OSPERT 2009:

International Workshop on Operating Systems Platforms for Embedded Real-Time Applications

Editors:
Stefan M. Petters
Peter Zijlstra

Pre-Workshop Proceedings

Workshop Date: 30.06.2009

Dublin, Ireland

Table of Contents

Message from the Workshop Chairs

The preparations of this year's instalment of the Workshop on Operating Systems Platforms for Embedded Real-Time Applications were focussed on one hand on creating a very interactive format and on the other hand on nurturing the exchange of ideas between industry and academia. The first thanks goes to Gerhard Fohler for giving us the opportunity to pull this off. We aimed to achieve the objectives by scheduling two discussion sessions one of which stacked with a high profile panel of academic researchers and industry practitioners.

In this context we would like to thank Nicholas Mc Guire, Thomas Gleixner, Scott Brandt and James Andersson for their willingness to share their thoughts on the exchange between industry and academia in the panel session. The second discussion session aims to shed some light on how a test bed for real-time research could look like and will conclude the workshop.

At the start of the day and between the discussion sessions we have scheduled two paper presentations sessions. The 6 papers presented were selected out of a total of 9 submissions. We thank all the authors for their hard work and submitting it to the workshop for selection, the PC members and reviewers for their effort in selecting an interesting program, as well as the presenters for ensuring interesting sessions.

Last, but not least, we would like to thank you the audience for your attendance. A workshop lives and breathes because of the people asking questions and contributing opinions throughout the day.

We hope you will find this day interesting and enjoyable.

The Workshop Chairs

Stefan M. Petters
Peter Zijlstra

Program Committee

Jim Andersson, University of North Carolina at Chapel Hill, USA

Neil Audsley, University of York, UK

Scott Brandt, University of California, Santa Cruz, USA

Peter Chubb, NICTA, Australia

Hermann Härtig, TU Dresden, Germany

Johannes Helander, Microsoft, Germany

Robert Kaiser, University of Applied Sciences Wiesbaden, Germany

Giuseppe Lipari, Scuola Superiore Sant'Anna, Italy

Stefan M. Petters, IPP-Hurray, Portugal

Peter Zijlstra, Red Hat, Netherlands

Workshop Program

09:00-10:30 Session 1

Extending RTAI/Linux with Fixed-Priority Scheduling with Deferred Preemption
Mark Bergsma, Mike Holenderski, Reinder J. Bril and Johan J. Lukkien
Technische Universiteit Eindhoven, Netherlands

Hierarchical Multiprocessor CPU Reservations for the Linux Kernel
Fabio Checconi, Tommaso Cucinotta, Dario Faggioli, Giuseppe Lipari
Scuola Superiore S. Anna, Italy

Threaded IRQs on Linux PREEMPT-RT
Luís Henriques
Intel Shannon, Ireland

10:30-11:00 Coffee Break

11:00-12:30 Session 2 Panel Discussion

Real-Time vs. real fast in academia and industry
Jim Andersson, *The University of North Carolina at Chapel Hill, USA*
Scott Brandt, *University of California, Santa Cruz, USA*
Thomas Gleixner, *Linuxtronix, Germany*
Nicholas Mc Guire, *OpenTech, Austria*
Peter Zijlstra, *Red Hat, Netherlands*

12:30 -14:00 Lunch 14:00-15:30 Session 3

Towards Unit Testing Real-Time Schedulers in LITMUS-RT
Malcolm S. Mollison, Björn B. Brandenburg, and James H. Anderson
The University of North Carolina at Chapel Hill, USA

Exception-Based Management of Timing Constraints Violations for Soft Real-Time Applications
Tommaso Cucinotta, Dario Faggioli
Scuola Superiore S. Anna, Italy
Alessandro Evangelista

Hardware Microkernels for Heterogeneous Manycore Systems
Jason Agron, David Andrews
The University of Arkansas, USA

15:30-16:00 Coffee Break

16:00-17:30 Session 4 Group Discussion

Testbed platforms for RT research:
Wish-list, problems, planning!

Extending RTAI/Linux with Fixed-Priority Scheduling with Deferred Preemption

Mark Bergsma, Mike Holenderski, Reinder J. Bril and Johan J. Lukkien
Faculty of Computer Science and Mathematics
Technische Universiteit Eindhoven (TU/e)
Den Dolech 2, 5600 AZ Eindhoven, The Netherlands

Abstract

Fixed-Priority Scheduling with Deferred Preemption (FPDS) is a middle ground between Fixed-Priority Preemptive Scheduling and Fixed-Priority Non-preemptive Scheduling, and offers advantages with respect to context switch overhead and resource access control. In this paper we present our work on extending the real-time operating system RTAI/Linux with support for FPDS. We give an overview of possible alternatives, describe our design choices and implementation, and verify through a series of measurements that indicate that a FPDS implementation in a real-world RTOS is feasible with minimal overhead.

1 Introduction

Fixed-Priority Scheduling with Deferred Preemption (FPDS) [4–7, 9] has been proposed in the literature as an alternative to Fixed-Priority Nonpreemptive Scheduling (FPNS) and Fixed-Priority Preemptive Scheduling (FPPS) [11]. Input to FPPS and FPNS is a set of tasks of which instances (jobs) need to be scheduled. FPDS is similar to FPNS but now tasks have additional structure and consist of (ordered) subtasks. Hence, in FPDS each job consists of a sequence of subjobs; preemption is possible only between subjobs. The benefits of FPDS, derived from FPNS are (i) less context-switch overhead thanks to fewer preemptions (ii) the ability to avoid explicit resource allocation and subsequent complex resource-access protocols. The fact that subjobs are small leads to FPDS having a better response time for higher priority tasks.

FPDS was selected as a desirable scheduling mechanism for a surveillance system designed with one of our industry partners. [10] With response times found to be too long under FPNS, FPDS was considered to have the same benefits of lower context switch overhead compared to FPPS with its arbitrary preemptions.

In this paper our goal is to extend a real-time Linux version with support for FPDS. For this purpose we selected the Real-Time Application Interface (RTAI) extension to Linux [1]. RTAI is a free-software community project that extends the Linux kernel with hard real-time functionality. We aim to keep our extensions efficient with respect to overhead, and as small and non-intrusive as possible in order to facilitate future maintainability of these changes. Our contributions are the discussion of the RTAI extensions, the implementation[1] and the corresponding measurements to investigate the performance of the resulting system and the introduced overhead.

The work is further presented as follows. We start with an overview of related work and recapitulation of FPDS, followed by a summary of the design and features of RTAI. Then we analyze how FPDS should be dealt with in the context of RTAI. We present our investigation, design and proof of concept implementation of FPDS in RTAI. This result is analyzed through a series of measurements. We conclude with a summary and future work.

2 Related work

As a recapitulation [4–7,9], in FPDS a periodic task τ_i with computation time C_i is split into a number of non-preemptive sub tasks $\tau_{i,j}$ with individual computation times $C_{i,j}$. The structure of all subtasks defining an FPDS task is defined by either the programmer through the use of explicit *preemption points* in the source, or by automated tools at compile time, and can have the form of a simple ordered *sequence*, or a directed acyclic graph (DAG) of subtasks. See Figure 1

[1]This work is freely available at http://wiki.wikked.net/wiki/FPDS

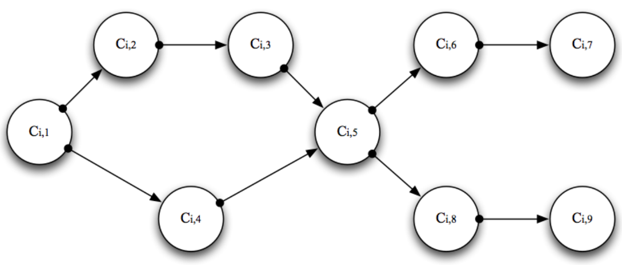


Figure 1. FPDS task with a DAG structure

for an example of the latter.

[9] presents a *rate-monotonic with delayed preemption (RMDP)* scheduling scheme. Compared to traditional rate-monotonic scheduling, RMDP reduces the number of context switches (due to strict preemption) and system calls (for locking shared data). One of the two preemption policies proposed for RMDP is *delayed preemption*, in which the computation time C_i for a task is divided into fixed size quanta c_i, with preemption of the running task delayed until the end of its current quanta. [9] provide the accompanying utilization based analysis and simulation results, and show an increased utilization of up to 8% compared to traditional rate-monotonic scheduling with context switch overheads.

Unlike [9], which introduces preemption points at fixed intervals corresponding to the quanta c_i, our approach allows to insert preemption points at arbitrary intervals, convenient for the tasks.

[3, 4] correct the existing worst-case response time analysis for FPDS, under arbitrary phasing and deadlines smaller or equal to periods. They observe that the critical instance is not limited to the first job, but that the worst case response time of task τ_i may occur for an arbitrary job within an i-level active period. They provide an exact analysis, which is not uniform (i.e. the analysis for the lowest priority task differs from the analysis for other tasks) and a pessimistic analysis, which is uniform.

The need for FPDS in industrial real-time systems is emphasized in [10], which aims at combining FPDS with reservations for exploiting the network bandwidth in a multimedia processing system from the surveillance domain, in spite of fluctuating network availability. It describes a system of one of our industry partners, monitoring a bank office. A camera monitoring the scene is equipped with an embedded processing platform running two tasks: a video task processing the raw video frames from the camera, and a network task transmitting the encoded frames over the network. The video task encodes the raw frames and analyses the content with the aim of detecting a robbery. When a robbery is detected the network task transmits the encoded frames over the network (e.g. to the PDA of a police officer). In data intensive applications, such as video processing, a context switch can be expensive: e.g. an interrupted DMA transfer may need to retransmit the data when the transfer is resumed. Currently, in order to avoid the switching overhead due to arbitrary preemption, the video task is non-preemptive. Consequently, the network task is activated only after a complete frame was processed. Often the network task cannot transmit packets at an arbitrary moment in time (e.g. due to network congestion). Employing FPDS and inserting preemption points in the video task in convenient places will activate the network task more frequently than is the case with FPNS, thus limiting the switching overhead compared to FPPS and still allowing exploitation of the available network bandwidth.

