period, the controller computes the error $e(k) = m_s - m_c(k)$ where m_s is the miss ratio set-point, e.g., 1%. Based on $e(k)$, it computes the control signal $\delta r(k)$. When $\delta r(k) < 0$, remote sensing nodes, e.g., UAVs or traffic monitoring nodes, are required to reduce the aggregate sensing and transmission rate by $|\delta r(k)|$. The QoS manager aware of application semantics or the instrumented DRE application itself divides $\delta r(k)$ among the incoming traffic flows to ensure that the most important flow, e.g., the flow tracking the largest number of targets, receives the highest rate. The resulting rates $\delta r_1, \delta r_2, ..., \delta r_n$ for n incoming sensor data flows are forwarded to a resource allocator, which accordingly reallocates resources between the n flows and sends the new required sensing rates to the remote sensor nodes, if necessary, to meet the QoS requirements. Below we describe some open research problems in developing an E2E timing control architecture.

Handling Input Queue Backlogs: A possible backlog in the incoming packet queues shown in Figure 6 can adversely affect the overall control performance. The system performance may not change immediately for a new control signal when there is a backlog to which the previous delay budget is already distributed [66]. Thus, there could be dead-time in control, which can result in a nonlinear system behavior [44]. One possible approach is redistributing the delay budget of the jobs already in the queues for the cost of performing resource re-allocation. In such situations, the resource allocation algorithms need to be lightweight so that one can sporadically re-allocate resources to improve real-time performance without affecting E2E tasks. We envision linearly approximating the system model via the relation between the aggregate packet arrival rate and E2E miss ratio. The miss ratio $m(k)$ at the k^{th} sampling period can be modeled by an n^{th} $(n \geq 1)$ order difference equation with unknown

coefficients $\{a_i, b_i | 1 \leq i \leq n\}$ initialized to zero:

$$m(k) = \sum_{i=1}^{n} a_i m(k-i) + \sum_{i=1}^{n} b_i r(k-i) \quad (2)$$

where $r(k-i)$ and $m(k-i)$ are the aggregate packet arrival rate and E2E deadline miss ratio at the $(k-i)^{th}$ sampling instant, respectively. Eq 2 denotes that the current miss ratio is dependent on the packet arrival rates and miss ratios measured at the previous sampling periods. System identification (SYSID) techniques [44, 42] can be applied to identify the model parameters in Eq 2. At the same time, there is a need to experimentally determine the specific order of Eq 2 producing high-accuracy SYSID results verified by standard techniques such as root mean square errors [44].

Self-Tuning Based Techniques: If the linear time invariant (LTI) control based on the system model and SYSID above is not applicable, e.g., due to a large backlog or wireless communication jitters in a noisy environment, it is also feasible to consider either a *self-tuning regulator* (STR) [53] or *self-tuning fuzzy controller* (STFC) [10]. A STR performs online SYSID to find the model coefficients in Eq 2 and tune the controller parameters online. In this way, the controller can tune itself if the system dynamics change in time. Fuzzy control is useful when the underlying system is hard to model mathematically. A linguistic rulebase instead of a mathematical system model can be designed for feedback control. For example, if the current overshoot is high and it is diverging from the set-point, the rulebase can generate a negative large control signal. On the other hand, if an overshoot shows a self-decaying pattern compared to the previous one, the rulebase generates a negative small signal for control stability. Especially, a STFC can tune itself, via a separate rulebase for self-tuning, to improve the performance considering the current system behavior. In real-time systems, the performance of several LTI models are compared [3], but other models are not compared extensively. Thus there is a need to undertake in-depth comparisons of the performance and complexity of the different control models, i.e., LTI and more advanced models, in the context of multi-resource DRE systems. As a result, control models achieving good performance with low complexity can be developed. Other key issues to be investigated along with control modeling include sampling period selection and stability analysis.

6. MURALS TESTBED

We are developing a prototype *MURALS* testbed on top of the TimeSys Linux [54] real-time operating system. The goal of *MURALS* is not to develop another comprehensive RTOS, but to demonstrate the effectiveness of core algorithms and techniques required for coordinated multiple resource allocation, with substantial rethinking of fundamental APIs and system architecture.

Figure 7 shows the architecture of the *MURALS* framework. Real-time applications access the *MURALS* API that interacts with the kernel subsystem using a set of *MURALS*-specific system calls. An admission controller makes admission decisions and performs deadline partitioning at the time a new DRE task registers itself. A performance monitor constantly tracks the timing behavior of E2E tasks to inform the feedback controller of the E2E miss ratio at every

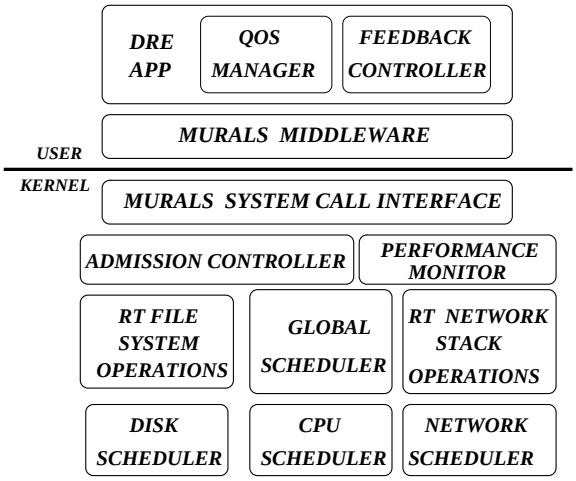


Figure 7: The architecture of the *MURALS* framework.

sampling instant.

The core of the *MURALS* kernel is organized around a two-level scheduling architecture. It consists of a *global scheduler*, which is aware of each application's task graph and estimated resource requirements, and a set of *local schedulers*, each of which correspond to a particular resource. The global scheduler ensures that a subtask's dependencies are all satisfied before it is executed by the the corresponding local resource scheduler. Local real-time schedulers manage individual resources and make scheduling decisions based on both the subtask deadlines and resource utilization efficiency. In this way, the global scheduler acts as a glue between local resource schedulers using its global knowledge of E2E deadlines and estimated resource requirements of each E2E task. Individual system resources employ rate-based real-time scheduling algorithms. CPU scheduler is a variant of Virtual Clock [65]. Disk scheduler is a rate-based variant of the our work-conserving DSSCAN algorithm [16]. Real-time network access is guaranteed by the wired and wireless variants of our Real-Time Ethernet (RETHER) protocol [50, 57]. These algorithms are modular and replaceable by any other rate-based schedulers.

The *MURALS* API library allows DRE application programmers to declaratively specify individual resource requirements and E2E timing constraints. The API facilitates creation and execution of a precedence graph, its component subtasks, and their dependencies. The library in turn informs the *MURALS* module in the kernel of the application requirements. A DRE application creates an initially empty precedence graph by first invoking the `Create_graph` function as `Graph_id = Create_graph(Deadline,Tolerance)` to specify that the precedence graph for an E2E task needs to be executed within the specified deadline and tolerance to delay violations. The application can possibly spawn many such precedence graphs that execute concurrently. The precedence graph can be populated by application programmers with individual subtasks such as read, write, or computation. For example, a write subtask can be added to the graph via `Register_write` call as `Subtask1 = Register_write( Graph_id, File_descriptor, Buffer, Size)` where the file descriptor can represent either regular files or network sock-

ets. Thus, the subtask can transmit data across the network or write data to the corresponding file. A read subtask can be registered in a similar fashion. Also, a computation subtask can be registered as `Subtask2 = Register_compute( Graph_id, Compute_func)`. Dependencies among subtasks can be specified in a pairwise manner. For example, `Depend(Subtask1, Subtask2, Graph_id)` specifies that Subtask2 can execute only after Subtask1 completes. The subtask dependencies are conveyed by the API to the *MURALS* kernel module verifying that the graph remains acyclic.

The registration of a subtask does not execute the subtask immediately. Rather, it informs the kernel *how* to execute this subtask *when* the kernel is asked to do so. This separation of *how* to execute a subtask from *when* to execute it is a departure from conventional operating system designs. It can provide greater flexibility in real-time application programming and resource scheduling. The DRE task can then be repeatedly executed by invoking `Exec_graph(Graph_id)`, which transparently executes all the subtasks according to precedence constraints within the specified E2E deadline.

7. RELATED WORK

A vast amount of research work has been conducted in real-time resource allocation and scheduling from different perspectives. We discuss the most relevant research results.

Multiple Resource Coordination: The body of work on real-time multi-resource coordination is relatively sparse, with none of the techniques supported in state-of-the-art RTOSs. The continuous media resource (CM-resource) model [4] is a framework meant for continuous media, e.g., digital audio and video, applications. Clients make resource reservations for the worst-case workload. The meta-scheduler coordinates with the CPU scheduler, network, and file-system to negotiate delay guarantees and the required buffer size on behalf of clients. Xu et. al. [63] present simulation results for a multi-resource reservation algorithm that determines the E2E QoS level for an application under resource availability constraints. The work on Cooperative Scheduling Server (CSS) [48] performs admission control for disk I/O requests by reserving both the raw disk bandwidth and CPU bandwidth required for processing disk requests. Timing constraints are partitioned into multiple stages and each of them is guaranteed to complete before its deadline on a particular resource. However, the deadline is partitioned based on a fixed slack sharing scheme rather than considering the resource utilization efficiency, possibly causing load imbalance across different resources. Q-RAM [31, 46, 13, 14] considers the problem of allocating multiple resources in one or more QoS dimensions to maximize the overall system utility. Spring Kernel [52] provides real-time support for multiprocessor and distributed systems using dynamic planning based scheduling. Real-time applications are written using Spring-C and resource requirements are specified using System Description Language. [33] models the effect of Linux network device driver on the schedulability analysis of real-time applications. Deadline partitioning has also been studied in the context of CPU resources alone for multiprocessor systems [51], for CPU and disk resource [17], and multi-link network paths [18].

Real-time Operating Systems: Numerous RTOSs support real-time scheduling for independent system resources, but lack a coordinated multi-resource allocation and scheduling mechanism to support E2E delay bounds. Some of these are Real-time Mach [55], Linux/RK [35], TimeSys Linux [54], RT-Linux [12], KURT Linux [30], QLinux [21], Eclipse/BSD [5], Rialto [25], and Nemesis [32].

Statistical Multiplexing: While statistical multiplexing has been extensively studied in relation to networks and to some extent in cluster-based services, relatively little attention has been paid towards statistical multiplexing effects in multi-resource real-time systems. Knightly and Shroff [28] provide an excellent overview of admission control approaches for link-level statistical QoS. Kurose [29] derived shared probabilistic bounds on delay and buffer occupancy using the concept of stochastic ordering for network nodes that use FIFO scheduling. Several analytical approaches [27, 9, 22, 47] have also considered multiplexing with shared buffers in single and multiple link settings. The notion of Effective Bandwidth, introduced by Kelly [26], is an important measure of bandwidth resource usage by flows relative to their peak and mean usage. A comparative study [8] of several MBAC algorithms [45, 23, 11, 15] under FIFO service concluded that none of them accurately achieve loss targets. Urgaonkar et. al. [56] perform resource overbooking via offline capacity profiling in shared hosting platforms for CPU and network resources. Vin et.al [59] explored statistical admission control algorithms for media servers. Vernick et.al [58] reported empirical measurements from implementations of statistical admission control algorithms in a fully operational disk-array video server.

Feedback Control: Most classical, open-loop real-time scheduling algorithms assume precise *a priori* knowledge of workloads. Feedback control has recently been applied to real-time performance management, because it does not require accurate system models for performance guarantees. A survey of feedback control for QoS management is provided in [2]. FC-UM [39, 40] provides the specified miss ratio and utilization in a single processor environment. DEUCON [61] manages the E2E CPU utilization in a multiprocessor environment via predictive model control. CAMRIT [60] applies control theoretic techniques to support the timeliness of image transmissions across one wireless link. All these approaches only consider a single resource, i.e., the CPU or transmission rate. HiDRA [49] manages both CPU and network bandwidth utilization via feedback control for target tracking. AQuoSA[36] proposes a control theoretic strategies to dynamically adapt CPU reservations in the Linux kernel. Control theoretic techniques have also been applied to manage the performance of a web server [1, 62, 38] and web cache [41]. However little research has been directed towards applying control theoretic techniques for real-time multi-resource applications.

8. CONCLUSION

Modern real-time systems increasingly consist of applications that need to use multiple heterogeneous resources to complete critical tasks within bounded end-to-end (E2E) delay. This argues for coordinated allocation and scheduling of multiple resources, such as the CPU, network, and disk, in real-time operating systems (RTOS). Unfortunately, state-

of-the-art RTOS do not support multi-resource coordination as a fundamental construct in the system design. This is presumably because the complex inter-resource interactions are poorly understood when it comes to resource allocation and runtime scheduling decisions at each resource in highly dynamic and distributed real-time systems. We examined some of the fundamental problems in this area and made the case for greater research effort into the development of theory and a runtime systems for coordinated allocation and scheduling of multiple resources in real-time operating systems. We discussed four open research problems and possible solution strategies in the areas of E2E deadline partitioning, explicit coordination across resource schedulers, statistical performance guarantees, and feedback control across multiple resources. We also described our current research efforts in the design and implementation of a *MURALS* testbed that addresses the above research problems. Even though we may not have covered all the major research issues in multi-resource allocation and scheduling, our paper has sought to identify and motivate investigation into some of the fundamental problems in this increasingly important area.

9. REFERENCES

[1] T. F. Abdelzaher and N. Bhatti. Adaptive Content Delivery for Web Server QoS. In *IWQoS*, 1999.

[2] T.F. Abdelzaher, J.A. Stankovic, C. Lu, R. Zhang, and Y. Lu. Feedback Performance Control in Software Services. *IEEE Control Systems*, 23(3), 2003.