[10] also propose the notion of *optional preemption points*, allowing a task to check if a higher priority task is pending, which will preempt the current task upon the next preemption point. At an optional preemption point a task cannot know if a higher priority task will not arrive later, however if a higher priority task is already pending, then the running task may decide to adapt its execution path, and e.g. refrain from initiating a data transfer on a exclusive resource that is expensive to interrupt or restart. Optional preemption points rely on being able to check for pending tasks with low overhead, e.g. without invoking the scheduler.

3 RTAI

RTAI[2] is an extension to the Linux kernel, which enhances it with hard real-time scheduling capabilities and primitives for applications to use this. RTAI provides hard real-time guarantees alongside the standard Linux operating system by taking full control of external events generated by the hardware. It acts as a *hypervisor* between the hardware and Linux, and intercepts all hardware interrupt handling. Using the timer interrupts RTAI does its own scheduling of real-time tasks and is able to provide hard timeliness guarantees.

Although RTAI has support for multiple CPUs, we choose to ignore this capability in the remainder of this

[2]The code base used for this work is version 3.6-cv of *RTAI* [1].

document, and assume that our FPDS implementation is running on single-CPU platforms.

3.1 The scheduler

RTAI Linux system follows a *co-scheduling* model: hard real-time tasks are scheduled by the RTAI scheduler, and the remaining idle time is assigned to the normal Linux scheduler for running all other Linux tasks. The RTAI scheduler supports the standard Linux *schedulables* such as (user) process threads and kernel threads, and can additionally schedule RTAI kernel threads. These have low overhead but they cannot use regular OS functions.

The scheduler implementation supports preemption, and ensures that always the highest priority runnable real-time task is executing.

Primitives offered by the RTAI scheduler API include periodic and non-periodic task support, multiplexing of the hardware timer over tasks, suspension of tasks and timed sleeps. Multiple tasks with equal priority are supported but need to use cooperative scheduling techniques (such as the `yield()` function that gives control back to the scheduler) to ensure fair scheduling.

3.2 Tasks in RTAI

RTAI supports the notion of tasks along with associated priorities. Tasks are instantiated by creating a schedulable object (typically a thread) using the regular Linux API, which can then initialize itself as an RTAI task using the RTAI specific API. Priorities are 16 bit integers with 0 being the highest priority.

Although the terminology of *jobs* is not used in RTAI, all necessary primitives to support periodic tasks with deadlines less than or equal to periods are available. Repetitive tasks are typically represented by a thread executing a repetition, each iteration representing a job. An invocation of the `rt_task_wait_period()` scheduling primitive separates successive jobs. Through a special return value of this function, a task will be informed if it has already missed the time of activation of the next job, i.e. the deadline equal to the period.

In each task control block (TCB) various properties and state variables are maintained, including a 16 bit integer variable representing the running state of the task. Three of these bits are used to represent mutually exclusive running states (*ready*, *running*, *blocked*), whereas the remaining bits are used as boolean flags that are not necessarily mutually exclusive, such as the flag *delayed* (waiting for the next task period), which

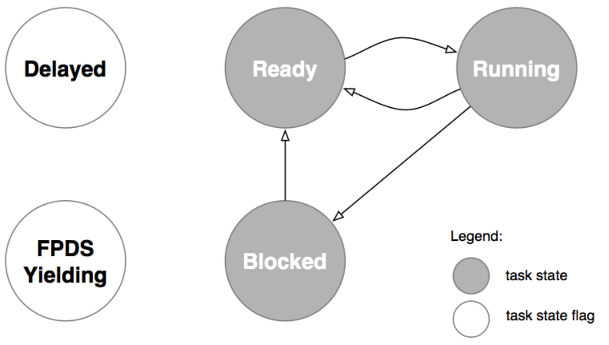


Figure 2. RTAI task states and flags

can be set at the same time as *ready* in the RTAI implementation. This implies that testing the *ready* state is not sufficient for determining the readiness of a task. See Figure 2 for an overview of the task states relevant for our work, including a new bit flag *FPDS Yielding* which we will introduce for our FPDS implementation in Section 5.4.

3.3 Scheduler implementation

In order to provide some context for the design decisions and implementation considerations that will follow, we briefly describe the implementation of the existing RTAI scheduler.

RTAI maintains a *ready queue* per CPU, as a *priority queue* of tasks that are ready to run (i.e., *released*), sorted by task priority. Periodic tasks are maintained with release times of their next job in a separate data structure, the so-called *timed tasks*. This data structure can be an ordered linked list or a red-black tree. If at any moment the current time passes the release time of the head element of the timed tasks list, the scheduler migrates this task to the ready queue of the current CPU. In practice this does not happen instantly but only upon the first subsequent invocation of the scheduler, e.g. through the timer interrupt, and therefore having a maximum latency equal to the period of the timer. The scheduler then selects the head element from the ready priority queue for execution, which is the highest priority task ready to run. The currently running task will be preempted by the newly selected task if it is different. The scheduler ensures that at any given time, the processor executes the highest priority task of all those tasks that are currently ready to execute, and therefore it is a FPPS scheduler.

The implementation of the scheduler is split over two main scheduler functions, which are invoked from different contexts, but follow a more or less similar structure. The function `rt_timer_handler()` is called from within the timer interrupt service routine, and is therefore time-triggered. The other functions, `rt_schedule()` is event-triggered, and performs scheduling when this is requested from within a system call. Each of the scheduler functions performs the following main steps:

1. Determination of the current time
2. Migration of runnable tasks from the timed tasks queue to the ready queue
3. Selection of the highest priority task from the ready queue
4. Context switch to the newly selected task if it is different from the currently running task

After a new task is selected, the scheduler decides on a context switch function to use, depending on the type of tasks (kernel or user space) being switched in and out. The context switch is then performed immediately by a call to this function.

4 Mapping FPDS tasksets

For the case of FPDS we need a different formulation of the taskset. This is because we now must indicate additional subtask structure within each task. There are several ways to approach this.

First, in the task specification we can mark subtask completion by an API call. Many operating systems already implement a primitive that can be used for this, viz., a `yield()` function. In fact, in a cooperative scheduling environment this would be exactly the way to implement FPDS. When a currently running task calls `yield()`, it signals the kernel that it voluntarily releases control of the CPU, such that the scheduler can choose to activate other tasks before it decides to return control to the original task, according to the scheduler algorithm. For the current case of RTAI we would need to modify `yield()` since it currently performs just cooperative scheduling among tasks of the same priority and we would need to ensure that tasks cannot be preempted outside `yield()` functions when in *ready* state.

Second, we can simply use the regular task model for the subtasks. However, this would imply significant overhead in the form of subtask to subtask communication, because the subtasks need to cooperate to maintain the precedence constraints while scheduling these subtasks, which are otherwise implied within the execution of a single task.

Finally, we can develop special notation for this purpose by special data structures and interaction points to be filled in by the user. This, however, would probably not differ a lot from the first case. The advantage would be that, unlike in the first two approaches, the kernel would be aware of the details about the subtask structure which is important for internal analysis by the system, for monitoring or for optimization.

In the first case the API calls play the role of explicit preemption points. These can be programmed directly, but also automated tools could generate preemption points transparently to the programmer guided by other primitives and cues in the source code such as critical sections. Moreover, the `yield()` approach incurs low overhead and limits the modifications to the kernel. We therefore decide for the first approach.

5 Design and implementation

While designing the implementation of our chosen FPDS task model, we have a number of aspects that lead our design choices. First of all, we want our implementation to remain *compatible*; our extensions should be conservative and have no effect on the existing functionality. Any FPDS tasks will need to explicitly indicate desired FPDS scheduling behaviour. *Efficiency* is important because overhead should be kept minimal in order to maximize the schedulability of task sets. Therefore we aim for an FPDS design which introduces as little run-time and memory overhead as possible. Due to the need of keeping time, we do not disable interrupts during FPDS tasks, so the overhead of *interrupt handling* should be considered carefully as well. Because we want to be able to integrate our extensions with future versions of the platform, our extensions should be *maintainable*, and written in an *extendable* way, with flexibility for future extensions in mind.

We aim for a FPDS implementation that is non-preemptive only with respect to other tasks; i.e. a task will not be *preempted* by another task, but can be *interrupted* by an interrupt handler such as the timer ISR.

The process of implementing FPDS in RTAI/Linux was done in several stages. Because the existing scheduler in RTAI is an FPPS implementation with no direct support for non-preemptive tasks, the first stage consisted of a proof of concept attempt at implementing FPNS in RTAI. The following stages then built upon this result to achieve FPDS scheduling in RTAI in accordance with the task model and important design aspects described above.

5.1 FPNS design

The existing scheduler implementation in RTAI is FPPS: it makes sure that at every moment in time, the highest priority task that is in *ready* state has control of the CPU. In contrast, FPNS only ensures that the highest priority ready task is started upon a job finishing, or upon the arrival of a task whenever the CPU is idle. For extending the FPPS scheduler in RTAI with support for FPNS, the following extensions need to be made:

- Tasks, or individual jobs, need a method to indicate to the scheduler that they need to be run non-preemptively, as opposed to other tasks which may want to maintain the default behaviour.
- The scheduler needs to be modified such that any scheduling and context switch activity is deferred until a non-preemptive job finishes.

Alternatively, arrangements can be made such that at no moment in time a ready task exists that can preempt the currently running FPNS task, resulting in a schedule that displays FPNS behaviour, despite the scheduler being an FPPS implementation. Both strategies will be explored.

5.1.1 Using existing primitives

Usage of the existing RTAI primitives for influencing scheduling behaviour to achieve FPNS would naturally be beneficial for the *maintainability* of our implementation.

When investigating the RTAI scheduler primitives exported by the API [12], we find several that can be used to implement FPNS behaviour. These strategies range from the blocking of any (higher priority) tasks during the execution of a FPNS job, e.g. through suspension or mutual exclusion blocking of these tasks, to influencing the scheduler decisions by temporary modifications of task priorities. What they have in common however is that at least 2 invocations of these primitives are required per job execution, resulting in RTAI in additional overhead of at least two system calls per job. Some of these methods, such as explicit suspension of all other tasks, also have the unattractive property of requiring complete knowledge and cooperation of the entire task set.