[3] M. Amirijoo, J. Hansson, S. Gunnarsson, and S. H. Son. Experimental Evaluation of Linear Time-Invariant Models for Feedback Performance Control in Real-Time System. *Real-Time Systems*.

[4] D.P. Anderson. Metascheduling for continuous media. *ACM Trans. on Computer Systems*, 11(3):226–252, Aug. 1993.

[5] J. Blanquer, J. Bruno, E. Gabber, M. Mcshea, B. Ozden, and A. Silberschatz. Resource management for QoS in Eclipse/BSD. In *Proc. of FreeBSD'99 Conf., Berkeley, CA, USA*, Oct. 1999.

[6] R. Boorstyn, A. Burchard, J. Leibeherr, and C. Oottamakorn. Statistical service assurances for traffic scheduling algorithms. *IEEE JSAC*, 18(13):2651–2664, Dec. 2000.

[7] J.-Y Le Boudec and M. Vojnovic. Stochastic analysis of some expedited forwarding networks. In *IEEE Infocom'02*, June 2002.

[8] L. Breslau, S. Jamin, and S. Shenker. Comments on performance of measurement-based admission control algorithms. In *Proc. of IEEE INFOCOM*, 2000.

[9] R.L. Cruz. A calculus for network delay, Part I: Network elements in isolation. *IEEE Trans. on Information Theory*, 37(1):114–131, 1991.

[10] D. Driankov, H. Hellendoorn, and M. Reinfrank. *An Introduction to Fuzzy Control*. Springer, 1996.

[11] S. Floyd. *Comments on measurement-based admission control for controlled load services*. Tech. Rep., Lawrence Berkeley Labs, 1996.

[12] FSMLabs. Rtlinux free. http://www.rtlinuxfree.com/.

[13] Sourav Ghosh, Ragunathan Rajkumar, Jeff Hansen, and John Lehoczky. Scalable Resource Allocation for Multi-Processor QoS Optimization. In *ICDCS*, 2003.

[14] Sourav Ghosh, Ragunathan Rajkumar, Jeffery Hansen, and John Lehoczky. Integrated Resource Management and Scheduling with Multi-Resource Constraints. In *RTSS*, 2004.

[15] R. Gibbens and F. Kelly. Measurement-based connection admission control. In *Proc. of 15th Intl. Teletraffic Conference*, June 1997.

[16] K. Gopalan and T. Chiueh. *Real-time Disk Scheduling Using Deadline Sensitive Scan*. Technical Report ECSL-TR-92, Experimental Computer Systems Lab, Stony Brook University, January 2001.

[17] K. Gopalan and T. Chiueh. Multi-resource allocation and scheduling for periodic soft real-time applications. In *Multimedia Computing and Networking*, Jan. 2002.

[18] K. Gopalan, T. Chiueh, and Y.J. Lin. Delay budget partitioning to maximize network resource usage efficiency. In *Proc. IEEE INFOCOM'04, Hong Kong, China*, March 2004.

[19] K. Gopalan, T. Chiueh, and Y.J. Lin. Probabilistic delay guarantees using delay distribution measurements. In *Proc. of ACM Multimedia 2004, New York, NY*, October 2004.

[20] K. Gopalan, L. Huang, G. Peng, T. Chiueh, and Y.J. Lin. Statistical admission control using delay distribution measurements. *ACM Trans. on Multimedia Computing, Commn, and Apps*, 2(4), Nov. 2006.

[21] P. Goyal, X. Guo, and H. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *SOSP*, Oct. 1996.

[22] F. M. Guillemin, N. Likhanov, R. R. Mazumdar, and C. Rosenberg. Extremal traffic and bounds for the mean delay of multiplexed regulated traffic streams. In *IEEE INFOCOM'02*, June 2002.

[23] S. Jamin, P. Danzig, S. Shenker, and L. Zhang. A measurement-based admission control algorithm for integrated services packet networks. *IEEE/ACM Transactions on Networking*, 5(1):56–70, Feb. 1997.

[24] R. Jejurikar and R. Gupta. Dynamic slack reclamation with procrastination scheduling in real-time embedded systems. In *Design Automation Conf.*, June 2005.

[25] M. B. Jones, D. Rosu, and M. Rosu. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Proc. of SOSP*, pages 198–211, Oct. 1997.

[26] F. Kelly. Notes on effective bandwidths. *In Stochastic Networks: Theory and Applications*, 4:141–168, 1996.

[27] G. Kesidis and T. Konstantopoulos. Worst-case performance of a buffer with independent shaped arrival processes. *IEEE Communication Letters*, 4(1):26–28, Jan. 2000.

[28] E.W. Knightly and N. B. Shroff. Admission control for statistical QoS. *IEEE Network*, 13(2):20–29, Mar 1999.

[29] J. Kurose. On computing per-session performance bounds in high-speed multi-hop computer networks. In *Proc. of ACM Sigmetrics'92*.

[30] KURT-Linux. http://www.ittc.ku.edu/kurt/.

[31] C. Lee, J.P. Lehoczky, D.P. Siewiorek, R. Rajkumar, and J.P. Hansen. A scalable solution to the multi-resource QoS problem. In *IEEE RTSS'99*.

[32] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham,

D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE JSAC*, 14(7):1280–1297, Sept. 1996.

[33] M. Lewandowski, M. Stanovich, T. Baker, K. Gopalan, and A. Wang. Modeling device driver effects in real-time schedulability analysis: Study of a network driver. In *RTAS, Bellevue, WA*, April 2007.

[34] C. Lin and S.A. Brandt. Improving soft real-time performance through better slack reclaiming. In *Proc. of Real-Time Systems Symposium (RTSS)*, 2005.

[35] LinuxRK. www.cs.cmu.edu/∼rajkumar/linux-rk.html.

[36] G. Lipari, L. Abeni, T. Cucinotta, L. Marzario, and L. Palopoli. Qos management through adaptive reservations. *Real-Time Systems*, 29, 2005.

[37] J. Loyall, R. Schantz, D. Corman, J. Paunicka, and S. Fernandez. A Distributed Real-Time Embedded Application for Surveillance, Detection, and Tracking of Time Critical Targets. In *IEEE RTAS*, 2005.

[38] C. Lu, Y. Lu, T.F. Abdelzaher, J.A. Stankovic, and S.H. Son. Feedback Control Architecture and Design Methodology for Service Delay Guarantees in Web Servers. *IEEE Transactions on Parallel and Distributed Systems*, 17(9), Sept. 2006.

[39] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Feedback Control Real-Time Scheduling: Framework, Modeling and Algorithms. *Real-Time Systems*, 23(1/2), May 2002.

[40] C. Lu, X. Wang, and C. Gill. Feedback Control Real-Time Scheduling in ORB Middleware. In *the 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2003.

[41] Ying Lu, Tarek F. Abdelzaher, and Avneesh Saxena. Design, Implementation, and Evaluation of Differentiated Caching Services. *IEEE Transactions on Parallel and Distributed Systems*, 15(4), May 2004.

[42] J. Oh and K. D. Kang. An Approach for Real-Time Database Modeling and Performance Management. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2007.

[43] A.K. Parekh and R.G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE Trans. on Networking*, 1(3):344–357, Jun 1993.

[44] C. L. Phillips and H. T. Nagle. *Digital Control System Analysis and Design (3rd edition)*. Prentice Hall, 1995.

[45] J. Qiu and E. Knightly. Measurement-based admission control with aggregate traffic envelopes. *IEEE/ACM Transactions on Networking*, 9(2):199–210, April 2001.

[46] R. Rajkumar, C. Lee, J.P. Lehoczky, and D. P. Siewiorek. Practical solutions for QoS-based resource allocation. In *IEEE RTSS*, Dec. 1998.

[47] M. Reisslein, K.W. Ross, and S. Rajagopal. A framework for guaranteeing statistical QoS. *IEEE/ACM Transactions on Networking*, 10(1):27–42, February 2002.

[48] S. Saewong and R. Rajkumar. Cooperative scheduling of multiple resources. In *IEEE RTSS'99*, pages 90–101, Dec. 1999.

[49] N. Shankaran, X. Koutsoukos, D. Schmidt, Y. Xue, and C. Lu. Hierarchical Control of Multiple Resources in Distributed Real-time and Embedded Systems. In *Euromicro Conference on Real-Time Systems*, 2006.

[50] S. Sharma, K. Gopalan, N. Zhu, G. Peng, P. De, and T. Chiueh. Implementation Experiences of Bandwidth Guarantee on a Wireless LAN. In *Multimedia Computing and Networking*, Jan 2002.

[51] J. W. S.Liu. *Real-Time Systems*. Prentice Hall, 2000.

[52] J. Stankovic and K. Ramamritham. The Spring Kernel: A new paradigm for real-time systems. *IEEE Software*, 8(3), May 1991.

[53] K. J. Åström and B. Wittenmark. *Adaptive Control*. Prentice Hall, 1995.

[54] Timesys Inc. *http://www.timesys.com/*.

[55] H. Tokuda, T. Nakajima, and P. Rao. Real-time mach: Toward a predictable real-time system. In *USENIX Mach Workshop*, 1990.

[56] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. In *OSDI*, Dec. 2002.

[57] C. Venkatramani and T. Chiueh. Design, implementation, and evaluation of a software-driven real-time ethernet protocol. In *SIGCOMM'95*.

[58] M. Vernick, C. Venkatramani, and T. Chiueh. Adventures in building the stony brook video server. In *ACM Multimedia*, Nov 1996.

[59] H. M. Vin, P. Goyal, , and A. Goyal. A statistical admission control algorithm for multimedia servers. In *ACM Multimedia'94*.

[60] X. Wang, H.-M. Huang, V. Subramonian, C. Lu, and C. Gill. CAMRIT: Control-based Adaptive Middleware for Real-time Image Transmission. In *RTAS*, 2004.

[61] X. Wang, D. Jia, C. Lu, and X. Koutsoukos. DEUCON: Decentralized End-to-End Utilization Control for Distributed Real-Time Systems. *IEEE Trans. on Parallel and Distributed Systems*, 18(7):996–1009, July 2007.

[62] J. Wei and C.Z. Xu. eQoS: Provisioning of Client-Perceived End-to-End QoS Guarantees in Web Servers. *IEEE Trans. on Comp.*, 55(12), Dec. 2006.

[63] D. Xu, K. Nahrstedt, A. Viswanathan, and D. Wichadakul. QoS and contention-aware multi-resource reservation. In *HPDC*, Aug. 2000.

[64] H. Zhang. Service disciplines for guaranteed performance service in packet-switching networks. In *Proc. of IEEE*, volume 83(10), pages 1374–1396,, 1995.

[65] L. Zhang. Virtual Clock: A new traffic control algorithm for packet switching networks. In *Proc. of ACM SIGCOMM'90, Philadelphia, PA, USA*, pages 19–29, Sept. 1990.

[66] R. Zhang, S. Parekh, Y. Diao, M. Surendra, T. Abdelzaher, and J. Stankovic. Control of Weighted Fair Queueing: Modeling, Implementation, and Experiences. In *Integrated Network Mgmt*, 2005.

Accurate Run-Time Prediction of Performance Degradation under Frequency Scaling

David C. Snowdon
NICTA*
University of New South Wales
Sydney, Australia

Godfrey Van Der Linden
NICTA
University of New South Wales
Sydney, Australia

Stefan M. Petters
NICTA
University of New South Wales
Sydney, Australia

Gernot Heiser
NICTA
University of New South Wales
Open Kernel Labs
Sydney, Australia

ABSTRACT

Dynamic voltage and frequency scaling is employed to minimise energy consumption in mobile devices. The energy required to execute a piece of software is highly depedent on its execution time, and devices are typically subject to timeliness or quality-of-service constraints. For both these reasons, the performance at a proposed frequency setpoint must be accurately estimated. The frequently-made assumption that performance scales linearly with core frequency has shown to be incorrect, and better performance models are required which take into account the effects, and frequency setting, of the memory architecture. This paper presents a methodology, based on off-line hardware characterisation and run-time workload characterisation, for the generation of an execution time model. Its evaluation shows that it provides a highly accurate (to within 2% on average) prediction of performance at arbitrary frequency settings and that the models can be used to implement operating-system level dynamic voltage and frequency scaling schemes for embedded systems.

1. INTRODUCTION

Energy consumption is an increasingly important factor in the design of computing systems. This is particularly true in embedded systems, where a lower energy consumption improves battery life and reduces size and cost, and has a significant impact on the commercial viability of a product.

Dynamic voltage and frequency scaling (DVFS) is a technique for reducing a circuit's energy consumption by modifying clock frequencies. Reducing frequency results in a lower power consumption and increased software execution time. It generally allows a circuit's supply voltage to be reduced, leading to quadratic savings.

While reducing the frequency to a particular circuit can improve its energy efficiency, other circuits may use more energy as a result of the longer execution time. Therefore the slowest frequency is not necessarily energy-optimal [13], and the energy-optimal frequency can only be chosen via knowledge of the expected execution time.

*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.

For example, in a totally memory-bound system, a reduction in CPU frequency will not result in an increase in execution time as the CPU is constantly stalled waiting on the memory bus (which is unaffected by CPU frequency). The reverse is also true: CPU bound applications' execution time will not be reduced by an increased bus or memory frequency, but the overall energy consumption will increase due to the higher bus or memory idle power. Furthermore, the dependence of the total execution time on the frequency is specific to the workload. Figure 1 shows the normalised execution cycles for two applications running at various CPU, bus and memory frequency combinations on an Xscale based processor (see Section 4 for further details). The difference between CPU-bound and memory-bound tasks is striking. Knowledge of the performance effects of frequency scaling is essential for choosing an energy-optimal setpoint.

The policy which selects when to switch frequency, and which frequency to switch to, is generally selected by the operating system.