5.1.2 RTAI kernel modifications

As an alternative to using existing RTAI primitives which are not explicitly designed to support FPNS, the notion of a non-preemptible task can be moved into the RTAI kernel proper, allowing for modified scheduling behaviour according to FPNS, without introducing extra overhead during the running of a task as induced by the mentioned API primitives. Looking ahead to our goal of implementing FPDS, this also allows more fine grained modifications to the scheduler itself, such that *optional* preemption points become possible in an efficient manner: rather than trying to disable the scheduler during an FPNS job, or influencing its decisions by modifying essential task parameters such as priorities, the scheduler would become aware of non-preemptible or deferred preemptible tasks and support such a schedule with intelligent decisions and primitives. It does however come at the cost of implementation and maintenance complexity. Without availability of documentation of the design and implementation of the RTAI scheduler, creating these extensions is more difficult and time consuming than using the well documented API. And because the RTAI scheduler design and implementation is not stable, as opposed to the API, continuous effort will need to be spent on maintaining these extensions with updated RTAI source code, unless these extensions can be integrated into the RTAI distribution. Therefore we aim for a patch with a small number of changes to few places in the existing source code.

An important observation is that with respect to FPPS, scheduling decisions are only made differently during the execution of a non-preemptive task. Preemption of any task must be initiated by one of the scheduling functions, which means that one possible implementation of FPNS would be to alter the decisions made by the scheduler if and only if a FPNS task is currently executing. This implies that our modifications will be *conservative* if they change scheduling behaviour during the execution of non-preemptive tasks only.

During the execution of a (FPNS) task, interference from other, higher priority tasks is only possible if the scheduler is invoked through one of the following ways:

- The scheduler is invoked from within the timer ISR
- The scheduler is invoked from, or as a result of a system call by the current task

The first case is mainly used for the release of jobs - after the timer expires, either periodically or one-shot, the scheduler should check whether any (periodic) tasks should be set *ready*, and then select the highest priority one for execution. The second case applies when a task does a system call which alters the state of the system in such a way that the schedule *may* be affected, and thus the scheduler should be called to determine this. With pure FPNS, all scheduling work can be deferred until the currently running task finishes execution. Lacking any optional preemption points, under no circumstances should the current FPNS task be preempted. Therefore, the condition of a currently running, ready FPNS task should be detected as early on in the scheduler as possible, such that the remaining scheduling work can be deferred until later, and the task can resume execution as soon as possible, keeping the overhead of the execution interruption small.

In accordance with our chosen task model in Section 4, we decide to modify the kernel with explicit non-preemptive task support, as described in section 5.1.2.

5.2 FPNS implementation

Towards an FPNS aware RTAI scheduler, we extend the API with a new primitive named `rt_set_preemptive()`, consistent with other primitives that can alter parameters of tasks, that accepts a boolean parameter indicating whether the calling task should be preemptible, or not. This value will then be saved inside the task's control block (TCB) where it can be referenced by the scheduler when making scheduling decisions. This *preemptible* flag inside the TCB only needs to be set once, e.g. during the creation of the task and not at every job execution, such that there is no additional overhead introduced by this solution.

Execution of the scheduler should be skipped at the beginning, if the following conditions hold for the currently running task:

- The (newly added) *preemptible* boolean variable is unset, indicating this is a non-preemptive task;
- The *delayed* task state flag is not set, indicating that the job has not finished;
- The *ready* task state flag is set, indicating that the job is ready to execute and in the ready queue.

We have added these tests to the start of both scheduler functions `rt_schedule()` and `rt_timer_handler()`, which resulted in the desired FPNS scheduling for non-preemptible tasks. For preemptive tasks, which have the default value of 1 in the *preemptible* variable of the TCB, the scheduling behaviour is not modified, such that the existing FPPS functionality remains unaffected.

5.3 FPDS design

Following the description of FPNS in the previous section, we move forward to the realisation of a FPDS scheduler, by building upon these concepts. An FPDS implementation as outlined in Section 4, where subtasks are modeled as non-preemptive tasks with preemption points in between, has the following important differences compared to FPNS:

- A job is not entirely non-preemptive anymore; it may be preempted at predefined preemption points, for which a scheduler primitive needs to exist.
- The scheduling of newly arrived jobs can no longer be postponed until the end of a non-preemptive job execution, as during a (optional) preemption point in the currently running job information is required about the availability of higher priority ready jobs.

There are several ways to approach handling interrupts which occur during the execution of nonpreemptive subjob. First, the interrupt may be recorded with all scheduling postponed until the scheduler invocation from `yield()` at the next preemption point, similar to our FPNS implementation, but at much finer granularity.

Alternatively, all tasks which have arrived can be moved from the pending queue to the ready queue directly (as is the case under FPPS), with only the context switch postponed until the next preemption point. This has the advantage that there is opportunity for optional preemption points to be implemented, if the information about the availability of a higher priority, waiting task can be communicated in an efficient manner to the currently running FPDS task for use at the next optional preemption point. These two alternatives can be described as *pull* versus *push* models respectively. They represent a tradeoff, and the most efficient model will most likely depend on both the task sets used, and the period of the timer.

On our platform we could not measure the difference between these two alternatives; any difference in efficiency between the two approaches was lost in the noise of our experiment. Therefore we opted to go with the last alternative, as this would not require extensive rewriting of the existing scheduler logic in RTAI, and thereby fit our requirements of *maintainability* and our extensions being *conservative.* The efficiency differences between these approaches may however be relevant on other platforms, as described in [10], based on [8].

5.4 FPDS implementation

It would appear that using a standard `yield()` type function, as present within RTAI and many other operating systems, would suffice for implementing a preemption point. Upon investigation of RTAI's yield function (`rt_task_yield()`) it turned out however that it could not be used for this purpose unmodified. This function is only intended for use with round-robin scheduling between tasks having equal priority, because under the default FPPS scheduler of RTAI there is no reason why a higher priority, ready task would not already have preempted the current, lower priority task. However with the non-preemptive tasks in FPDS, a higher priority job may have arrived but not been context switched in, so checking the ready queue for *equal* priority processes is not sufficient. An unconditional call to scheduler function `rt_schedule()` should have the desired effect, as it can move newly arrived tasks to the ready queue, and invoke a preemption if necessary. However, the modified scheduler will evaluate the currently running task as non-preemptive, and avoid a context switch. To indicate that the scheduler is being called from a preemption point and a higher priority task is allowed to preempt, we introduce a new bit flag `RT_FPDS_YIELDING` to the task state variable in the TCB, that is set before the invocation of the scheduler to inform it about this condition. The flag is then reset again after the scheduler execution finishes.

Due to the different aim of our FPDS yield function in comparison to the original yield function in RTAI we decided not to modify the existing function, but create a new one specific for FPDS use instead: `rt_fpds_yield()`. The FPDS yield function is simpler and more efficient than the regular yield function, consisting of just an invocation of the scheduler function wrapped between the modification of task state flags. This also removed the need to modify the existing code which could introduce unexpected regressions with existing programs, and have a bigger dependency on the existing code base, implying greater overhead in maintaining these modifications in the future.

5.4.1 Scheduler modifications

Working from the FPNS implementation of Section 5.1, the needed modifications to the scheduling functions `rt_schedule()` and `rt_timer_handler()` for FPDS behaviour are small. Unlike the FPNS case, the execution of the scheduler cannot be deferred until the completion of a FPDS (sub)task if we want to use the push mechanisms described in Section 5.3, as the scheduler needs to finish its run to acquire this information. Instead of avoiding the execution of the scheduler completely, for FPDS we only defer the invocation of a context switch to a new task, if the currently running task is not at a preemption point.

The existing `if` clause introduced at the start of the scheduler functions for FPNS is therefore moved to a section in the scheduler code between parts 3 and 4 as described in Section 3.3, i.e. at which point the scheduler has decided a new task should be running, but has not started the context switch yet. At this point we set a newly introduced TCB integer variable `should_yield` to *true*, indicating that the current task should allow itself to be preempted at the next preemption point. This variable is reset to *false* whenever the task is next context switched back in.

With these modifications, during a timer interrupt or explicit scheduler invocation amidst a running FPDS task, the scheduler will be executed and wake up any timed tasks. If a higher priority task is waiting at the head of the ready queue, a corresponding notification will be delivered and the scheduler function will exit without performing a context switch. To allow an FPDS task to learn about the presence of a higher priority task waiting to preempt it, indicated by the *should_yield* variable in its TCB in kernel space which it however does not have permissions for to read directly, we introduced a new RTAI primitive `rt_should_yield()` to be called by the real-time task, which returns the value of this variable. In the current implementation this does however come at the cost of performing a system call at every preemption point.

In a later version of our FPDS implementation in RTAI, not yet reflected in the measurements in this paper, we improved the efficiency of preemption points by removing the need for this system call. The location of the *should_yield* variable was moved from the TCB in kernel space to user space in the application's address space, where it can be read efficiently by the task. Upon initialization, the FPDS task registers this variable with the kernel, and makes sure the corresponding memory page is locked into memory. The contents of this variable are then updated exclusively by the kernel, which makes use of the fact that updates are only necessary during the execution of the corresponding FPDS task, when its address space is already active and loaded in physical memory. This design is similar to the one described in [2] for implementing non-preemptive sections.

5.5 Abstraction

For *simplicity* of usage of our FPDS implementation, we created a dynamic library called *libfpds* which can be linked to a real-time application that wants to use FPDS. A preemption point can then be inserted into the code by inserting an invocation of `fpds_pp()`, which internally performs `rt_should_yield()` and `fpds_yield()` system calls as necessary. Alternatively, the programmer can pass a parameter to indicate that the task should not automatically be preempted if a higher task is waiting, but should merely be informed of this fact.

6 Key performance indicators

Our implementation should be checked for the following elements, which relate to the design aspects mentioned in Section 5:

- The interrupt latency for FPDS. This is intrinsic to FPDS, i.e. there is additional blocking due to lower priority tasks. It has been dealt with in the analysis in [3, 4];
- The additional *run-time* overhead due to additional code to be executed. This will be measured in Section 7;
- The additional *space* requirements due to additional data structures and flags. Our current implementation introduces only two integer variables to the TCB, so the space overhead is minimal;
- The number of *added*, *changed*, and *deleted* lines of code (excluding comments) compared to the original RTAI version. Our extension adds only 106 lines and modifies 3 lines of code, with no lines being removed;
- The *compatibility* of our implementation. Because our extensions are conservative, i.e. they don't change any behaviour when there are no non-preemptive tasks present, compatibility is preserved. This is also verified by our measurements in Section 7.1.

7 Measurements

We performed a number of experiments to measure the additional overhead of our extensions compared to the existing FPPS scheduler implementation. The hardware used for these tests was an Intel Pentium 4 PC, with 3 Ghz CPU, running Linux 2.6.24 with (modified) RTAI 3.6-cv.

7.1 Scheduling overhead

The goal of our first experiment is to measure the overhead of our implementation extensions for existing real-time task sets, which are scheduled by the standard RTAI scheduler, i.e. following FPPS. For non-FPDS task sets, scheduling behaviour has not been changed by our conservative implementation, but our modifications may have introduced additional execution overhead.

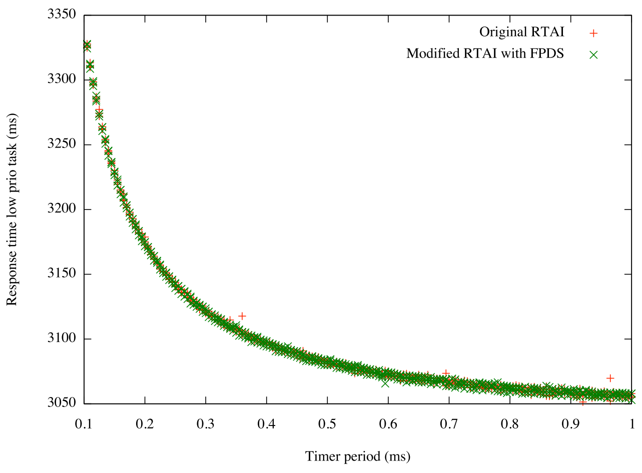


Figure 3. Overhead of the modified kernel for FPPS task sets

As our example task set we created a program with one non-periodic, low priority, long running FPPS task τ_l, and one high priority periodic FPPS task τ_h. τ_l consists of a `for` loop with a parameterized number of iterations m, to emulate a task with computation time C_l. The computation time of the high priority task, C_h, was 0; the only purpose of this empty task is to allow for measurement of overhead of scheduling by presenting an alternative, higher priority task to the scheduler. T_h was kept equal to the period of the timer, such that a new high priority job is released at every scheduler invocation from the timer event handler.

Since, from the perspective of an FPPS task, the only modified code that is executed is in the scheduler functions, we measured the response time of task τ_l under both the original and the modified FPDS real-time RTAI kernel, varying the period of the timer interrupt, and thereby the frequency of scheduler invocations encapsulated by the timer event handler. The results are shown in Figure 3.

As expected, there is no visible difference in the overhead of the scheduler in the modified code compared to the original, unmodified RTAI kernel. For an FPPS task set the added overhead is restricted to a single `if` statement in the scheduler, which references 3 variables and evaluates to *false*. This overhead is unsubstantial and lost in the noise of our measurements. We conclude that there is no significant overhead for FPPS task sets introduced by our FPDS extensions.

7.2 Preemption point overhead

With an important aspect of FPDS being the placement of preemption points in task code between subtasks, the overhead introduced by these preemption points is potentially significant. Depending on the frequency of preemption points, this could add a substantial amount of additional computation time to the FPDS task. In the tested implementation, the dominating overhead term is expected to be the overhead of a system call (C_{sys}) performed during every preemption point, to check whether the task should yield for a higher priority task. The system call returns the value of a field in the TCB, and performs no additional work.

We measured the overhead of preemption points by creating a long-running, non-periodic task τ_l with fixed computation time C_l implemented by a `for` loop with $m = 100M$ iterations, and scheduled it under both FPPS and FPDS. The division into subtasks of task τ_l has been implemented by invoking a preemption point every n iterations, which is varied during the course of this experiment, resulting in $\lceil m/n \rceil$ preemption point invocations.

For the FPPS test the same task was used, except that every n iteration interval only a counter variable was increased, instead of the invocation of a preemption point. This was done to emulate the same low priority task as closely as possible in the context of FPPS.

The response time R_l was measured under varying intervals of n for both FPPS and FPDS task sets. The results are plotted in Figure 4.

Clearly the preemption points introduced in the lower priority task introduce overhead which does not exist in a FPPS system. The extra overhead amounts to about 440 μs per preemption point invocation, which corresponds well with a measured value C_{sys} of 434 μs per general RTAI system call overhead which we obtained in separate testing. This suggests that the overhead of a preemption point is primary induced by the `rt_should_yield()` system call in the preemption point implementation, which is invoked unconditionally.

7.3 An example FPDS task set

Whereas the previous experiments focussed on measuring the overhead of the individual extensions and primitives added for our FPDS implementation, we performed an experiment to compare the worst case response time of a task set under FPPS and FPDS as well. The task set of the previous experiment was extended with a high priority task τ_h with a non-zero computation time C_h. For this experiment we varied the period of the high priority task. To keep the workload of the low priority task constant, we fixed the interval n of preemption points to a value (5000) frequent enough to allow preemption by τ_h without it missing any deadlines under all values of $T_h \geq 1.2ms$ under test. The response time of the low priority task R_l is plotted in Figure 5.

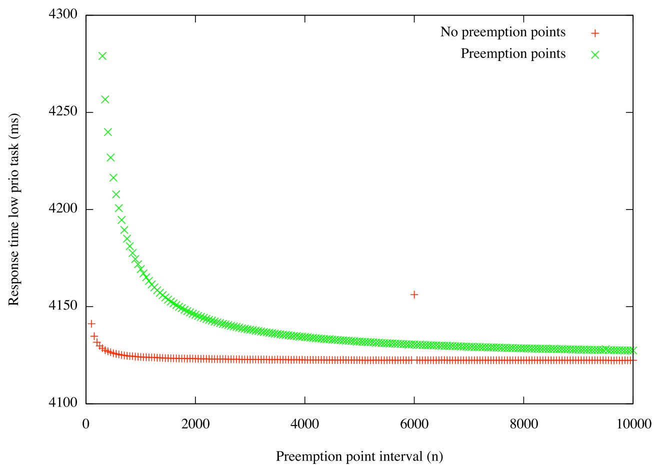


Figure 4. Overhead of preemption points

The relative overhead appears to depend on the frequency of high priority task releases and the resulting preemptions in preemption points, as the number of preemption points invoked in the duration of the test is constant. The results show an increase in the response time of τ_l for FPDS of at most 17% with a mean around 3%. The large discrepancy of the results can probably be attributed to the unpredictable interference from interrupts and the resulting invalidation of caches. Considering the relatively low mean overhead of FPDS, we would like to identify the factors which contribute to the high variation of the task response time, and investigate how these factors can be eliminated (see Section 8).

8 Conclusions and future work

In this paper we have presented our work on the implementation of FPDS in the real-time operating system, RTAI/Linux. We have shown that such an implementation in a real-world operating system is feasible, with only a small amount of modifications to the existing code base in the interest of future maintainability. Furthermore, a set of experiments indicated that our modifications introduced no measurable overhead for

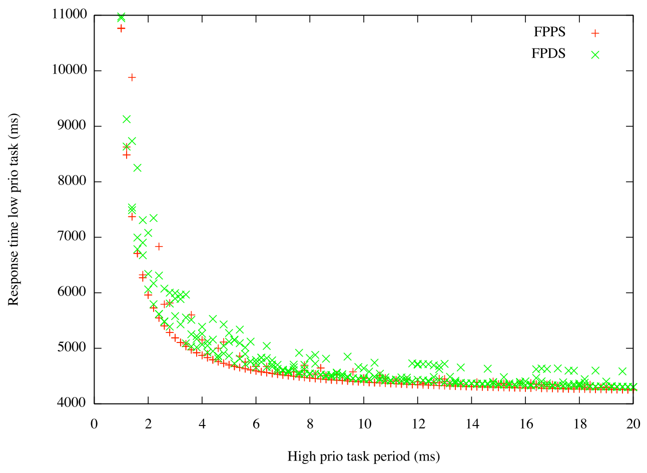


Figure 5. A task set scheduled by FPPS and FPDS

FPPS task sets, and only small mean overhead introduced by converting a FPPS task set into FPDS.

As a follow up to this work, we would like to further investigate the tradeoffs between regular preemption points and optional preemption points. In particular we would like to gain quantitive results exposing the tradeoffs in moving tasks to the ready queue during the execution of a non-preemptive subjob, with respect to saved system calls, invalidation of caches and other overheads.

Finally, we would like to research how additional information about FPDS task structures in the form of a DAG can benefit the scheduling decisions. A DAG will specify the worst-case computation times of subtasks and thus form a sort of contract between the tasks and the scheduler, allowing to combine FPDS with reservations. Methods for monitoring and enforcing these contracts need to be investigated.

References

[1] RTAI 3.6-cv - The RealTime Application Interface for Linux from DIAPM, 2009.

[2] B. B. Brandenburg, J. M. Calandrino, A. Block, H. Leontyev, and J. H. Anderson. Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin? In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 342–353. IEEE Computer Society, 2008.

[3] R. Bril, J. Lukkien, and W. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited. In *Proc. 19th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 269–279, July 2007.

[4] R. Bril, J. Lukkien, and W. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited – with extensions for ECRTS'07 –. Technical Report CS Report 07-11, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven (TU/e), The Netherlands, April 2007.

[5] A. Burns. Preemptive priority based scheduling: An appropriate engineering approach. In S. Son, editor, *Advances in Real-Time Systems*, pages 225–248. Prentice-Hall, 1994.