Estimating and predicting the runtime of a piece of software is an important component in an effective power management system. The energy required for a task is heavily dependent on time ($E = Pt$). While the CPU's power consumption will be reduced by frequency scaling, core frequency is unlikely to have an effect on the power for the rest of the system. The energy required by components other than the CPU will be proportional to the overall execution time, leading to a complex relationship between core frequency and overall energy use. This is further complicated by the adjustment of memory, bus and IO interface clock frequencies.

In addition, accurate estimation of the execution time of a task is essential for meeting *real-time* (RT) deadlines and *quality-of-service* (QoS) requirements while employing DVFS.

A further factor complicating execution-time estimation is the increasingly dynamic nature of embedded systems. The applications themselves, the nature of their input data and stimuli, and the environment in which they are run are dynamic. It is therefore not practical to characterise the applications' behaviour a-priori, and any estimation must be performed at run-time.

This paper presents a methodology and mechanism for the accurate

run-time estimation of the performance of a given piece of software at an arbitrary frequency setpoint. Our specific contributions are: (i) a model providing accurate estimates of the runtime of a workload at an arbitrary frequency; (ii) a sound methodology for generating an execution-time model from *performance monitoring counter* (PMC) measurements; (iii) a sound methodology for selecting the optimum PMCs; (iv) an efficient algorithm for the calculation of the performance at any frequency setpoint; and (v) an evaluation of our approach using an extensive and representative benchmark suite.

We first summarise related work in Section 2. Section 3 describes our model for the execution time of an application, before detailing the parameter selection methodology, model generation and finally discussing runtime performance prediction. We describe our evalutation environment in Section 4 and present the results we have obtained in Section 5 before concluding with a summary and an outlook into our future work.

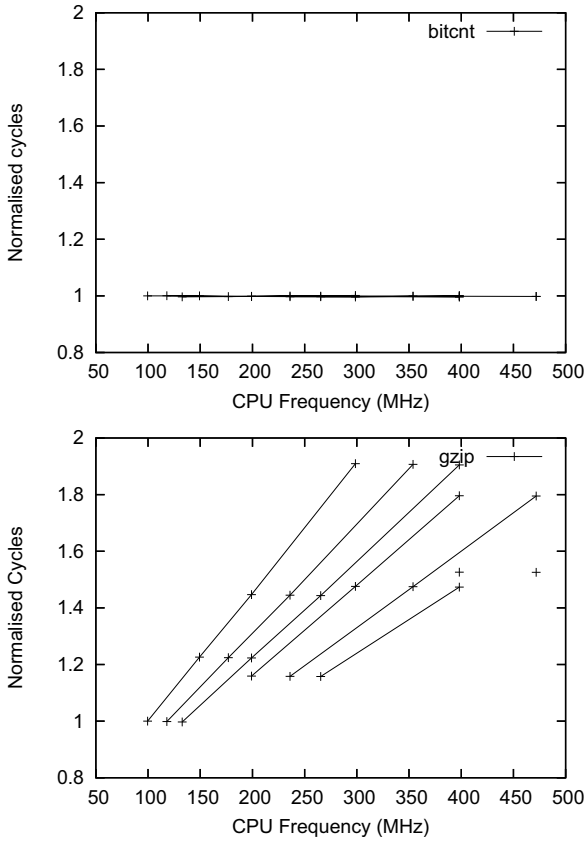


Figure 1: Execution cycles vs. CPU frequency, normalised to the slowest frequency cycles, for `bitcnt` and `gzip`, grouped by constant bus/memory frequency

2. RELATED WORK

The performance benefits of DVFS has been an active area of research ever since the pioneering work of Weiser et al. [15]. The work related to operating-system level scheduling can be divided into two broad categories based on the OS's assumed a-priori knowledge.

Real-time systems, which are required to deliver results by a deadline, require knowledge of the system timing, frequently in the form of *worst case execution times* (WCET). Previous work explored the potential for CPU frequency scaling without missing real-time deadlines [1,12]. This was extended to use the CPU's memory stall rate in a feedback loop with the scheduler [9,11].

Off-line techniques [5,17] use a detailed static analysis of a workload by the compiler. Other approaches include an *a priori* characterisation of a workload by running it at two different frequencies, in order to derive a slowdown relation [14]. These off-line results are then used by a DVFS-aware scheduler to scale the processor frequency.

Systems where no a-priori characterisation is performed generally aim to get the best energy efficiency for a given performance impact [7,8]. Such early work was typically based on the incorrect assumption that performance was proportional to CPU frequency. The non-linear dependency has since been the subject of considerable investigation.

Most of this research uses PMCs as a guide to predicting the likely performance impact of a frequency change. Process cruise control [16] used instructions, memory accesses and cycles to index a precomputed table of frequency settings which led to a constant performance impact. Other research groups have investigated a more flexible technique, using on-line regression to calculate the ratio of off-chip (CPU frequency independent) to on-chip cycles [2,4], however the computational overheads and response time are only cursorily discussed and there is reason to believe that they are substantial (i.e. the evaluation of a regression requires significant CPU time).

A theoretical model of a classification system between memory-bound and CPU-bound applications [18] assumes the availability of a large number of concurrently-usable performance counters. Limitations of this work include a lack of a detailed justification of performance-counter selection, insufficient evaluation with only a small number of benchmarks, and lack of statistical rigour (the evaluation is performed with the same data that is used to obtain the parameters of the model).

The models and methodology in this paper represent a significant advance over the above-outlined work. Software is characterised at run-time and our technique can therefore be used for arbitrary workloads with dynamic input data (in contrast with off-line techniques [5,14,17]). Compared with the previous state of the art [2,4], our model and methodology can be applied on arbitrary platforms, we do not require a run-time regression (resulting in a low overhead), we present our method of selecting the performance counters best suited to the task of performance estimation, and an evaluation has been performed with a much more extensive range of benchmarks.

3. APPROACH

The objective of this work is to predict the runtime of a workload at one frequency, given measurements at another frequency. This execution time model can then be used for making true energy vs. performance frequency scaling decisions as discussed in Section 1.

3.1 Execution Time Model

While the CPU core is a major contributor to a system's power consumption, other subsystems, such as memory and I/O, are also significant and can in some cases even dominate the CPU. Moreover,

such contributions generally are independent of the core frequency.

For example, the time the CPU stalls while waiting on main memory depends on the bus and memory clocks, not the core clock. On a processor with a single issue, in-order pipeline and a simple cache architecture (the typical scenario in embedded systems) the CPU is always stalled during memory activity. In this paper, we focus on this class of system. The effects of a superscalar architecture are expected to be small, but will be subject to future research.

For this class of system, we have an overall execution time T which is a function of the various clock rates f_x, used in the system:

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

The coefficients $C_{cpu}, C_{bus}, C_{mem}, C_{io} \dots$ depend on the instruction stream of the actual workload. The task of execution-time estimation comes down to obtaining good estimates for those parameters at run-time, without *a priori* analysis of the particular workload. I/O effects are beyond the scope of the paper, so we will, from here on, focus solely on the CPU and memory subsystem.

3.2 Application characterisation

While the coefficients C_x depend on the workload, they represent the total number of cycles used for particular actions (eg. CPU-only instructions or memory reads); each is the product of the number of such actions and the cycle cost of such an action. The former is a characteristic of the workload, the latter of the system architecture.

The fundamental idea behind our approach is to perform an off-line characterisation of the architectural parameters, and use run-time measurements, using performance monitoring counters (PMCs), for workload characterisation. Specifically, we postulate that each coefficient can be represented by some linear combination of PMC readings:

$$C_{bus} = \alpha_1 PMC_1 + \alpha_2 PMC_2 + \dots \quad (2)$$

$$C_{mem} = \beta_1 PMC_1 + \beta_2 PMC_2 + \dots \quad (3)$$

The accuracy of the model will depend on the architecture and the suitability of the PMCs. The architecture-specific coefficients α_x, β_x, ...are properties of the hardware platform and can be determined once, by evaluating a suitable set of benchmarks representing the range of different workloads. Linear regression on Equation 2 and Equation 3 will establish the values of the coefficients and allow selecting the most suitable PMCs. This is important, as the hardware typically supports the concurrent use of only a small number of PMCs, and the correct choice is not obvious as will be shown.

The total number of CPU cycles, C_{tot}, can be directly obtained from the CPU's cycle counter. It is the product of total execution time, T, and core frequency, f_{cpu}, allowing us to rewrite Equation 1 as

$$C_{cpu} = C_{tot} - \frac{f_{cpu}}{f_{bus}} C_{bus} - \frac{f_{cpu}}{f_{mem}} C_{mem} \quad (4)$$

Hence, we only need to determine C_{bus} and C_{mem} from PMCs.

3.3 Performance prediction

Being able to estimate a workload's C_x values from PMC readings at a particular setpoint (characterised by a particular combination of clock frequencies, f_x) it is possible to predict the performance of the same workload at a different setpoint with frequencies f'_x:

$$\begin{aligned} C'_{tot} = C_{tot} & \\ - \frac{f_{cpu}}{f_{bus}} C_{bus} - \frac{f_{cpu}}{f_{mem}} C_{mem} & \\ + \frac{f'_{cpu}}{f'_{bus}} C_{bus} + \frac{f'_{cpu}}{f'_{mem}} C_{mem} & \end{aligned} \quad (5)$$

We define the performance, s, as the execution time at a particular setpoint normalised to execution time at maximum frequency setpoint, f_x^{max}:

$$s = \frac{t^{max}}{t'} = \frac{f'_{cpu}}{f_{cpu}^{max}} * \frac{C_{tot}^{max}}{C'_{tot}} \quad (6)$$

For the present setpoint, $f'_{cpu} = f_{cpu}$ and $C'_{tot} = C_{tot}$, the latter being the performance counter reading. Hence, the performance at the current frequency settings is a linear equation of performance counter and frequency cross terms. This allows a single regression to be applied across all workloads to calculate α_x and β_x, given a pre-calculated s avoiding the intermediate step of C_x.

$$s_{cur} = \frac{f_{cpu}}{f_{cpu}^{max}} * \frac{C_{tot}^{max}}{C_{tot}} \quad (7)$$

Similarly, we can calculate performance relative to the current setpoint using C_{tot} in place of C_{tot}^{max}.

4. EVALUATION

Our model and methodology were evaluated on a typical embedded systems platform (Section 4.1) using a number of benchmarks (Section 4.2). The model was used to implement an on-line estimation system (Section 4.3).

4.1 Platform

The hardware platform used in all experiments was PLEB 2, a single board computer based on an Intel PXA255 processor [6]. The PXA255 is based on an Xscale core, with split L1 caches and TLBs, write and fill buffers. The data cache supports a write-back policy. PLEB2 integrates 64MB SDRAM and 8MB Flash memory. The core voltage remained constant at 1.5V.

Only specific combinations of f_{cpu}, f_{bus} and f_{mem} can be generated by the processor. For our experiments, we use 22 setpoints which are detailed in Table 1. Note that most of these setpoints are outside the manufacturer's recommended limits, but were found to work reliably and used in order to obtain more data.

The PXA255 provides three performance counters – a cycle counter and two configurable performance counters. The configurable counters can each count any one of 14 events [6] (outlined in Table 2).

We conducted all experiments on Linux 2.4.19, having written kernel modules and modifications to support per-process performance

	f_{cpu} (MHz)	f_{bus} (MHz)	f_{mem} (MHz)
1	99.531	49.766	99.531
2	117.964	58.981	117.964
3	132.71	66.354	132.71
4	149.299	49.766	99.531
5	176.946	58.981	117.964
6	199.064	49.766	99.531
7	199.064	66.354	132.71
8	199.064	99.531	99.531
9	235.929	58.981	117.964
10	235.929	117.964	117.964
11	265.420	66.354	132.710
12	265.420	132.710	132.710
13	298.598	49.766	99.531
14	298.598	99.531	99.531
15	353.894	58.981	117.964
16	353.894	117.964	117.964
17	398.131	66.354	132.71
18	398.131	99.531	99.531
19	398.131	132.71	132.71
20	398.131	199.064	99.531
21	471.858	117.964	117.964
22	471.858	235.929	117.964

Table 1: PXA255 frequency setpoints

counter reading. The PMCs were read and accumulated after each scheduler invocation.

When executing benchmarks, the only other runnable thread was kupdated, which flushes the file system buffers. The network device was disabled during benchmark execution. The timer tick was not disabled, which will be a small source of error in the measurements since the number of timer ticks which occur during a benchmark run is dependent on its real-world execution time.

4.2 Benchmarks

Sound experimental methodology requires that the workloads used for evaluation must be different from those used for calibration. A further requirement on the benchmarks is that the total amount of work is the same in each run, independent of the frequency settings.

PMC	Description
0x0	ICache miss
0x1	ICache stall cycles
0x2	Data dependency stalls
0x3	ITLB miss
0x4	DTLB miss
0x5	Branch instruction executed
0x6	Branch mispredicted
0x7	Instruction executed
0x8	DCache buffer stall cycles
0x9	DCache buffer stall
0xa	DCache access
0xb	DCache miss
0xc	DCache write-back
0xd	Software changed the PC

Table 2: PXA255 performance counter events

For calibration we used a total of 37 benchmarks. Most are from the MiBench suite [3], a set of real-world applications which are representative of the tasks found in different types of embedded systems. Several benchmarks (`djpeg`, `susan_corners`, `susan_edges`, `tiff2bw`) were removed as their total execution time was too short to be useful (less than 0.25s at maximum frequencies). Two short-running ones (`adpcm` and `stringsearch`) were modified to iterate a number of times in order to extend the overall run time. Two others (`sphinx`, `pgp`) were removed as their amount of work differs between runs under identical circumstances.

We added further benchmarks to the calibration set. Four (`gzip`, `mpg123`, `vision` and celp32c) have been previously described [13]. We also added three synthetic benchmarks to cover extreme behaviour: `cpubound` executes an unrolled loop of NOP instructions entirely in cache. `membound` and `readbound` execute an unrolled loop of out-of-cache writes and reads respectively.