[6] A. Burns. Defining new non-preemptive dispatching and locking policies for ada. *Reliable SoftwareTechnologies —Ada-Europe 2001*, pages 328–336, 2001.

[7] A. Burns, M. Nicholson, K. Tindell, and N. Zhang. Allocating and scheduling hard real-time tasks on a parallel processing platform. Technical Report YCS-94-238, University of York, UK, 1994.

[8] A. Burns, K. Tindell, and A. Wellings. Effective analysis for engineering real-time fixed priority schedulers. *IEEE Transactions on Software Engineering*, 21(5):475–480, May 1995.

[9] R. Gopalakrishnan and G. M. Parulkar. Bringing real-time scheduling theory and practice closer for multimedia computing. *SIGMETRICS Perform. Eval. Rev.*, 24(1):1–12, 1996.

[10] M. Holenderski, R. J. Bril, and J. J. Lukkien. Using fixed priority scheduling with deferred preemption to exploit fluctuating network bandwidth. In *ECRTS '08 WiP: Proceedings of the Work in Progress session of the 20th Euromicro Conference on Real-Time Systems*, 2008.

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

[12] G. Racciu and P. Mantegazza. *RTAI 3.4 User Manual, rev 0.3*, 2006.

Hierarchical Multiprocessor CPU Reservations for the Linux Kernel[*]

Fabio Checconi, Tommaso Cucinotta, Dario Faggioli, Giuseppe Lipari
Scuola Superiore S. Anna, Pisa, Italy

Abstract

This paper presents ongoing work in the development of a scheduling framework that will improve the service guarantees for soft real-time applications deployed on Linux. The scheduler has been designed around the current kernel infrastructure, trying to keep the changes minimal, and basing the scheduling policy on strong theoretical results. The main goal is to achieve hierarchical distribution of the available computing power on multiprocessor platforms, avoiding alterations to the existing user interfaces.

The proposed framework exploits the hierarchical arrangement of tasks within groups and subgroups that is already possible within the Linux kernel. However, it adds the capability for each group to be assigned a precise fraction of the computing power available on all the processors, using existing uni-processor resource reservation techniques. Tasks are scheduled globally within each single group, and the partitions assigned to each group need not to be static, but can be dynamically balanced. Furthermore, the proposed mechanism can be used to support a variety of possible partitioning schemes using processor affinities.

1 Introduction

Nowadays, the Linux Operating System is being enriched with more and more real-time capabilities. In the last few years, valuable efforts have been spent for decreasing the scheduling and interrupt latencies of the kernel, by embedding such features as full preemption, priority inheritance, reduced computation complexity of the scheduler, support for high-resolution timers. Also, the `linux-rt` branch adds such experimental features as running interrupt handlers in dedicated kernel threads rather than in interrupt context, so as to allow system designers to have an improved control over the interference of the peripheral drivers with respect to the running applications.

While such features make the Linux kernel a very appealing platform for multimedia applications, still the support for real-time scheduling is somewhat inappropriate for dealing with requirements posed by the challenging scenarios of the upcoming years, that demand for predictable scheduling mechanisms able to achieve a good degree of temporal isolation among complex concurrent software components, low response times and high interactivity. One such scenario is the one in which multiple virtual machines run within the same OS, hosting software components realizing professional services that need to run with predictable QoS levels and high interactivity requirements, possibly managed through a service-oriented approach, as discussed for example in [1].

The Linux kernel embodies the POSIX compliant priority-based real-time scheduling classes (`SCHED_FIFO` and `SCHED_RR`). These may be sufficient for dealing with embedded real-time applications, but they turn out to be inadequate for providing temporal isolation among complex software components such as the ones mentioned above. In fact, the implementation of such policies in Linux has been enriched by non-standard features such as support for hierarchies of tasks and *throttling*. However, lacking of a sound design in the domain of real-time scheduling, such capabilities struggle at constituting a solid base for providing an adequate real-time scheduling support.

This paper makes one step further in this direction, presenting a novel real-time scheduling strategy for the Linux kernel, that may be analyzed by means of hierarchical real-time schedulability analysis techniques. The proposed infrastructure has a good degree of flexibility, allowing for a variety of configurations between two traditionally antithetic settings: on one side, the perfect compatibility with the current POSIX compliant priority-

[*]The research leading to these results has been supported by the European Commission under grant agreement n.214777, in the context of the IRMOS Project. More information at: http://www.irmosproject.eu.

based semantics, and on the other side an improved usage of resources by means of a partitioned EDF.

1.1 Paper Contributions

This paper presents a hierarchical multiprocessor scheduling framework for the Linux kernel. The main advantages of the presented approach over prior works are:

- tight integration with the existing Linux code;
- no need for the introduction of new interfaces nor new scheduling classes;
- support for multiple configuration schemes, including fully partitioned approaches;
- strong theoretical background justifying the relevance of the approach, mainly inspired to [2], with the derivation of an appropriate admission test for the tasks to be scheduled;
- capability to handle accesses to shared resources.

1.2 Paper Outline

The rest of the paper is organized as follows. Section 2 reviews related work in the area, then Section 3 introduces considered system model and scheduling algorithm, summarizing its formal properties. Section 4 describes the implementation of the framework in the Linux kernel, and Section 5 reports experimental results that validate the approach. Finally, Section 8 contains a few concluding remarks.

2 Related Work

The growing interest in having more advanced real-time scheduling support within the Linux kernel has been witnessed in the last years by various research projects. The first approach that has been undertaken has been the addition of a hypervisor to the Linux kernel, so as to obtain a highly predictable hard real-time computing platform where real-time control tasks are scheduled very precisely, and the entire Linux OS is run in the background. Such an approach, adopted in the RTLinux [3] and RTAI [4] projects, however is not adequate for interactive nor multimedia applications, due to the high limitations it poses on the services available to real-time applications.

An alternative trend is constituted by the addition of a (soft) real-time scheduling policy directly within the Linux kernel, that allows for a more predictable execution of unmodified Linux applications. Projects that fall in this category comprise the following.

The Adaptive Quality of Service Architecture [5] (`AQuoSA`) for Linux provides hard CBS [6], an EDF based real-time policy, which has also been enhanced with the Bandwidth Inheritance protocol [7] for dealing with shared resources. However, having been developed in the context of the FRESCOR [1] European Project for embedded systems, `AQuoSA` suffers from the main limitation of not supporting SMP systems.

The $Litmus^{RT}$ project [8, 9] provides (among others) Pfair [10], a real-time scheduling strategy theoretically capable of saturating SMP systems with real-time tasks. However, it contains major changes of the Linux kernel internals, and it is currently more a testbed for experimenting with real-time scheduling within Linux, rather than something that aims at being integrated in the mainline kernel.

Recently, an implementation of the POSIX `SCHED_SPORADIC` [11] real-time policy for Linux has been proposed to the Linux kernel community [12]. This scheduler has been developed with the aim of being integrated into the mainstream kernel, by proposing a very limited set of modifications to the kernel scheduler, and exploiting existing user-space APIs such as the `cgroups`. The great advantage of such scheduling policy is the one of having been standardized by POSIX, however it suffers of the limitations typical of priority-based policies, such as the well-known utilization limit of 69% on uni-processor systems.

3 Scheduling Algorithm

The design of the scheduling algorithm started in a quite unusual way, analyzing the existing Linux scheduler, and trying to derive a formal model for the policy it is implementing for real-time scheduling, especially concerning the part of hierarchical scheduling. It turned out that the model in [2] is not far from matching the Linux implementation. The work we present in this paper aims to achieve a convergence between a hierarchical scheduling infrastructure that is minimally invasive as compared to the current Linux scheduler code base, and a theory of hierarchical real-time schedulers that is quite generic to be adapted to the Linux case.

We exploited the current user-space interface for the *throttling* mechanism, which offers to applications the possibility to assign a pair (Q_i, P_i) to the i-th group of tasks. However, these parameters are reinterpreted as the scheduling parameters (the budget and period, respectively) to be assigned to the group according to the well known resource reservation paradigm [6]: Q_i units of time are available to the group every period of length

[1] Framework for Real-Time embedded Systems based on Contracts (FRESCOR), European Project No. FP6/2005/IST/5-034026, more information at: http://www.frescor.org.

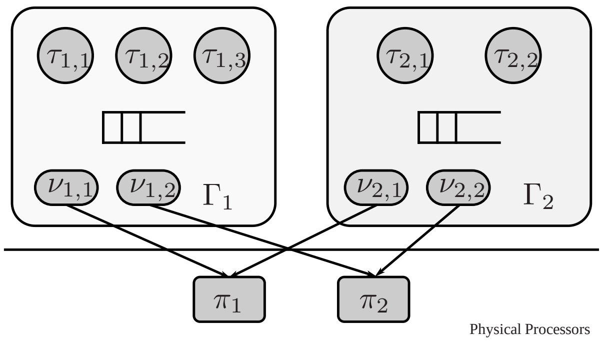


Figure 1: System Architecture.

P_i. The scheduling guarantee is given to each group as a whole, including all the tasks attached to the group itself and to all the nested subgroups. However, the framework allows each group and subgroup to posses its own set of scheduling parameters. On multiprocessor systems, the Q_i/P_i assignment is replicated on all the processors in the system, but the resulting schedulers on the various CPUs run independently from one another, minimizing synchronization overheads.

3.1 System Model and Terminology

Throughout the paper we stick to the Linux terminology as much as possible; when referring to entities that do not have a counterpart in the current Linux code yet, we derive our notation and terminology from [2].

In the model we consider, a task group Γ_i is composed by set of n_i sporadic tasks $\Gamma_i = \{\tau_{i,j}\}_{j=1,\ldots,n_i}$. Each task is described by its worst-case execution time $C_{i,j}$, its relative deadline $D_{i,j}$ and its minimum inter-arrival time $T_{i,j} : \tau_{i,j} = (C_{i,j}, D_{i,j}, T_{i,j})$. A task $\tau_{i,j}$ is a sequence of jobs $\tau^k_{i,j}$, each characterized by its own release time, computation time and deadline, denoted by $r^k_{i,j}, c^k_{i,j}$ and $d^k_{i,j}$, respectively.