For validation we used SPEC CINT95 benchmarks. `vortex` was excluded because of memory constraints, `go` and `m88ksim` due to runtime errors. The "test" dataset was used and the input data size for `compress` was reduced to 420000 bytes to reduce overall execution time. This leaves 5 benchmarks used for validation.

All benchmarks were compiled or assembled using gcc 3.4.4 with softfloat. The linux kernel and ramdisk was compiled using gcc 3.3.2.

4.3 Implementation

We used the approach presented in Section 3 to estimate, while executing at a particular frequency setpoint, the performance that would be achieved at the maximum-frequency setpoint. This can then be compared to direct measurements of an execution at f_x^{max}.

For evaluation purposes, we also used the techniques to aim for a particular pre-determined performance, which can then also be compared to the actual performance, obtained by measuring the overall execution time and comparing to the execution time at f_x^{max}. While this technique is unlikely to choose an energy-optimal setpoint, it does demonstrate the ability of the system to predict the performance of a benchmark at run-time, which we have discussed as being crucial to energy-optimality.

The approach is based on the well-established model of temporal locality which underlies many operating-system policies. In our case this means that we assume that the behaviour of a particular task does in most cases not change significantly between subsequent time slices. At the end of each time slice, the OS collects the PMC readings and *estimates* the slowdown at the present setpoint using our model. When the task is next scheduled, the slowdown estimate is compared to the target slowdown, and the frequency setting adjusted if necessary.

Note that, without a system-level energy model, it is impossible to select an energy-optimal frequency. Furthermore, the benefit of slowdown will be different for each application, as well as for different phases of a given application. We leave the selection of an energy-optimal frequency to future work and simply aim to accurately predict performance.

A key advantage of our approach is the lightweight nature of the estimation calculations. The XScale is typical for embedded pro-

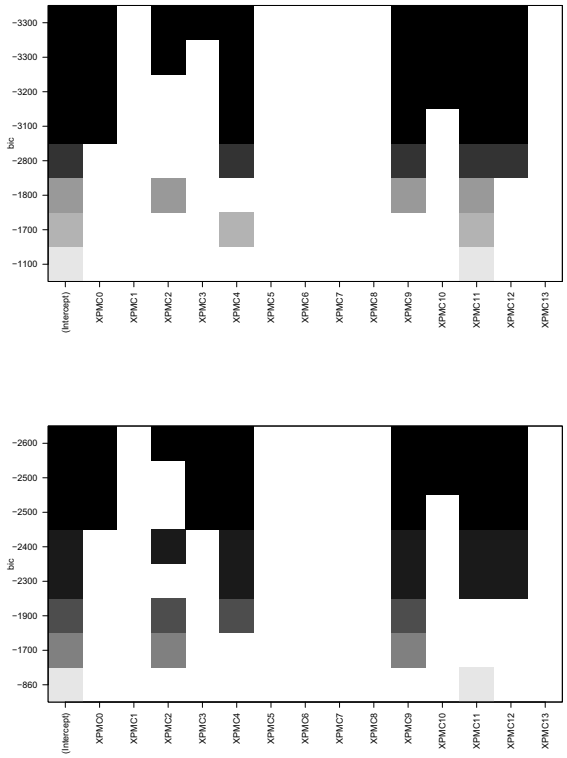


Figure 2: Parameter selection for C_{bus} and C_{mem} models

cessors in that it lacks an FPU and a hardware divide instruction. We therefore use only fixed-point arithmetic and avoid divisions.

5. RESULTS

5.1 Time vs. Frequency validation

Each benchmark in the calibration suite was run at each of the 22 setpoints. Equation 1 was fit to the data for each of the benchmarks using least-squares linear regression. The fit was extremely good, the value of $1 - R$, which indicates effects in the data that are not explained by the model, ranging between 5×10^{-4} and 3×10^{-8}. This is a strong indication that the model can accurately account for the architectural features of this class of processor. Furthermore, the intercepts were negligible, indicating that, for these benchmarks, in this system, the execution time depends solely on these frequencies.

5.2 Performance counter selection

We investigated the best choices of performance counters when only a small number of them can be used concurrently (as on most hardware, including ours). The models were formulated by equating C_{bus} and C_{mem} with all possible linear combinations of the available performance counters, as well as several potentially relevant cross terms.

Each model of each size was compared using a criterion function. i.e. every possible combination of performance counters was used to predict each of C_{bus} and C_{mem}. The best n-parameter model was selected using the BIC criterion function (a measure of the model's predictive capability), although R^2 would rank the models in the same way. (e.g. the n-parameter model with the highest BIC or R^2 is selected as the best). This procedure was performed using the `regsubsets` command in `R` [10]. In this way we determine the best performance counters for performance prediction of these calibration workloads. The results of the parameter selection for C_{bus} and C_{mem} are shown in Figure 2. The figures show which parameters are selected for each of the n-parameter models (each row represents a model with one more parameter than the row below). The differences in the graphs are likely due to noise in the data, insufficient variance in the benchmarks. Our future work has included a unified approach to the parameter selection.

We observe that, for this benchmark suite, the best single parameter model uses data cache misses (PMC11), the best dual parameter model also uses data TLB misses (PMC4). The best three parameter model uses PMC11, data cache buffer stalls (PMC9) and data dependency stall cycles. The best four parameter model uses PMC11, PMC4, PMC9 and data cache write-backs.

The results also show that the model does not improve significantly beyond four PMCs, and two PMCs perform almost as well as three (indicating strong correlations between cache and TLB misses). The PXA255 only supports two simultaneous measurements (in addition to the cycle counter), so our on-line prediction system in this platform is based on data TLB misses and data cache misses.

Importantly, unlike previous work [2, 4], since our model does not implicitly require the number of instructions, the parameter selection is free to choose any two events.

5.3 Slowdown prediction

While the actual parameters selected by this procedure depend on the benchmarks used for calibration, the BIC values indicate that this should not have a dramatic effect on the overall results. This can be verified by validation runs using independent benchmarks (our SPECINT subset).

An offline evaluation (i.e., using end-to-end data obtained running the validation suite over several executions) yields an average error of 1.7% and maximum error of 7% for the two parameter model. For comparison, the same data yields an average error of 10% and a maximum of 38% if the estimate is based only on the CPU frequency ("naive model"). Figure 3 shows the errors in the naive model, and Figure 3 for the 2-parameter model. The improvement over the naive model is obvious.

5.4 Frequency scaling error

We then ran each of the validation benchmarks aiming for 17 predetermined performance values ranging from 20 and 100%, each time recording actual and estimated run time at f_x^{max}. As the frequency settings are not continuous, the system cannot normally chose a setting that is estimated to produce exactly the target performance. Instead, the frequency selection policy simply chooses the setpoint which gives the closest approximation to the desired performance; the actual performance and desired performance will therefore differ, even if our estimates were totally correct. To account for this fact, we present the values again as the error in the estimated performance (estimated minus actual) against the estimated performance, Figure 5. These are on-line estimation errors: the performance is calculated for each time slice.

The maximum error observed was 7%, the average was 1.9%.

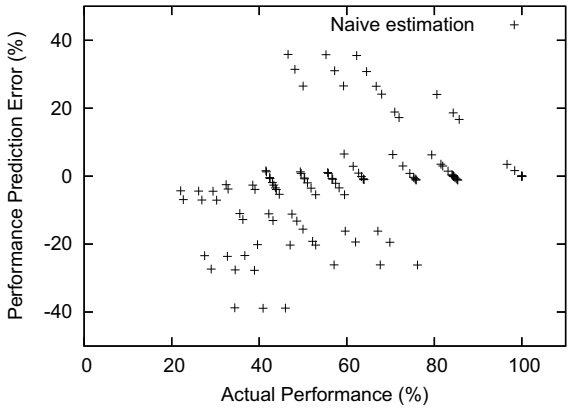


Figure 3: Naive model estimated performance error vs. actual performance

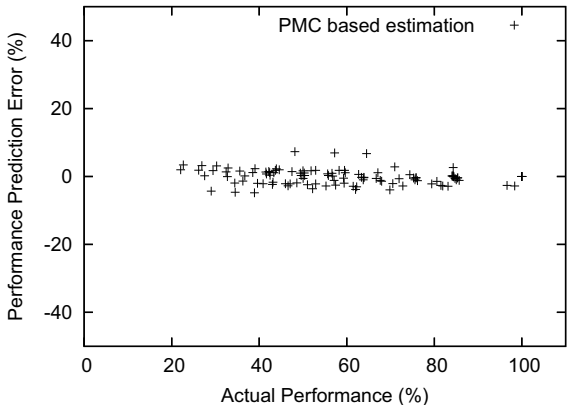


Figure 4: 2 parameter PMC-based model estimated performance error vs. actual performance

These compare favorably with the most accurate published performance estimations (4–6% [2]), and are consistently more accurate than those presented in other work, despite having been tested with a much larger variety of workloads. In addition, previous work generally used the same benchmarks for calibration and validation. Thus the errors observed in most previous work are not indicative of their models' predictive capability.

5.5 Frequency scaling overheads

The cost of the frequency selection calculations were measured to be 5000—7750 cycles. This averages $24\mu s$, which compares favourably to Choi's $100\mu s$ [2]. That work requires an on-line regression calculation, yet does not consider multiple memory frequencies.

6. CONCLUSIONS

This paper has first motivated, and then presented a general and sound model of execution time under frequency scaling. It is based on an off-line characterisation of the hardware platform, combined with on-line evaluation of application characteristics using performance counters. The model has been implemented and validated on a processor typical for use in high-end mobile systems.

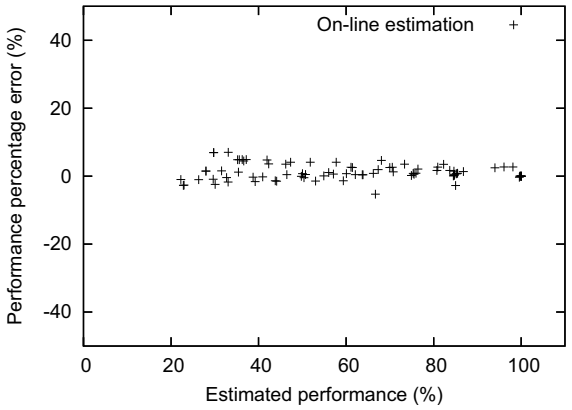


Figure 5: On-line estimated performance error vs. estimated performance

The model, once developed on a set of representative benchmarks, has demonstrated an excellent ability to predict the performance of new applications. The on-line evaluation implies that the model can quickly adjust to changes in application behaviour. The approach is general in the sense that it should be readily portable to different processor platforms providing basic performance monitoring hardware. The model performed well with only two performance counters, without the need for time-multiplexing.

In addition, this work has taken a rigorous approach to the model evaluation, with two large, disjoint published sets of benchmarks used for calibration and validation. The system was tested with an order of magnitude more workloads than comparable work.

The model has clear applications as part of an energy-saving framework, which could enable an accurate trade between performance and energy. Our subsequent work will show the necessity of a performance model when building an accurate model for the prediction of energy consumption under frequency scaling.

In the future we plan to validate the model's generality by deploying it on other platforms. Another obvious next step is an evaluation in a multi-processing context. Performance loss related to IO devices, and the effect of interupts and DMA should be examined.

7. REFERENCES

[1] H. Aydin, V. Devadas, and D. Zhu. System-level energy management for periodic real-time tasks. *Proceedings of the 27th IEEE Real-Time Systems Symposium*, 0:313–322, 2006.

[2] 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. *IEEE Transactions on CAD ICAS*, 24(1):18–28, Jan. 2005.

[3] M. R. Guthaus, J. S. Reingenberg, D. Ernst, T. M. Austing, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 4th IEEE Annual Workshop on Workload Characterization*, Dec. 2001.

[4] C.-H. Hsu and W. chun Feng. Effective dynamic voltage scaling through CPU-boundedness detection. In *Proceedings of the 2004 Workshop on Power-Aware Computer Systems*, pages 135–149, 2004.

[5] C.-H. Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. *SIGPLAN Not.*, 38(5):38–48, 2003.

[6] Intel Corporation. Intel PXA250 and PXA210 applications processors developers manual. http:

//www.intel.com/design/pca/products/pxa255/techdocs.htm, 2005.

[7] T. L. Martin and D. P. Siewiorek. Nonideal battery and main memory effects on cpu speed-setting for low power. *IEEE Trans. Very Large Scale Integr. Syst.*, 9(1):29–34, 2001.

[8] A. Miyoshi, C. Lefurgy, E. V. Hensbergen, R. Rajamony, and R. Rajkumar. Critical power slope: understanding the runtime effects of frequency scaling. In *Proceedings of the 16th International Conference on Supercomputing*, pages 35–44, New York, NY, USA, 2002. ACM Press.

[9] C. Poellabauer, L. Singleton, and K. Schwan. Feedback-based dynamic voltage and frequency scaling for memory-bound real-time applications. In *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium*, volume 00, pages 234–243, 2005.

[10] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2006. ISBN 3-900051-07-0.

[11] D. Rajan, R. Zuck, and C. Poellabauer. Workload-aware dual-speed dynamic voltage scaling. In *Proceedings of the 12th International Conference on Embedded and Real-Time Computing and Applications*, pages 251–256, 2006.

[12] K. Seth, A. Anantaraman, F. Mueller, and E. Rotenberg. FAST: Frequency-aware static timing analysis. *ACM Transactions on Embedded Computing Systems*, 5(1):200–224, 2006.

[13] D. C. Snowdon, S. Ruocco, and G. Heiser. Power Management and Dynamic Voltage Scaling: Myths and Facts. In *Proceedings of the 2005 Workshop on Power Aware Real-time Computing*, New Jersey, USA, Sept. 2005.