Following [2], we call *virtual platform* $\mathcal{V}_i$ a set of m_i virtual processors $\mathcal{V}_i = \{\nu_{i,l}\}_{l=1,\ldots,m_i}$. Each virtual processor $\nu_{i,l}$ is characterized by a supply function $Z_{i,l}(t)$ representing the amount of service $\nu_{i,l}$ can provide in any time interval of duration t.

Tasks are grouped in task groups, organized in a hierarchical fashion. Each task group is assigned a virtual platform, one per physical processor $\pi_m \in \{\pi_m\}_{m=1,\ldots,M}$ in the system.

Fig. 1 depicts the global structure of our model, in the case of two physical processors, π_1 and π_2, and five tasks organised in two groups: $\tau_{1,1}, \tau_{1,2}, \tau_{1,3}$ inside Γ_1 and $\tau_{2,1}, \tau_{2,2}$ inside Γ_2; each task group is assigned two virtual processors, one for each physical processor.

3.2 Main Algorithm

The proposed algorithm can be described as a two-layer hierarchical scheduler, with the first layer scheduler selecting which task group to execute on each processor, and the second layer selecting which task to run within the selected task group.

Each task group Γ_i is assigned a set of virtual processors; these virtual processors are scheduled using partitioned resource reservation techniques. Each virtual processor is allocated a share of one of the physical processors in the system. The algorithm used to schedule virtual processors on physical processors is the Hard Constant Bandwidth Server (H-CBS) [6].

In other words, the first layer is composed by M independent partitioned H-CBS schedulers which manage all the virtual processors $\nu_{i,l}$ assigned to their respective physical processor. Looking again at Fig. 1, there are two H-CBS schedulers, one to schedule virtual processors running on π_1, and one for the ones running on π_2. The H-CBS on π_1 schedules the first virtual processors of the groups in the system ($\nu_{1,1}$ and $\nu_{2,1}$), while the H-CBS on π_2 schedules the second ones ($\nu_{1,2}$ and $\nu_{2,2}$).

Within each group tasks are kept in a global fixed-priority queue[2], and tasks belonging to the same task group are scheduled globally according to their priority. At every time instant, if a virtual platform $\mathcal{V}_i$ is in execution on m physical processors, then its m highest priority tasks are executing. Note that $m \leq M$ changes over time due to the asynchronous scheduling of virtual processors over the physical ones.

3.3 Formal Properties

The proposed scheduling strategy falls within the class of schedulers identified in the theoretical schedulability analysis framework presented in [2]. Therefore, for purposes related to schedulability analysis, the same system model and analysis techniques may be adopted, with an additional extension to support hierarchical scheduling.

When an arbitrary hierarchy is considered, the problem of scheduling an application Γ on a group with bandwidth allocated on multiple processors is reduced to the problem of scheduling Γ on the M$\alpha\Delta$ abstraction corresponding to the service provided by the given group. Known techniques can be used to derive the parameters for the M$\alpha\Delta$ abstraction representing the group.

In this section we present well-known results and adapt them to our framework; a schedulability test will be derived from Theorem 1 and Theorem 3 in [2].

[2]As Section 4 will explain, the global policy of the queue is implemented using per-processor queues.

3.3.1 The Supply Function

An abstraction to model the minimum CPU time provided in a given interval of time is the *supply function* [2, 13]. To introduce the supply function first we need the concept of *time partition*.

Definition 1 *A* time partition $\mathcal{P}$ *is a countable union of non-overlapping time intervals*

$$\mathcal{P} = \bigcup_{i \in \mathbb{N}} [a_i, b_i) \quad a_i < b_i < a_{i+1}. \tag{1}$$

Without loss of generality, we set the time when the first virtual processor starts in the system equal to 0.

Given a time partition $\mathcal{P}$, its supply function [2, 13] measures the minimum amount of CPU time provided by the partition in any time interval.

Definition 2 *Given a time partition $\mathcal{P}$, its* supply function $Z_{\mathcal{P}}(t)$ *is the minimum amount of CPU time provided by the partition in any time interval of length $t \geq 0$, i.e.,*

$$Z_{\mathcal{P}}(t) = \min_{t_0 \geq 0} \int_{\mathcal{P} \cap [t_0, t_0+t]} \mathrm{d}x. \tag{2}$$

Since, given a virtual processor ν, it is not possible to determine the time partition $\mathcal{P}$ it will provide, the above definition cannot be used in practice; the following two definitions generalize the considered time partition to all the possible partitions that can be generated by a virtual processor, and extend Def. 2 to be actually usable.

Definition 3 *Given a virtual processor ν,* $\mathsf{legal}(\nu)$ *is the set of time partitions $\mathcal{P}$ that can be allocated by ν.*

Definition 4 *Given a virtual processor ν, its* supply function $Z_\nu(t)$ *is the minimum amount of CPU time provided by the server ν in every time interval of length $t \geq 0$,*

$$Z_\nu(t) = \min_{\mathcal{P} \in \mathsf{legal}(\nu)} Z_{\mathcal{P}}(t). \tag{3}$$

A virtual processor ν implemented through a H-CBS with budget Q and period P, when active, conforms to the Explicit Deadline Periodic model [14] with deadline equal to the period. As a consequence, we can use the well-known supply function:

$$Z_\nu(t) = \max\{0, t - (k+2)(P-Q), kQ\}, \tag{4}$$

with $k = \left\lfloor \frac{t-P+Q}{P} \right\rfloor$.

3.3.2 The (α, Δ) Abstraction

A simpler abstraction, still able to model the CPU allocation[3] provided by a virtual processor, but using fewer parameters, and easier to derive is the "bounded delay partition," described by two parameters: a bandwidth α, and a delay Δ. The bandwidth α measures the rate at which an active virtual processor provides service, while the delay Δ represents the worst-case service delay.

The formal definitions of α and Δ, from [13], are given below.

Definition 5 *Given a virtual processor ν with supply function $Z_\nu(t)$, its* bandwidth α_ν *is defined as*

$$\alpha_\nu = \lim_{t \to \infty} \frac{Z_\nu(t)}{t}. \tag{5}$$

Definition 6 *Given a virtual processor ν with supply function $Z_\nu(t)$ and bandwidth α_ν, its* delay Δ_ν *is defined as*

$$\Delta_\nu = \sup_{t \geq 0} \left\{ t - \frac{Z_\nu(t)}{\alpha_\nu} \right\}. \tag{6}$$

Using the two definitions above, the supply function $Z_\nu(t)$ of a virtual processor ν can be lower bounded as follows:

$$Z_\nu(t) \leq \max\{0, \alpha_\nu(t - \Delta_\nu)\}, \tag{7}$$

which gives an intuitive definition of the (α, Δ) abstraction, as a way to extract a lower bound for the actual supply function of a virtual processor; α represents the share of the physical processor time assigned to the virtual processor, while Δ represents the responsiveness of the allocation. In the case of a H-CBS virtual processor of budget Q and period P, we have:

$$\alpha = \frac{Q}{P}, \quad \Delta = 2P - 2Q. \tag{8}$$

3.3.3 (α, Δ) Abstractions and Multiprocessors

In [2] an extension of the (α, Δ) abstraction for multiprocessors is given, along with the calculation of the (α, Δ) parameters for several algorithms described in literature.

Definition 7 *The Multi-(α, Δ) ($\mathsf{M}\alpha\Delta$) abstraction of a set $\mathcal{V} = \{\nu_j\}_{j=1,\dots,m}$ of virtual processors, represented by the m pairs $\{(\alpha_j, \Delta_j)\}_{j=1,\dots,m}$ is a multi-supply function defined by the set of supply functions $\{Z_{\nu_j} : Z_{\nu_j}(t) = \max(0, \alpha_j(t - \Delta_j))\}_{j=1,\dots,m}$.*

[3]The same abstraction does not apply to CPU time only [15], but here we consider only CPU time.

3.3.4 Schedulability Analysis

We consider the schedulability of a single task group Γ (composed of n tasks) over a set $\mathcal{V} = \{\nu_j\}_{j=1,\ldots,m}$ of virtual processors, with supply functions $Z_j(t) = Z_{\nu_j}(t)$. First, assuming to know the time partition $\mathcal{P}_j$ provided by each ν_j, we define *the characteristic function* $S_j(t)$, defined as follows:

$$S_j(t) = \begin{cases} 1 & t \in \mathcal{P}_j \\ 0 & t \notin \mathcal{P}_j \end{cases}. \tag{9}$$

Without loss of generality, assume the tasks $\{\tau_k\}$ within Γ are ordered by non-increasing priority. Consider a single task $\tau_k \in \Gamma$. L_ℓ denotes the sum of the duration of all the time intervals over $[0, D_k)$ where ℓ virtual processors provide service in parallel:

$$\forall \ell : 0 \le \ell \le m, L_\ell = \left| \left\{ t \in [0, D_k) : \sum_{j=1}^{m} S_j(t) = \ell \right\} \right|. \tag{10}$$

With W_k we denote the workload of jobs with higher priority interfering with τ_k, and I_k denotes the total duration in $[0, D_k)$ in which τ_k is preempted by higher priority jobs. From [16] we know that, for a fixed priority scheduler, the workload $\overline{W}_k^{\mathsf{FP}}$ can be bounded using:

$$\overline{W}_k^{\mathsf{FP}} = \sum_{i=1}^{k-1} \overline{W}_{k,i}, \tag{11}$$

where

$$\overline{W}_{k,i} = N_{k,i} C_i + \min\{C_i, D_k + D_i - C_i - N_{k,i} T_i\}, \tag{12}$$

with $N_{k,i} = \left\lfloor \frac{D_k + D_i - C_i}{T_i} \right\rfloor$.

The following theorems, proved in [2], allow us to build a schedulability test.