[14] V. Venkkatachalam, C. Probst, and M. Franz. A new way of estimating compute boundedness and its application to dynamic voltage scaling. *International Journal on Embedded Systems*, 1(1):64–74, 2006.

[15] M. Weiser, B. Welch, A. J. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, pages 13–23, 1994.

[16] A. Weissel and F. Bellosa. Process cruise control—event-driven clock scaling for dynamic power management. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, Grenoble, France, Oct. 8–11 2002.

[17] F. Xie, M. Martonosi, and S. Malik. Compile-time dynamic voltage scaling settings: Opportunities and limits. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 49–62, New York, NY, USA, 2003. ACM Press.

[18] F. Xie, M. Martonosi, and S. Malik. Efficient behavior-driven runtime dynamic voltage scaling policies. In *Proceedings of the 3rd International Conference on Hardware/Software Codesign and System Synthesis*, pages 105–110, 2005.

Run-time mechanisms for property preservation in high-integrity real-time systems*

Juan Zamorano
Universidad Politécnica de Madrid (UPM), Spain
jzamora@datsi.fi.upm.es

Juan A. de la Puente
Universidad Politécnica de Madrid (UPM), Spain
jpuente@dit.upm.es

Jérôme Hugues
GET-Télécom Paris – LTCI-UMR 5141 CNRS, France
hugues@infres.enst.fr

Tullio Vardanega
University of Padova, Italy
tullio.vardanega@math.unipd.it

ABSTRACT

Classical real-time kernels tend to leave to the application level the burden of policing those stipulations that the designer deems crucial to warrant the correct operation of the system. In fact, in the general case, there exist forms of reflective computing at application level that may be happy with that arrangement. Where "continuity of proof" and "preservation of properties" are central to the development paradigm instead permissive kernels are arguably inferior to proactive execution platforms which are capable of: (i) policing the critical stipulations; and (ii) preventing their violations at run time. In this short paper we illustrate some constructive principles of an execution environment that follows the latter paradigm.

1. INTRODUCTION

One distinctive objective pursued by the ASSERT project (cf. footnote 1) is to attain "preservation of properties" throughout the entire development process, from system modeling down to run-time execution. A crucial ingredient to attain this objective entails the design and implementation of an execution environment that exhibits two fundamental features:

1. to ensure that the run-time entities (e.g. threads, locks, queues, etc.) that are deployed to implement the system model do have *exactly* the same semantics as was assumed in the verification and validation of the system model

2. to ensure that the run-time attributes which decorate all elements of the system model are also attached, without semantic distortion, to the run-time entities that implement them and, when they designate *stipulations* (e.g. a given budget, whether in time for execution, in size for storing, or in bandwidth for communication) they be actively policed during execution so that no single violation of them may either occur or go unnoticed.

*This work has been funded in part by the Sixth Framework Programme of the European Commission under project FP6-IST-2004 004033 (ASSERT).

Meeting requirement 1 in a guaranteed manner is considerably facilitated if visibility of the kernel API is hidden away from the designer so that calls to it can only be issued at places strictly controlled by the development process. (In the context of Model-Driven Engineering, for example, those places would be determined by the model transformation logic as opposed to by manual programming, as it is still often the case in the development of high-integrity real-time systems.)

To satisfy requirement 2, instead, the execution platform that we have designed for use in ASSERT had to be rigidly inflexible in hosting, executing and actively policing the run-time behaviour of the allowable run-time entities. In order to underline its distinctive character, which sets it aside from classical real-time kernels (which tend to be permissive when it comes for the policing of stipulated run-time behaviour), we have chosen to name our execution platform: "Virtual Machine" (VM), with a connotation that intends to evoke the correct interpretation of an intended semantics and its active enforcement at run time.

The ASSERT VM concept exhibits a number of important characteristics:

a) it is a run-time environment that only hosts and supports "legal" entities, i.e., those that are explicitly retained as the target of the automated model transformation process which realizes the system specification (the equivalent of a PIM) in terms of correct-by-construction aggregates and interconnections of allowable run-time entities (the equivalent of a PSM);

b) it provides run-time services that aid those run-time entities to actively preserve their designated properties; mechanisms and services of interest allow for instance to:

- accurately measure the actual execution time that can be attributed to individual threads of control
- attach and replenish a monitored execution time budget to a thread, and then prompt an alarm when the thread exceeded its time budget
- segregate threads into distinct groups, attaching

a monitored budget to individual groups, to be handled in the same way as for threads

- enforce the minimum inter-arrival time stipulated for sporadic threads
- build fault containment regions around individual threads and groups thereof
- attain distribution and replication transparency in inter-thread communication.

c) it is bound to a compilation system that only produces executable code for "legal" entities and rejects the non-conforming ones; run-time checks provided by the virtual machine shall cover the extent of enforcement that cannot be exhaustively achieved at compile and link time

d) it realizes a concurrent computational model provably amenable to static analysis; the model must permit threads to interact with one another (directly, by some form of synchronization, and indirectly, by preemptive interference) in ways that do not incur non-determinism.

It is worth noting that the concept that underpins the VM arguably goes beyond the current state of the art (cf. e.g. [1]) in that it incorporates more than just overrun detection, but rather a whole range of features and mechanisms that actively "police" the continued compliance of the system behaviour at run time to its specification. In the remainder of this short paper we shall briefly discuss some of the most noteworthy features of the ASSERT VM in this respect.

2. DISTINGUISHING FEATURES OF THE VIRTUAL MACHINE

In order to comply with the required characteristics, the ASSERT VM semantics is based on the Ravenscar computational model [2], a concurrency model enabling predictable behaviour. Legal entities at the program code level are those accepted by an Ada 2005 compiler restricted by the Ada Ravenscar profile [9].

2.1 Activation of sporadic tasks

A foremost property to be preserved for any sporadic task is the minimum interval time that is stipulated to occur between any two successive activations (a.k.a. minimum interarrival time). This property is essential in ensuring the feasibility of response-time analysys, for its violation may result in unexpectedly high interference for lower-priority tasks [3].

The ASSERT system generation process enforces this property by using an Ada pattern based on a `delay until` statement, as shown in [7]. The semantics of this statement is supported by the VM using a delay queue, in a similar way as was done in the ORK kernel [11].

2.2 Execution-time monitoring

Execution-time monitoring is a means to prevent execution-time overruns from occurring. Overruns may be caused by erroneous estimation of the worst-case execution time (WCET) of a task, or by some misbehaviour resulting in the task executing for longer than stipulated. In either case

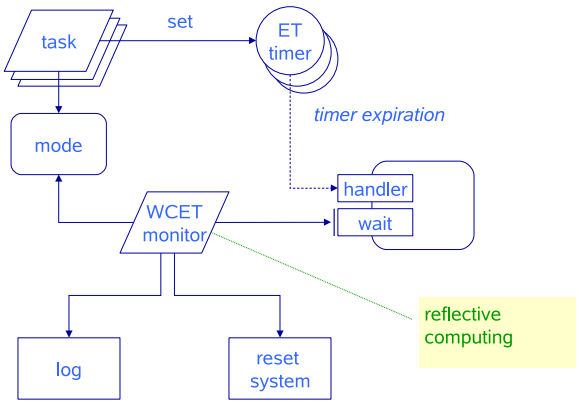


Figure 1: Execution-time monitoring.

appropriate corrective actions must be taken to preventing the offending task from jeopardizing the temporal behaviour of other tasks in the system.

Execution-time monitoring in the ASSERT VM is based on execution-time timers, a specific feature of the new Ada 2005 standard [9]. Each task has an execution-time clock which only advances while the task is actually executing. Based on it, an execution-time timer can be defined which can be set to a time interval or to an absolute value. If the timer expires, a protected procedure handler is executed in a way similar to an interrupt handler.

Execution-time timers are not currently allowed in the Ravenscar Profile [9]. However, in view of their strategic interest to the objectives of the project, the ASSERT VM implements a minimal extension to the Ravenscar Profile by including at most one statically declared timer per task. The code archetypes for execution-time monitoring (cf. listing 1 for an example) use one such timer to detect WCET overruns.

When an overrun is detected, a high-priority monitoring task is released to recover from the error (cf. figure 1). Various forms of corrective measures can be contemplated within the confines of the Ravenscar Profile: our current orientation is to force a mode change to the offending task, which will take effect at the next activation, assuming that the task will end its current violating execution without needing external intervention. (While this assumption may be over-optimistic in the general case, we reckon it is not in the case of ASSERT which pursues a zero-programming development paradigm and thus should be less exposed to erroneous application code.)

2.3 High-integrity distribution

Adding distribution features to High-Integrity (HI) systems such as those targeted by ASSERT requires first to categorize the possible causes of software failures that may stem from distribution, and then to define the set of preventive measures which may exclude them. We in particular selected three typical concerns of distribution middleware and took special care in addressing them in a HI setting by combining run-time mechanisms and modeling artifacts: 1) no dynamic memory allocators; 2) no dynamic skeleton dispatchers; 3) prevention of overruns on both the client and the server side.

Listing 1: WCET overrun detection.

```
My_Identity : aliased constant Task_Id := Periodic_Task'Identity;
WCET_Timer  : Ada.Execution_Time.Timers.Timer(My_Identity'Access);

task body Periodic_Task is
  Next_Activation : Ada.Real_Time.Time := Epoch;
begin
  loop
     WCET_Timer.Set_Handler
       (In_Time=>My_WCET_Budget,
        Handler=>My_Monitor.Overrun_Handler'Access);
    delay until Next_Activation;
    Do_Actual_Work;
    Next_Activation := Next_Activation + My_Period;
  end loop;
end Periodic_Task;
```

2.3.1 No dynamic memory allocators

Dynamic memory allocation typically occurs when the developer does not know beforehand how much memory it takes to perform the required actions. In a middleware setting, this situation usually arises when buffers must be allocated to: handle incoming requests; initialize connections; or store internal data to manage internal state information.

In fact, all of those situations and the relevant needs can be deduced from a careful analysis of the application model interfaces and the intended system topology. Further assistance of course comes from imposing suitable restrictions on those elements:

1. interfaces are restricted to only use types of bounded size so that the maximum size of buffers needed to support each remotely invokable function can be computed statically
2. prior knowledge about the application topology (i.e., direction and number of connections) permits to statically precompute the required tables of naming information and to store them at elaboration time at all places where they are needed.

By combining restrictions (1) and (2) one becomes able to statically allocate all required resources at compile time, thereby renouncing the need for run-time allocation.

2.3.2 No dynamic skeleton dispatchers

Skeleton dispatchers are required to map network messages onto the call to designated local procedures. To avoid incurring unpredictability in both time and space we infer from the system model the information required to allocate static arrays of dispatchers so that the incoming requests can be directly marshalled into one index in the store. It is worth noting that the latter step is performed in $O(1)$ time, in contrast with standard CORBA middleware, which requires a string-to-index mapping and thus incurs the overhead of hashing [8].

2.3.3 Preventing overruns

The prevention of overruns (which in a distributed context may result in denial of service situations) is a major safety problem. Situations of this kind happen when either a client sends too much data (the "babbling client"), or a server receives more requests than it is able to process. These situations are handled by transport-level primitives of the ASSERT VM middleware respectively by:

1. preventing the client to send more than a given N number of requests per time unit, for a grand total of k bytes in that time interval, using a simple timer and an a byte budget counter;
2. stopping the server from being too responsive to incoming requests, by disabling the monitoring of a I/O source, or by ignoring specific connections in case the device permits it;
3. ensuring that only authorized clients interact with the server. The server will silently ignore all requests coming from other clients.

Measures 1 and 2 are supported by the transport stack, which maintains a table of resource consumption and uses monitoring techniques similar to those discussed in section 2.2 for task execution time. The exact definition of what corrective actions should be performed in the face of a violation follows from thorough analysis of the application needs, with the intent of permitting nominal operation to continue undisturbed while also ensuring that violations are trapped and not permitted (to continue) to occur.

Measure 3 simply entails a run-time check to be performed into the name tables built for each node.

3. IMPLEMENTATION STATUS AND LESSONS LEARNED

An implementation of the ASSERT VM for the LEON2 computer architecture [5] has been almost completely achieved

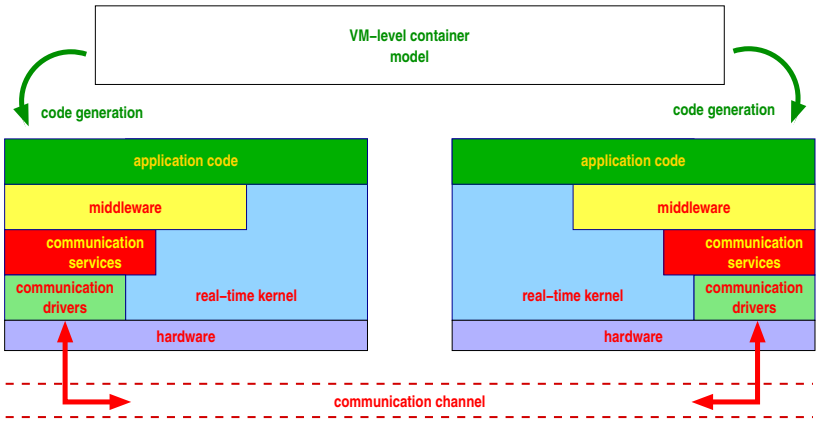


Figure 2: Top-level VM architecture.

to date, in full keeping with the principles discussed in this paper. The VM architecture includes (cf. figure 2):

- A real-time kernel realized as an evolution of ORK [10] and integrated with the GNAT for LEON compiler[1].
- A communications stack for the SOIS Message Transfer Service (MTS) [4].
- A middleware layer based on PolyORB-HI [6].