Theorem 1 *Given a multi-supply function characterized by the lengths $\{L_\ell\}_{\ell=0,\ldots,m}$ over a window $[0, D_k)$, the interference I_k on τ_k produced by a set of higher priority jobs with total workload W_k cannot be larger than*

$$\overline{I}_k = L_0 + \sum_{\ell=1}^{m} \min \left(L_\ell, \frac{\max\left(0, W_k - \sum_{p=1}^{\ell-1} p L_p\right)}{\ell} \right). \tag{13}$$

Now that we know how to calculate an upper bound to the interference $\overline{I}_k^{\mathsf{FP}}$, substituting $W_k = \overline{W}_k^{\mathsf{FP}}$ in the equation above, we can use the following theorem (again, from [2]) to derive a schedulability test.

Theorem 2 *A task set $\Gamma = \{\tau_i\}_{i=1,\ldots,n}$ is schedulable by a fixed priority algorithm on a set of virtual processors $\mathcal{V} = \{\nu_j\}_{j=1,\ldots,m}$ modeled by $\{Z_j\}_{j=1,\ldots,m}$, if*

$$\forall k \in \mathbb{N} : 1 \le k \le n \qquad C_k + \overline{I}_k^{\mathsf{FP}} \le D_k, \tag{14}$$

using the following values for the lengths $\{L_\ell\}_{\ell=0,\ldots,m}$:

$$\begin{aligned} L_0 &= D_k - Z_1(D_k) \\ L_\ell &= Z_\ell(D_k) - Z_{\ell+1}(D_k) \\ L_m &= Z_m(D_k). \end{aligned} \tag{15}$$

The symmetry of our bandwidth distribution allows for a simplification in the above test. In fact we assign the same bandwidth and the same period to all the virtual processors corresponding to the same task group; thus the intermediate lengths L_ℓ are zero, and Eq. (13) can be simplified, resulting in the following equation for the interference $\overline{I}_k$:

$$\overline{I}_k = L_0 + \min \left(L_m, \frac{\max(0, W_k - m L_m)}{m} \right). \tag{16}$$

As a final note, to multiplex different task groups on the same set of physical processors, the basic (necessary and sufficient) H-CBS admission test over the virtual processors $\{\nu_i\}_{i=1,\ldots,n}$ executing on the same physical processors must be verified:

$$\sum_{i=1}^{n} \frac{Q_i}{P_i} \le 1. \tag{17}$$

3.4 Shared Resources

In order to support access to shared resources, the priority inheritance and boosting mechanisms, already present in the Linux kernel, may be exploited. Within the Linux kernel, the fact that internal mutexes do not adopt priority inheritance mechanisms limits the possibility of giving formal upper bounds to blocking times.

In our implementation, we are exploring the usage of non-preemptive critical sections, realized raising the priority of the task executing in critical section to the maximum one available in the system. We are trying to adapt the approaches and the analysis in [17] and [18] to our model; a formal treatment of the topic is left as a future work.

3.5 Policy and Mechanisms

The proposed scheduling framework can be used as the basis for implementing several different variations, using mechanisms already present in the kernel, or introducing small modifications.

As an example, consider a user willing to adopt a purely partitioned approach: said user needs only to use the `cpuset` mechanism to specify a CPU affinity for the task groups, and no changes are required to the scheduler itself. The partitioned queues make handling this case quite efficient, while the H-CBS scheduler takes care of partitioning the bandwidth among the task groups on the same physical processor, according to the specified timing constraints.

Another open issue is the optimal bandwidth assignment between virtual processors. We stick to the current Linux model of using the same assignment on each physical processor, both for its simplicity and for lack of an interface to express different assignments. Anyway the H-CBS scheduler would support asymmetric partitions too, and exploiting this capability would came at the cost of adding the user interface to set Q_i/P_i on a per-virtual processor basis, again, with no modification to the scheduler structure.

4 Implementation

We implemented our framework in the Linux kernel. We modified the existing real-time scheduling class, changing how task groups are selected for service.

The existing code represents groups of tasks using `struct task_group` objects; tasks can be grouped on the basis of their user id or on the basis of the cgroup they belong to. Each task group contains an array of per-processor runqueues and scheduling entities. Each runqueue contains the scheduling entities belonging to all its (active) child nodes in the hierarchy. Tasks are leaf nodes, represented only by their own scheduling entity. Each processor has its own runqueue, containing the scheduling entities belonging to tasks and groups from the highest level in the hierarchy; a task group has a different scheduling entity on each processor it can run on.

Fig. 3 shows the main differences introduced to the kernel data structures: the priority array in `struct rt_rq` has been substituted with a red-black tree, and a new field (`rt_deadline`) had to be added. The per-group high-resolution timer previously used for implementing the throttling limitation was replaced by a per-runqueue timer. The `rt_rq`s of a same task group are scheduled independently on the processors with H-CBS, thus the limitation periods are asynchronous among each other. If the high resolution tick is enabled on the system, the scheduler will use it to deliver accurate end-of-instance preemptions.

The two arrays added to `struct task_group` are used to store all the tasks for the given task group on each processor. The problem here is that tasks are not scheduled using H-CBS, and there is no easy way to mix their entities with the ones associated to intermediate nodes in the hierarchy. Our solution was to add a leaf runqueue to each intermediate runqueue, to store its tasks.

```
static inline int
rt_entity_before(struct sched_rt_entity *a,
                 struct sched_rt_entity *b)
{
        struct rt_rq *rqa = group_rt_rq(a), *rqb =
            group_rt_rq(b);

        if ((!rqa && !rqb) || (rqa->rt_nr_boosted &&
            rqb->rt_nr_boosted))
                return rt_se_prio(a) < rt_se_prio(b);

        if (rqa->rt_nr_boosted)
                return 1;

        if (rqb->rt_nr_boosted)
                return 0;

        return (s64)(rqa->rt_deadline -
            rqb->rt_deadline) < 0;
}
```

Figure 2: Entity Ordering.

```
struct rt_edf_tree {
        struct rb_root rb_root;
        struct rb_node rb_leftmost;
};

struct rt_rq {
        struct rt_edf_tree active;
        u64 rt_deadline;
        struct hrtimer rt_period_timer;
        /* ... */
};

struct task_group {
        struct sched_rt_entity **rt_task_se;
        struct rt_rq **rt_task_rq;
        /* ... */
};
```

Figure 3: Data Structures.

Just to give a rough sketch of how the active tree is handled, Fig. 2 shows the function used to order entities. When inserting into a leaf runqueue both entities are tasks, so their priorities are compared. When both entities are runqueues they are ordered by priority if both of them are boosted (i.e., executing inside a critical section), otherwise boosted runqueues are favored over non-boosted ones. If none of them is boosted, they are ordered by deadline.

The cgroup interface exported by the scheduler has been extended, in order to allow the definition of the CPU reservation for the tasks belonging to a group. As we previously said, all the tasks in a group are scheduled using a “ghost” runqueue, which gets its own CPU share; the only change we made to the current cgroup user interface was adding the filesystem parameters to specify the bandwidth allocated to this ghost queue, i.e., the bandwidth allocated to the tasks in each given group.

5 Experiments

This section presents some preliminary results obtained with our implementation of the scheduler described so far. Our primary focus is evaluating the overhead introduced by the mechanism, thus we compare it to the current throttling implementation.

We measured the time spent by the scheduler inside each of the class-specific hooks, filtering out the callbacks registered by the other scheduling classes. For our measurements we instrumented the scheduler code and then we used an ad-hoc minimal tracer[4], that measured the time spent in the main scheduling functions using the timestamp counter present in all the modern x86 CPUs. The values acquired using the TSC were stored in a per-processor ring buffer and copied to userspace using a daemon reading from a character device; the fact that all the functions we profiled are called under the runqueue locks assured that measurement errors due to interrupts or preemptions were avoided.

The functions we measured are:

- `check_preempt_curr_rt()`, which, given the current task and a newly woken one, checks if the latter is entitled to preempt the former;
- `task_tick_rt()`, which handles the system tick for RT tasks (mainly it checks for timeslice expiration of round-robin tasks);
- `enqueue_task_rt()`, which adds a task to the RT runqueues. In our implementation this function is responsible of updating the deadline according to the H-CBS rules, if necessary;
- `dequeue_task_rt()`, which removes a task from the RT runqueues;
- `put_prev_task_rt()`, which moves the running task back to the ready (but not running) state;
- `pick_next_task_rt()`, which selects the next task to run (if any).

The system used was a quad-core Intel Q6600, clocked at 2.40GHz, equipped with 2GB of RAM. The synthetic load we chose was the Fixed Time Quanta [19] benchmark, executed at RT priority.

Fig. 4 shows the execution times for the RT scheduling-related functions mentioned above, in the case there are four (one per core) application threads running. With our approach, the enqueue and dequeue paths are slower, as one would expect with the substitution of the previous $O(1)$ priority array implementation.

[4]We didn't use the `ftrace` infrastructure because on our configuration it introduced non-negligible overheads.

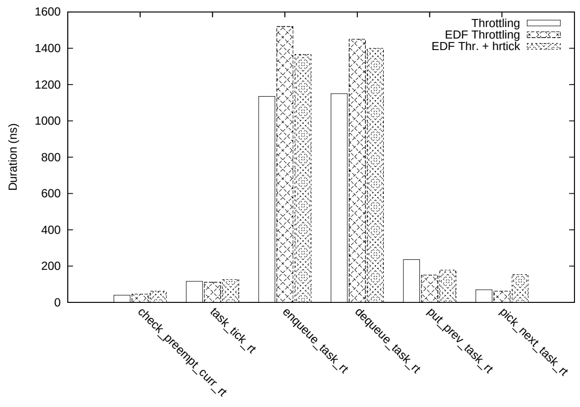


Figure 4: Scheduling Overhead—Flat Hierarchy.