The completion of the implementation is due in early June 2007 and its use in a suite of end-of-project industrial case studies shall start immediately after that, with culmination in an integrated public demonstration scheduled for late November 2007.

Earlier versions of the ASSERT VM were distributed to the industrial partners in the project and used to explore the implementation of exploratory elements of the intended case studies. Feedback from industrial use is very encouraging. The implementation effort required to date and expected for completion has proven to be ordinate and, thus perfectly acceptable. The early performance figures (with regard to timing and sizing particularly) also seem encouraging though a more thorough assessment of them will most certainly come from the planned case studies.

Overall the message we wish to bring to the reader is that the strategic direction we have taken in the project seems not only intellectually attractive but it also appears to be relevant to industry as well as practical to engineer and use.

4. REFERENCES

[1] S. 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 Real-Time Systems Symposium*, pages 396–409. IEEE, December 2003.

[2] A. Burns, B. Dobbing, and T. Vardanega. Guide for the use of the ada ravenscar profile in high integrity systems, 2003.

[1] `www.adacore.com`

[3] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley, 3 edition, 2001.

[4] Consultative Committee for Space Data Standards (CCSDS). *CCSDS Spacecraft On-board Interface Services Green Book – CCSDS 830.0-G-0.4*, Dec. 2004. Draft.

[5] Gaisler Research. *LEON2 Processor User's Manual*, 2005.

[6] J. Hugues, B. Zalila, and L. Pautet. Middleware and tool suite for high integrity systems. In *Proceedings of RTSS-WiP'06*, pages 1–4, Rio de Janeiro, Brazil, December 2006. IEEE.

[7] J. A. Pulido, S. Urueña, J. Zamorano, and J. A. de la Puente. Handling temporal faults in Ada 2005. In N. Abdennadher and F. Kordon, editors, *Reliable Software Technologies — Ada-Europe 2007*, number 4498 in LNCS, pages 15–28. Springer-Verlag, 2007.

[8] I. Pyarali, C. O'Ryan, D. Schmidt, N. Wang, W. Kachroo, and A. Gokhale. Applying optimization principle patterns to desgin real-time ORBs. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*. USENIX, 1999.

[9] S. T. Taft, R. A. Duff, R. L. Brukardt, E. Plöedereder, and P. Leroy, editors. *Ada 2005 Reference Manual. Language and Standard Libraries. International Standard ISO/IEC 8652/1995(E) with Technical Corrigendum 1 and Amendment 1*. Number 4348 in Lecture Notes in Computer Science. Springer-Verlag, 2006.

[10] S. Urueña, J. A. Pulido, J. Zamorano, and J. A. de la Puente. Adding new features to the Open Ravenscar Kernel. In *1st International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2005)*, Palma de Mallorca, Spain, July 2005.

[11] J. Zamorano, J. F. Ruiz, and J. A. de la Puente. Implementing Ada.Real_Time.Clock and absolute delays in real-time kernels. In A. Strohmeier and D. Craeynest, editors, *Reliable Software Technologies — Ada-Europe 2001*, number 2043 in LNCS, pages 317–327. Springer-Verlag, 2001.

Lazy Queueing and Direct Process Switch — Merit or Myths?

Kevin Elphinstone *
National ICT Australia
and
University of New South Wales
Sydney, Australia
kevine@cse.unsw.edu.au

David Greenaway
National ICT Australia
and
University of New South Wales
Sydney, Australia
davidg@cse.unsw.edu.au

Sergio Ruocco
Nomadis Lab., DISCo,
Università di Milano-Bicocca
Milano, Italy
ruocco@disco.unimib.it

ABSTRACT

The L4 microkernel, like many first and second generation microkernels, was designed to maximise best-effort performance. One component of its functionality critical to overall system performance is its interprocess communication primitive. L4 uses two techniques to minimise communication costs: direct process switching and lazy queue management. These techniques improve performance at the expense of real-time predictability of the scheduler. Now that L4 is being adopted in the embedded space, which features real-time requirements, we must determine if there is continued merit in using the optimisations. In this paper we quantitatively analyse the two optimisations using different kernel implementations and measure the performance improvements of the optimisations directly, and indirectly using the Re-aim benchmark suite. We find that the system-level performance improvements are marginal for this Unix-like workload.

1. INTRODUCTION

The functionality and the overall system complexity of high-end embedded systems is rapidly approaching, and in some cases surpassing, those of desktop systems. At the same time, they are expected to be much more reliable and robust than desktop systems as in most cases embedded systems cannot be managed by their users, and often they cannot be physically serviced.

However, traditional operating systems that are cut down to run in the embedded space usually struggle to provide strong real-time guarantees as their original design aimed at best-effort system performance, and kernel components such as interrupt handlers run outside the scheduler's control. While timeliness can be addressed in part by running the OS on top of a realtime executive, this does not help with the reliability and complexity issues. In fact, an already large desktop operating system is expanded further by a real-time executive, and potentially real-time tasks that run without any isolation next to the desktop kernel.

To address these requirements embedded systems are moving, on the hardware side, towards processors featuring full memory management (i.e., translation and protection), and, on the software side, towards microkernel-based systems, where operating system services run as separated user-level applications, safely isolated by hardware protection.

In principle, compared to a monolithic system, in a microkernel-based system it should be easier to tame complexity and provide timeliness for high-end embedded systems. System services are decomposed into user-level services that contain most of the system functionality (and hence complexity). These user-level services execute under microkernel enforced protection boundaries (processes), which should result in improved system reliability through well defined modular structuring, and better fault isolation and fault identification.

Timeliness benefits from the smaller kernel in two ways. Firstly, being smaller in size, the kernel should be more amenable to whole kernel analysis and carefully targeted modifications to provide or improve real-time behaviour, as there are significantly less lines of privileged code to analyse or modify. Secondly, a real-time capable microkernel provides its real-time guarantees to higher-level services including interrupt handlers, device drivers and other traditional kernel services. Kernel activities that are difficult to account for (or are ignored completely) in monolithic systems, become user-level applications under control of the scheduler and the guarantees it provides.

In practice, like traditional monolithic systems, general-purpose microkernels stem from performance-driven designs, and have ingrained in their design or implementation many optimisations that aim to improve best-effort system performance at the expense of the predictability in scheduling required for real-time systems [18].

L4 [11] is a general-purpose microkernel well-known in academic circles for its contributions to low overhead communication between processes [13]. Recently L4 is gaining an industrial foothold as a basis for high-end embedded and mobile systems and as a virtualisation platform.

In this paper we focus on two performance optimisations performed in the L4 microkernel interprocess communication primitive, commonly known in the L4 community as *the IPC path. Direct process switching*, that avoids running the scheduler along the kernel's critical paths, and *lazy queue-*

*National ICT Australia 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.

ing, that defers the updating of its ready queue. These optimisations decrease the cost of interprocess communication (IPC), but, as a side-effect, the first can temporarily violate the scheduling policy of the system, while the second, in pathological cases, may increase its latency to external events. We describe these two optimisations in detail, provide qualitative arguments both for and against their use, and quantify their performance benefits to allow kernel engineers and users to weigh their pros and cons in both best-effort and real-time scenarios.

2. INTERPROCESS COMMUNICATION OPTIMISATION

2.1 The Pursuit of IPC Performance

The intended structure of microkernel-based systems puts heavy demands on the performance of IPC. In microkernel-based systems, traditional operating system services — such as device drivers, filesystems and network stacks — are provided by processes (servers) running at user-level. Thus, instead of a system call to a traditional monolithic operating system, in a microkernel-based system all interactions between applications and system services involve IPC to and from servers implementing those services.

In the L4 microkernel the basic IPC mechanism is used not only to transfer messages between user-level threads, but also to deliver interrupts, asynchronous notifications, memory mappings, thread startups, thread preemptions, exceptions and page faults. Because of its pervasiveness, IPC is likely to be used very frequently. It is also evident that any kernel change that increases IPC costs will increase overhead.

It is then clear why IPC performance in L4 (and in general) has received so much attention (including, but not limited to [2,10,13,19]), with achieved performance being sufficient to support near-monolithic system performance when all system-call-like invocations are implemented by IPC to a system server, which in this case was Linux [3]. Härtig *et al.* also directly compared their system with MkLinux (a directly comparable version of Linux based on a microkernel with slower IPC performance) and demonstrated that the version of Linux running on the slower microkernel exhibited a 25% performance penalty. IPC performance was critical to overall system performance.

The requirement for high IPC performance is further motivated when device drivers are run as user-level servers to improve robustness and reduce kernel complexity [7,12]. In L4, hardware interrupt delivery is via IPC to interrupt handling threads. Interrupt delivery overhead can be critical for hardware devices such as gigabit Ethernet where, to reduce the performance impact of high interrupt rates, hardware-based interrupt throttling is now common for even normal interrupt delivery.

IPC performance affects not only the overall performance of the system, but also its design space. It has been observed early in the evolution of microkernels that given poor IPC performance, system builders will work around it by either co-locating services back within the kernel, or by composing the system with much coarser granularity than they would otherwise [4].

In addition to supporting decomposed services and applications, microkernels can also support virtual machine monitors (VMMs) as an approach to supporting legacy operating systems [3,9,17], while concurrently providing isolated environments for microkernel-based applications and services that, to implement security or temporally critical services, rely only on the guarantees provided only by the microkernel [14].

VMMs require efficient exception handling to emulate privileged instructions present in the hosted operating system. Also relevant for VMMs is the vectoring of native system calls to a paravirtualised operating system server running on the microkernel [1].

In the case of a paravirtualised Linux running on top of L4, both exception and syscall delivery are again via IPC from the Linux applications to the Linux server, or from the Linux server to the virtual machine monitor. Again IPC performance plays a primary role in determining system performance for system call-intensive (or exception-intensive) applications.

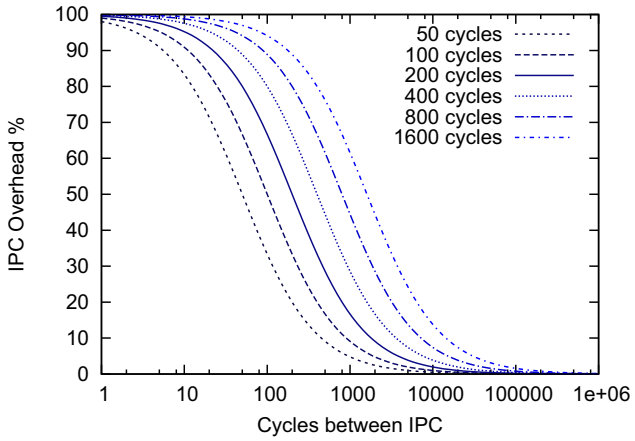


Figure 1: The IPC overhead for various IPC costs against the average cycles between successive IPC.

Figure 1 illustrates the sensitivity of IPC overhead to the raw cost of IPC, and the average cycles between successive IPCs. The x-axis represents the average number of cycles between each IPC, on a log scale. Each line is a plot of the percentage overhead attributable to IPC for hypothetical IPC costs of 50, 100, 200, 400, 800 and 1600 cycles, which are within the range of typical IPC warm-cache costs. It shows how the overhead is particularly sensitive to IPC costs in the range surrounding 5000 cycles. This corresponds broadly to an hypothetical gigabit Ethernet interrupt delivery intervals (the time to receive a minimum sized packet on the wire) if hardware-based interrupt throttling is not used. Fine-grained synchronisation would also be in this range.

Summarising, there is strong motivation for minimising IPC costs if it is invoked frequently, and this is certainly the case for L4. In the remainder of this paper we will discuss two generally-applicable IPC optimisations used in L4 that

reduce IPC cost, but affect in major and minor ways its realtime scheduling behaviour. In the following sections, to illustrate the two optimisations we will consider the case of a interprocess communication where one process *calls* another to request a service, resulting in the caller becoming blocked and the called becoming runnable; the reverse happens as a result of the response.

2.2 Direct Process Switch

In principle, every operating system, when the current process blocks, invokes the scheduler to choose the next process to run based on the specific scheduling policy it implements, e.g., the highest priority runnable process will run. However, if the process blocks in the IPC path, invoking the scheduler can be a costly operation that impacts IPC performance. Therefore L4 avoids it and switches directly to the newly runnable IPC destination, often disregarding relevant scheduling criteria, such as priority of other threads in the system.

The advantages of direct switch in the IPC path are threefold: (i) the overhead involved in calling the scheduler in the performance-critical IPC path is avoided, (ii) the latency of reaction to events delivered via IPC is reduced (also the interrupt fastpath performs a direct switch), and (iii) the cache working set may be reduced.

The first benefit does not warrant further explanation. The second benefit is advantageous (e.g. during interrupt handling) as it gives a process the opportunity to service the interrupt earlier, and therefore potentially request the next I/O operation earlier, improving I/O utilisation. The third benefit occurs as a client and server which interact closely can share the cache without the scheduler interfering by polluting the cache with the correct scheduling of a third process.

Direct process switch was first proposed by Liedke [10] to improve microkernel IPC performance. However, it makes the real-time schedulability analysis for any specific scheduling policy difficult, if not impossible, as the scheduler is not involved in the majority of scheduling decisions. In fact, scheduling decisions due to IPCs happen a few thousands of times per second, one or two orders of magnitude more frequently than those due to the scheduler running after timeslice preemptions, which happens a few hundreds of times per second.

Historically, direct switch has also been applied inconsistently in L4. To decide which process should run some L4 implementations consider the priorities of the communicating processes and the type of IPC performed, others do not, but all of them bypass the scheduler on the critical path. Ruocco [18] provides a detailed analysis of the direct switch behaviours in two recent kernels of the L4 family, and their implications for priority-driven real-time scheduling.

Notably, also the real-time OS QNX Neutrino seems to perform a direct switch in synchronous IPCs when data transfer is involved [16]:

> **Synchronous message passing**
>
> This inherent blocking synchronizes the execution of the sending thread, since the act of requesting that the data be sent also causes the sending thread to be blocked and the receiving thread to be scheduled for execution. This happens without requiring explicit work by the kernel to determine which thread to run next (as would be the case with most other forms of IPC). Execution and data move directly from one context to another.

2.3 Lazy Queueing

When performing a remote procedure call (RPC) over synchronous IPC, the sender thread blocks after sending the message, and the waiting receiver thread is unblocked after receiving the message. The blocking and unblocking of threads results in ready queue manipulation. The blocked thread must be removed from the ready queue, and the unblocked thread must be inserted in the ready queue. If two threads interact in a tight client-server loop, this happens continuously, undoing work just performed, and then performing it again.

Lazy queueing consists of the kernel deferring work in the hope that it is eventually unneeded. L4 performs lazy queue management with the following two techniques:

1. A blocking thread is not immediately removed from the ready queue. Its removal is deferred until the scheduler is called. The scheduler then removes any blocked thread(s) it encounters in the course of searching the next thread to run in the ready queue.

2. The kernel preserves the invariant that at least all ready threads not currently running must be in the ready queue. The currently running thread is not required to be in the ready queue.

   If the currently running thread is preempted (changes state from *running* to *ready*), it is added to the queue if it is not already present. Thus, switching briefly to a newly runnable thread does not require adding it to the ready queue.

In L4, the combination of these two techniques ensures that when IPC results in just one thread blocking and another running, short-lived updates that elide each other are avoided, and unavoidable queue maintenance is deferred as much as possible. The ready queue is finally updated when the messaging is preempted and the scheduler runs, typically as a result of timeslice exhaustion or a blocking IPC to a busy thread.

The pros of lazy queueing are saving the direct cost of the queue management, the indirect cost of an increased number of cache lines polluted by the queue manipulation, and avoiding the potential pollution of TLB entries, depending on the architecture and virtual memory mapping.

While lazy queueing cannot result in more overall processing performed compared to strict queue management, it does defer queue maintenance to the scheduler, where the scheduler

may encounter (and remove) blocked threads in the ready queue. The number of blocked threads encountered is difficult to predict, resulting in latency of scheduling operations also being difficult to predict.

Finally, a note on terminology. Liedtke [10] calls lazy scheduling what in this paper we call lazy queueing. We use the latter term to avoid confusion with direct process switch, which can be considered a form of lazy scheduling. That said, in the L4 community and literature the term 'lazy scheduling' is sometimes used loosely to indicate a generic optimisation in scheduling, and thus can refer to direct process switch, lazy queueing, or even both optimisations.

2.4 Related Work

While there is a body of work on IPC performance, and on real-time kernels, besides the analysis mentioned above [18], there is little in the literature on the trade-off between the two IPC optimisations described, and a kernel's ability to support real-time workloads. The most relevant is Steinberg *et al.*, who proposed extending L4's IPC mechanism to donate scheduling context in order to support various classes of real-time scheduling disciplines, including priority inheritance, and reservation-based realtime systems on L4 [20].

They augment L4's IPC to do the book keeping required to track dependencies and time-slice donations. This work is complementary to our work in that they also demonstrate the need to modify their microkernel's IPC implementation to achieve their desired scheduling behaviour. They acknowledge that performance is an issue, and argue qualitatively that their system's approach adds little overhead. However, no quantitative results are given and their baseline IPC overhead is up to an order of magnitude higher than the costs we have quantified. Given a highly optimised IPC, on a favourable architecture, it is unclear that their changes would continue to be "little overhead".

2.5 Summary

In this section we have argued that IPC performance is important, and described two optimisations used in the L4 microkernel to reduce the direct and indirect costs of IPC, but with a detrimental effect on real-time workloads. Direct switch comes at the expense of the system no longer strictly adhering to its own scheduling policy — especially in the case of priority-driven scheduling — and precluding the schedulability analysis of a real-time system. Lazy queueing can increase latency in pathological cases.

In the remainder of the paper we quantify the cost in terms of performance of direct switch and lazy queueing by benchmarking the standard kernel, then removing selectively each optimisation, and finally both of them. We aim to clarify the trade-off between using and not using the two optimisations, to offer microkernel designers and users an educated choice between performance and predictability.

3. EXPERIMENTS

We performed three experiments to quantify performance differences between various optimisation configurations. The first was to instrument the kernel to collect statistics on number of IPCs, context switches, queueing operations, and scheduler invocations to determine how often queueing or scheduling is avoided. The second experiment microbenchmarks the IPC performance directly by timing repeated ping-pong messages. The third experiment measures throughput for individual components of the Re-aim benchmark suite running on a paravirtualised Linux, which, in turn, runs on the microkernel. Each experiment is described in more detail in the following sections.

For this paper we used L4-embedded N2 v1.3.0 [15], derived from L4::Ka Pistachio 0.4 [6], as a representative microkernel for experimentation. Both of them feature both the direct switch and lazy queueing optimisations described earlier. The hardware platform we used was a Gumstix Connex 400xm, which has an XScale PXA255 clocked at 400 MHz, and 64 MB of RAM. In addition to L4-embedded, we also use the Iguana operating system personality running on L4, together with Wombat (a version of Linux paravirtualised to run on Iguana on L4), to provide a system for higher-level benchmarking. Further details follow in Section 3.2.

3.1 Kernel Internal Scheduler Interface

To experiment with various combinations of scheduler optimisations, we constructed an internal scheduling interface within the L4 kernel that allows a compile-time selection of schedulers with different optimisations.

Scheduling in L4 is scattered through various parts of its source code, where scheduling decisions are made implicitly in the source each time two threads interact. For instance, when two threads communicate using IPC, the IPC code determines which thread should execute next at the conclusion of the operation without involving the scheduler, often — but not always — by directly comparing the priorities of the two threads.

The creation of an internal interface involved refactoring the code to remove the implicit scheduling decisions, which can then be centralised and performed explicitly according to a uniform and easily changeable policy.

One significant impact of this centralisation of scheduling was that the highly-optimised assembly language IPC path (known as the *fastpath*) was disabled. Instead, IPC is routed to a slower *C* language path, which uses the new internal interface.

Each of the new scheduler interface calls that involve a schedule operation also takes an additional parameter we termed a *scheduling hint*. Scheduling hints were introduced to allow the same interface to support both the behaviour of existing L4 implementations, which often dictate which thread is to be scheduled next, while also allowing other scheduling policies to be implemented, such as strict priority observance.

Listing 1 illustrates three different scheduling hints that were required to mimic the existing behaviour of L4: (i) a hint indicating that the highest priority thread in the system should be scheduled; (ii) a hint that the most recently enqueued thread should be scheduled (in the case of IPC, this emulates direct process switch); and (iii) a hint indicating that either the currently running thread or the most recently

enqueued thread should be scheduled, whichever has the highest priority (used in the case of send-only IPC or interrupts).

```
/* Hints describing traditional L4 behaviour */
typedef enum {

    /* Schedule highest priority thread */
    HINT_HIGHEST_PRIORITY,

    /* Schedule most recently enqueued thread */
    HINT_NEW,

    /* Schedule the current or just-enqueued thread */
    HINT_CURRENT_OR_NEW,

} hint_t;

/* Ready queue manipulations */
void enqueue (tcb_t *);
void dequeue (tcb_t *);
void swap    (tcb_t *, tcb_t *);

/* Request scheduler to perform a context switch */
void sched (hint_t);

/* Manipulate ready queues and perform a switch */
void enqueue_sched (tcb_t *,          hint_t);
void dequeue_sched (tcb_t *,          hint_t);
void swap_sched    (tcb_t *, tcb_t *, hint_t);
```

Listing 1: The new L4 internal scheduling API

In addition to the hints, there are four functions which: *enqueue* or *dequeue* a thread in the ready queue, atomically block one thread and start another (*swap*), and *schedule* which chooses which thread to run next, and context switches to it. There are also three more interface functions which are clearly combinations of the previous four.

3.1.1 Measured Schedulers

In our experiments we investigated five variants of the L4-embedded kernel. The first variant was an unmodified L4-embedded kernel **Unmod**. As mentioned earlier, this kernel features an optimised assembly IPC path which was not used for the remainder of the measured variants, as we are yet to write assembly language versions of the measured schedulers that would be suitable for inclusion on an assembly language IPC path. **Unmod** simply represents a best case for comparison to gauge the effect the C internal kernel scheduling interface has on IPC performance.

The other four variants we investigated used the internal scheduling interface, described in detail in Section 3.1. These four variants implement combinations of either direct process switching (DS) or full scheduling (FS), and lazy queueing (LQ) or eager queueing (EQ). The combinations are as follows:

DS/LQ L4 with the scheduler bypassed during IPC (direct switch) with lazy queue management;

FS/EQ L4 with a full scheduler call in the IPC path together with eager queue management;

FS/LQ L4 with a full scheduler call in the IPC path together with lazy queue management;

DS/EQ L4 with the scheduler bypassed during IPC (direct switch) with eager queue management.

Note that the **DS/LQ** kernel variant reproduces the scheduling behaviour of the **Unmod** kernel using the new internal scheduling API. Therefore, the difference between **DS/LQ** and **Unmod** reflects the overhead of the C-based scheduler interface, and the lack of an optimised assembly IPC path. Obviously excluding **Unmod**, the remaining four variants are similarly implemented and are directly comparable.

3.2 Benchmarks

We compared the four scheduler variants (together with the unmodified kernel) using three approaches. Firstly, we measured the number of raw operations to determine how much queue and scheduling avoidance occurs when the optimisations are applied. Secondly, we directly measured the cost of raw IPC. Thirdly, we measured how the scheduler variants impact on the throughput of a para-virtualised version of Linux. A more detailed description of the specific benchmarks begins in Section 3.2.2, but before that we describe in more detail Wombat, our para-virtualised Linux environment, together with the Re-aim benchmark suite.

3.2.1 Wombat and Re-aim

To measure the effect on overall system performance of the scheduler variants we use *Wombat* [8], a paravirtualised version of Linux running on top of L4/Iguana [5]. A Wombat system is structured as depicted in Figure 2. The Iguana embedded OS acts as a virtual machine monitor for Wombat. Iguana provides services such as address spaces, threads, and some services (such as device drivers) that run as Iguana applications. Linux is modified by providing an L4/Iguana CPU architecture which, instead of performing direct low-level, privileged CPU operations, it uses IPC to request Iguana to provide threads and provide and manipulate address spaces for Linux processes running on Wombat, and to Wombat itself.

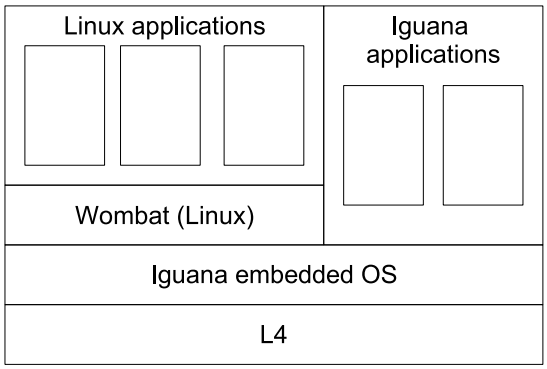


Figure 2: Wombat, Iguana OS and L4.

Overall system performance will depend on (i) the cost of propagating native system-call exceptions as IPCs from Linux applications to the Wombat instance acting as a Linux server for the Linux applications, and (ii) the cost of IPC to

the Iguana virtual machine monitor when Wombat requires changes to the low-level hardware artifacts. Thus any variation in raw IPC costs may be visible depending on the level of interaction between Iguana, Wombat, and Linux (applications) processes.

To determine the relative performance of Wombat, we used the Re-aim benchmark suite [21]. Re-aim provides two modes of determining system performance. First, Re-aim has a *single-user* mode which measures the throughput of a series of operations, such as the number of TCP/IP operations performed per second, number of processes created per second, the number of floating point operations per second, and so on.

Second, Re-aim has a *multi-user* mode which attempts to simulate real-world workloads. We ran the full Re-aim multi-user benchmark with five processes, each of which carries out a series of tasks in a pseudo-random order exercising both the CPU and kernel. In order to ensure the results were reproducible, we modified the Re-aim benchmark source code to seed its random number generator on the child-number of the benchmark processes, instead of the normally used Linux process-id.

Note that Wombat and Re-aim use a RAM disk to back the file system, so no real I/O occurs in the benchmarks.

3.2.2 Avoiding Work

This specific experiment uses the *multi-user* Re-aim benchmark described above. The scheduler variant is **DS/LQ**, but it has been modified to keep statistics such as number of IPCs, total actual enqueue and dequeue operations, and also total number of operations that would have occurred with eager queueing. This instrumentation is only included for this particular experiment, and only counts events. The instrumentation is not used in cases where we measure performance elsewhere in the paper.

The statistics can be used to determine how many enqueue, dequeue and scheduler operations are avoided, to illustrate the effectiveness of the technique.

3.2.3 Ping Pong

This microbenchmark consists of *ping pong*, where a low-priority client sends a message of a fixed length to a high-priority server, which then in turn responds immediately back with a message of the same length. The benchmark directly measures L4 and thus is independent of Wombat and Iguana.

We benchmark 1 000 000 iterations of ping pong using the cycle counter in the performance monitoring unit of the PXA255, and the average number of cycles of a single IPC is determined. The process is repeated for messages of various lengths. The final number of cycles counted includes both the time required for the user-level threads to call the kernel and the time spent in the kernel performing the IPC operation. We run this benckmark for each of the kernel configurations we have.

3.2.4 Re-aim Throughput

Benchmark	Task
`brk_test`	Carry out the `brk` syscall in a loop.
`creat_clo`	Create and then close files in a loop.
`dgram_pipe`	Send and receive random-length datagram packets.
`dir_rtns_1`	Carry out various directory querying syscalls.
`exec_test`	Create children with `fork`, which in turn carry out an `exec`.
`fork_test`	Create and wait for child processes using `fork` and `wait`.
`link_test`	Create and destroy hard links to individual files.
`misc_rtns_1`	Carry out miscellaneous Unix query syscalls.
`page_test`	Allocate and deallocate memory with `sbrk`.
`pipe_cpy`	Send and receive random-length packets over a Unix pipe.
`shared_memory`	Perform semaphore operations and read/write operations on shared memory.
`shell_rtns`	Execute simple shell scripts in a loop.
`signal_test`	Send and catch Unix signals in a loop.
`stream_pipe`	Send and receive random amounts of data of a Unix stream.
`udp_test`	Send and receive random-length UDP packets over loopback.

Table 1: Descriptions of the Re-aim single-user benchmark tasks tested.

Our last experiment takes selected single-user throughput benchmarks from the Re-aim suite. The throughput is determined by counting the number of completed activities in an interval of approximately 10 seconds (the cycle-counter on the PXA255 is used to get an accurate measurement of the length of the interval). The specific activity counted was dependant on the actual benchmark component under test. For example, the UDP test counts the number of packets sent and received. Each benchmark specific throughput was an average of 4 runs. Each runs was closely consistent with the others, the standard deviation was always less than 0.5 percent , and the average standard deviation was 0.08 percent.

Table 1 briefly describes the selected benchmarks. The selection excludes the CPU oriented benchmarks whose performance is largely independent of the underlying operating system architecture and implementation and thus not relevant for this paper.

4. RESULTS

4.1 Work Avoidance

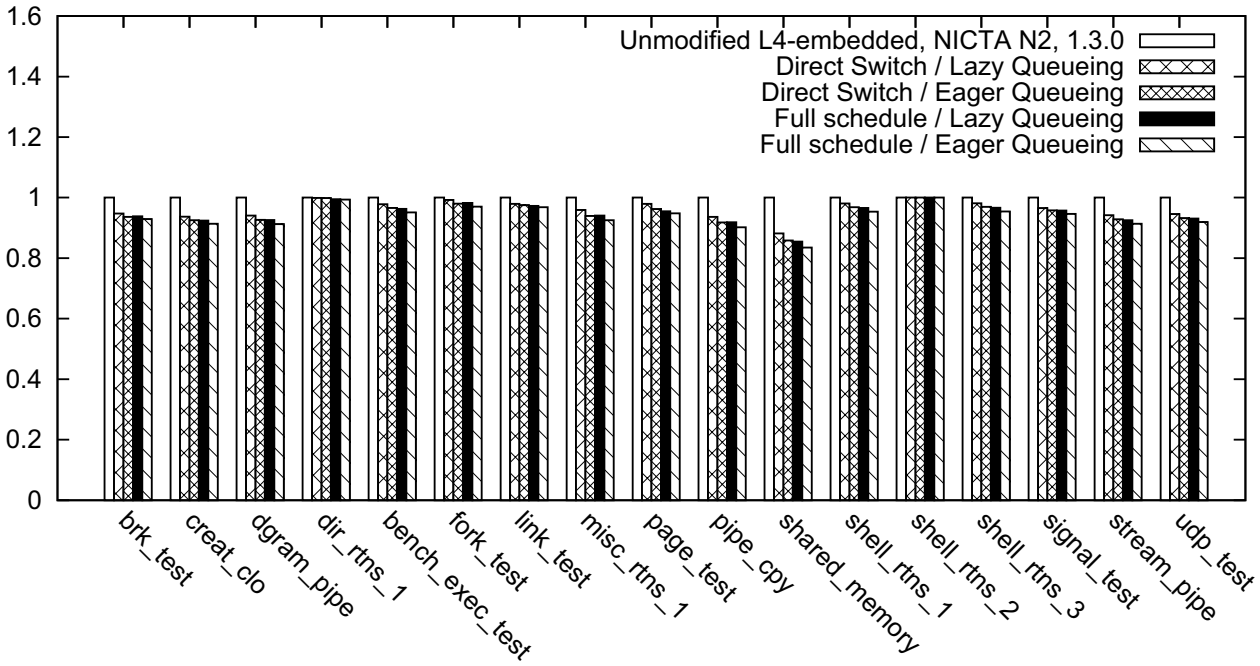


Figure 4: Individual Re-aim benchmark components normalised to standard fastpath L4

Operation	Result
Benchmark Length (seconds)	222.16
IPCs	1450474
Average IPC Length (32-bit words)	6.26
Eager enqueue operations	1509011
Actual enqueue operations	62482
At a deferred time	19719
Enqueue operations avoided	95.86%
Eager dequeue operations	1509056
Actual dequeue operations	62482
At a deferred time	40289
Dequeue operations avoided	95.86%
Context switches	1571609
Scheduling queue lookups	80749
Queue lookups avoided	94.86%

Table 2: Breakdown of L4 operations for the Re-aim multi-user 'all tests' benchmark

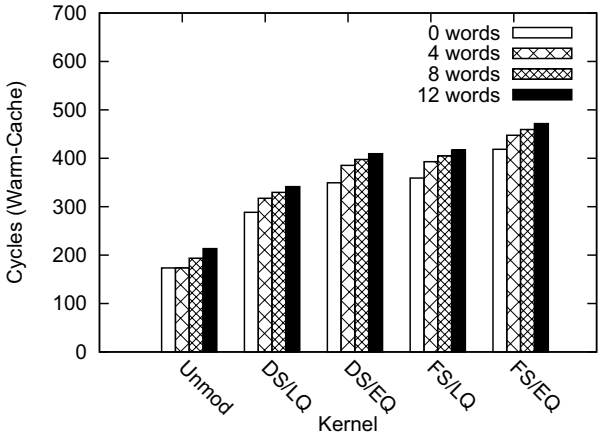


Figure 3: Raw IPC costs for various scheduler implementations.

Table 2 summarises the results of our experiment on work avoidance. The table is divided into 4 sections: general statistics of the Re-aim multi-user benchmark, enqueue operations, dequeue operations, and scheduler invocations. We see that the overall benchmark takes 222 seconds to run, averaging one IPC per 150 microseconds, which is every 62000 cycles. As Figure 1 suggests, in this inter-IPC cycle range, small variations in IPC duration should have a very small effect on the overall run-time of Re-aim, unless IPC costs become substantially greater than 1000 cycles.

Looking at the queueing results, we see that the application of lazy queue management reduces the number of queue operations substantially. We see that 96% of queue operations are avoided altogether with the technique, even when we include queueing operations that are not avoided entirely and are only deferred to a later point in time. The scheduling results show a similar percentage (95%) of scheduler invocations are avoided by the direct process switch technique.

Summarising, we see that direct process switching and lazy queue management are very effective in avoiding scheduling and queueing costs. However, given the infrequency of IPC

in the Re-aim multi-user benchmark, we don't expect to see IPC overheads greater that a percentage point or two on average. However, we will see later that some of the individual benchmarks do vary significantly.

4.2 Ping Pong

Figure 3 shows the results of the ping pong benchmark for the 5 kernels. We see that **Unmod** kernel with its assembly IPC path is significantly faster (174 cycles for a zero-sized message) than **DS/LQ** (289 cycles) despite implementing the same algorithm.

This difference is attributable to two factors. The first factor is the assembly only IPC path in **Unmod** avoids preparing the kernel stack to call C, and avoids preserving the C compiler function calling convention on a context switch. The second factor is that the C path used in **DS/LQ** is a modified version of a slower IPC path (the IPC *slowpath*), also written in C. While the fastpath can handle only a frequently-used subset of all IPC cases, the slowpath can handle all of them.

Examining the four directly comparable results for the kernels with different combinations of direct switch and lazy queueing, we have the following results for zero-sized messages: **DS/LQ** 289, **DS/EQ** 350, **FS/LQ** 359, and **FS/EQ** 419. We see that direct switching saves 70 cycles off the IPC path, and lazy queue management saves 60 cycles off the IPC path. We see that there are comparatively large savings to be made to the raw cost of IPC by using both optimisation techniques.

4.3 Re-aim performance

The results for the single-user Re-aim benchmarks are shown in Figure 4. We see throughput results for the individual benchmark tests within the suite, normalised to the throughput of **Unmod**. The influence IPC performance has over the individual benchmarks varies from virtually no influence in the case of `dir_rtns_1` and `shell_rtns_2`, to a significant difference of a 17% reduction in throughput for the `shared_memory`, when comparing the assembly path kernel **Unmod** to the slowest C path kernel **FS/EQ**.

Now examining comparable results, we see the biggest differences (between **DS/LQ** and **FS/EQ**) is in the `shared_memory` benchmark, with a reduction of 5% in throughput. The average reduction in throughput was 2.5% for all the individual benchmarks.

5. CONCLUSIONS

We have described and motivated two general IPC optimisations that have historically been used in the L4 microkernel: direct process switching and lazy queueing. We have argued that the optimisations have negative consequences on real-time predictability as they undermine the scheduling policy, and defer a difficult-to-predict amount of work for when the scheduler is eventually invoked.

However, we also determined via measurement that these optimisations avoided scheduling related activity from IPC in over 95% of IPC invocations. The consequent improvement in best-effort system performance was dependent of the relative costs of IPC and scheduling activity, and the frequency of IPC invocation.

When we quantified their effect on the overall system performance, we found that the performance gains are modest. As expected, the overhead of IPC depends on its frequency. Removing the optimisations reduced system throughput by 2.5% on average, 5% in the worst case. Thus the case for including the optimisations at the expense of real-time predictability is weak for the cases we examined. For much higher IPC rate applications, it might still be worthwhile.

We acknowledge two weak points in our comparison that we intend to address in future work. Firstly, the work was done in C, which for IPC incurs a substantial overhead compared to the optimised assembly version. The sensitivity of IPC overhead to scheduler implementation may change in a faster assembler-only implementation. Secondly, we only examine the PXA255: other processor architectures and implementations have much larger or smaller relative IPC cost to which the results may be sensitive to.

However, our results confirm that the trade-off between performance and real-time predictability exists. For the cases we investigated, which are designed to model Unix system use, the performance gain is small, and unjustified when considering the loss of predictable scheduler behaviour.

6. REFERENCES

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on OS Principles*, pages 164–177, Bolton Landing, NY, USA, October 2003.

[2] Charles Gray, Matthew Chapman, Peter Chubb, David Mosberger-Tang, and Gernot Heiser. Itanium — a system implementor's tale. In *Proceedings of the 2005 Annual USENIX Technical Conference*, pages 264–278, Anaheim, CA, USA, April 2005.

[3] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on OS Principles*, pages 66–77, St. Malo, France, October 1997.

[4] Wilson C. Hsieh, M. Frans Kaashoek, and William E. Weihl. The persistent relevance of IPC performance: New techniques for reducing the IPC penalty. In *Workshop on Workstation Operating Systems*, pages 186–190, 1993.

[5] Iguana OS. URL http://www.ertos.nicta.com.au/iguana/.

[6] L4Ka Team. L4Ka::Pistachio kernel. http://l4ka.org/projects/pistachio/.

[7] Ben Leslie, Peter Chubb, Nicholas Fitzroy-Dale, Stefan Götz, Charles Gray, Luke Macpherson, Daniel Potts, Yueting (Rita) Shen, Kevin Elphinstone, and Gernot Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5):654–664, September 2005.

[8] Ben Leslie, Carl van Schaik, and Gernot Heiser. Wombat: A portable user-mode Linux for embedded

systems. In *Proceedings of the 6th Linux.Conf.Au*, Canberra, April 2005.

[9] Joshua LeVasseur and Volkmar Uhlig. A sledgehammer approach to reuse of legacy device drivers. In *Proceedings of the 11th SIGOPS European Workshop*, 2004.

[10] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on OS Principles*, pages 175–188, Asheville, NC, USA, December 1993.

[11] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on OS Principles*, pages 237–250, Copper Mountain, CO, USA, December 1995.

[12] Jochen Liedtke. Towards real microkernels. *Communications of the ACM*, 39(9):70–77, September 1996.

[13] Jochen Liedtke, Kevin Elphinstone, Sebastian Schönberg, Herrman Härtig, Gernot Heiser, Nayeem Islam, and Trent Jaeger. Achieved IPC performance (still the foundation for extensibility). In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, pages 28–31, Cape Cod, MA, USA, May 1997.

[14] Frank Mehnert, Michael Hohmuth, and Hermann Härtig. Cost and benefit of separate address spaces in real-time operating systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, Austin, TX, USA, 2002.

[15] National ICT Australia. *NICTA L4-embedded Kernel Version N2 v. 1.3.0*. http://www.ertos.nicta.com.au/software/kenge/pistachio/latest/.

[16] QNX Neutrino IPC. URL http://www.qnx.com/developers/docs/6.3.0SP3/neutrino/sys_arch/kernel.html#NTOIPC.

[17] Timothy Roscoe, Kevin Elphinstone, and Gernot Heiser. Hype and virtue. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, San Diego, CA, USA, May 2007.

[18] Sergio Ruocco. Real-Time Programming and L4 Microkernels. In *Proceedings of the 2006 Workshop on Operating System Platforms for Embedded Real-Time Applications*, Dresden, Germany, July 2006.

[19] Jonathan S. Shapiro, David F. Faber, and Jonathan M. Smith. The measured performance of fast local IPC. In *Proceedings of the 5th IEEE International Workshop on Object Orientation in Operating Systems*, pages 89–94, Seattle, WA, USA, October 1996. IEEE.

[20] U. Steinberg, J. Wolter, and H. Härtig. Fast component interaction for real-time systems. In *Proc. 17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, Palma de Mallorca, Spain, July 2005.

[21] Cliff White. Performance testing the Linux kernel. In *Proceedings of the Linux Symposium*, Ottowa, Canada, July 2003.