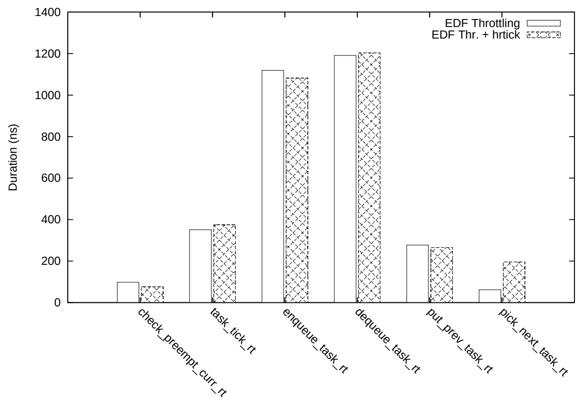


Figure 5: Scheduling Overhead—Full Hierarchy.

It is also worth noting that using the high resolution tick does not seem to affect the performance of the scheduling functions, except for `pick_next_task_rt()`, which is the function that programs the timer.

Fig. 5 shows the execution times of the RT scheduling class methods when there are more than a single root group, and with groups using different periods. We could not show the current scheduler behavior, as it does not support non-uniform periods. Performance is not too far from what shown in Fig 4 for native Linux, while in this case the overhead for posting the high resolution tick is more evident.

6 Availability

The implementation of the scheduler described in this paper is available as a patch to the Linux kernel, version 2.6.30-rc8, the latest available at the time of writing. It can be downloaded from

```
http://feanor.sssup.it/~fabio/linux/edf-throttling/
```

7 Future Work

The scheduler presented in this paper is still a work in progress. Our final objective is obtaining an implementation that can be considered for merging by the Linux community, yet based on sound theoretical principles.

About the implementation, we need a detailed study of the introduced overheads, along with the analysis of the computational cost given by keeping partitioned queues to implement a global scheduling strategy. From the theoretical standpoint, the biggest hole that needs to be filled in our opinion is the analysis of shared resources access.

8 Conclusion

In this paper we introduced a scheduling framework extending the Linux scheduler in order to improve its support for real-time workloads on multiprocessor systems. The main contribution of the paper is the synthesis between known theoretical results and the simplicity of the scheduler implementation, along with the specification of a complete strategy to solve the different issues that must be considered when designing a CPU scheduler (i.e., it deals with shared resources, CPU affinities, partitioned data structures and so on).

References

[1] T. Cucinotta, G. Anastasi, and L. Abeni, "Respecting temporal constraints in virtualised services," in *To appear in Proceedings of the 2^{nd} IEEE International Workshop on Real-Time Service-Oriented Architecture and Applications (RTSOAA 2009)*, Seattle, Washington, July 2009.

[2] E. Bini, G. Buttazzo, and M. Bertogna, "The multi supply function abstraction for multiprocessors," *to appear,available online at http://feanor.sssup.it/∼marko/RTCSA09.pdf*, 2009.

[3] RTLinux homepage, http://www.rtlinux.org.

[4] RTAI homepage, http://www.rtai.org.

[5] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari, "AQuoSA — adaptive quality of service architecture," *Software – Practice and Experience*, vol. 39, no. 1, pp. 1–31, 2009.

[6] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proc. IEEE Real-Time Systems Symposium*, Madrid, Spain, 1998.

[7] D. Faggioli, G. Lipari, and T. Cucinotta, "An efficient implementation of the bandwidth inheritance protocol for handling hard and soft real-time applications in the Linux kernel," in *Proceedings of the 4^{th} International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2008)*, Prague, Czech Republic, July 2008.

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

[9] B. Brandenburg, J. M. Calandrino, and J. H. Anderson, "On the scalability of real-time scheduling algorithms on multicore platforms: A case study," in *Proceedings of the Real-Time Systems Symposium*, Barcelona, 2008.

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

[11] IEEE, *Information Technology -Portable Operating System Interface (POSIX)- Part 1: System Application Program Interface (API) Amendment: Additional Realtime Extensions.*, 2004.

[12] D. Faggioli, A. Mancina, F. Checconi, and G. Lipari, "Design and implementation of a POSIX compliant sporadic server," in *Proceedings of the 10^{th} Real-Time Linux Workshop (RTLW)*, Mexico, October 2008.

[13] A. K. Mok, X. A. Feng, and D. Chen, "Resource partition for real-time systems," *Real-Time and Embedded Technology and Applications Symposium, IEEE*, vol. 0, p. 0075, 2001.

[14] A. Easwaran, M. Anand, and I. Lee, "Compositional analysis framework using edp resource models," *Real-Time Systems Symposium, IEEE International*, vol. 0, pp. 129–138, 2007.

[15] D. Stiliadis and A. Varma, "Latency-rate servers: A general model for analysis of traffic scheduling algorithms," in *IEEE/ACM Transactions on Networking*, 1996, pp. 111–119.

[16] M. Bertogna, M. Cirinei, and G. Lipari, "Schedulability analysis of global scheduling algorithms on multiprocessor platforms," *IEEE Transactions on Parallel and Distributed Systems*, 2008.

[17] M. Bertogna, F. Checconi, and D. Faggioli, "Non-preemptive access to shared resources in hierarchical real-time systems," in *1st Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, Barcelona, Spain, December 2008.

[18] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson, "A flexible real-time locking protocol for multiprocessors," in *RTCSA '07: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 47–56.

[19] FTQ http://rt.wiki.kernel.org/index.php/FTQ.

Threaded IRQs on Linux PREEMPT-RT

Luís Henriques
Intel Shannon, Ireland
Email: luis.henriques@intel.com

Abstract

In recent years, usage of GNU/Linux-based systems in embedded applications has increased and several Linux-enabled devices are currently available. From network devices to mobile phones, the Linux kernel is being adopted by developers and companies as an alternative to other proprietary operating systems traditionally used in embedded systems. However, markets grow very fast and the industry is pushing real-time requirements into the Linux kernel.

This article presents an overview of a feature called "Threaded IRQs" that has been used by the PREEMPT-RT *patch for years. This feature, currently being pushed to the mainline kernel by the patch maintainers, allows interrupt handlers to be executed as regular kernel threads. Along with the description of the implementation details of this feature, some experimental results to measure the effectiveness of threaded* IRQs *in the Linux kernel are also presented.*

Index Terms

Linux, kernel, real-time, PREEMPT-RT*,* IRQ*, cyclictest.*

1. Introduction

In contrast to traditional real-time operating systems that have been designed from the beginning to provide predictable response time, the Linux kernel[1], as a general-purpose operating system, was designed around fairness and good average performance under sustained load conditions. There are, however, several attempts to provide Linux with real-time capabilities[2] and this article will focus on one of these attempts: the PREEMPT-RT patch[3], [4].

This article presents an overview of one of the features implemented in the PREEMPT-RT patch that is not yet available in the mainline Linux kernel: the ability to process interrupts in a threaded context. Some experimental measurements have also been defined in order to evaluate the impact introduced by threaded IRQs (*Interrupt Requests*) in system throughput and system latencies.

The remainder of this paper is structured as follows. Section 2, *Related work*, describes some implementations of the threaded IRQs concept in systems other than Linux. Section 3, PREEMPT-RT *overview*, presents an overview of the main characteristics of the PREEMPT-RT patch. Section 4, *Threaded IRQs in* PREEMPT-RT, provides details on implementation of threaded IRQs in the PREEMPT-RT patch. Section 5, *Experiment description*, describes the test scenarios and benchmark selected to measure the impact of threaded IRQs in the Linux kernel. Section 6, *Experiment results*, provides data collected by tests. Finally, Section 7, *Conclusions*, summarizes the results obtained and draws some conclusions.

2. Related work

The usage of threaded contexts in interrupt handling is a pattern that can be found in several multi-threaded UNIX® kernels. Threaded interrupt handlers have been used with two major purposes: 1) to simplify the programming model by having an uniform synchronization model both for processes/threads and interrupts and 2) to improve system predictability by decreasing the number of times in which interrupts are masked.

One example of threaded interrupt handling can be found in the FreeBSD[5][6] kernel. By implementing interrupt threads, the FreeBSD kernel allows the interrupt handlers to perform operations typically not allowed in an interrupt handler's context, for example, to block on locks. Since full context-switching is performed whenever an interrupt thread is to be scheduled, FreeBSD implements these threads with real-time kernel priorities in order to decrease latencies. This way, interrupt threads are guaranteed to execute with higher priority than other user-level threads. Note, however, that not all the interrupt handlers are executed in a threaded context on the FreeBSD kernel. Currently, there are two interrupt handlers that do not have their own context: the clock interrupt and the serial I/O device interrupts. These two handlers are called "fast interrupt handlers" and they are not allowed to acquire blocking locks — they can only use spin mutexes.

A different approach is implemented in the Solaris 2 kernel [7]: interrupts behave like asynchronously-created threads. However, since thread creation operations are very expensive, it is not feasible to create new threads when interrupts occur. Thus the kernel keeps per-CPU pools of threads in a pre-initialized state. This way, when an interrupt occurs, the interrupted thread is set into a non-runnable state (*pinned*, in Solaris terminology) and the interrupt thread is executed. Since no context-switch has actually occurred, a *pinned* thread can not be scheduled while the interrupt thread is running, not even in a different CPU. This mechanism avoids the overhead of completely saving the thread's state or putting threads on run queues. Full context-switch actually occurs when the interrupt handler thread needs to block (e.g., on a mutex). When this occurs, the handler is changed into a regular thread, capable of being scheduled. The *pinned* thread is set as runnable and the scheduler is invoked. This mechanism allows the Solaris kernel to avoid the overhead of creating a full thread if the interrupt handler does not need to block — this overhead is postponed to the moment the interrupt thread actually blocks.

An example of a non-UNIX® operating system that postpones the completion of interrupt servicing until after the ISR returns is the Microsoft® Windows® operating system[8][9]. In this system, the interrupt servicing consists typically of two components: an *Interrupt Service Routine* and a *Deferred Procedure Call* (DPC). The ISR part is executed at a high priority, while the DPC part is executed at a lower priority (Windows® uses its own interrupt priority scheme on top of the interrupt priorities imposed by the interrupt controllers, known as *Interrupt Request Level* or IRQL). For example, an ISR registered from a device driver typically has minimal interaction with the hardware device, limiting its operation to reading state registers and masking its interrupts. Then, the ISR can request the kernel to schedule a DPC, which will be executed later, at a lower IRQL. By scheduling a DPC at a lower IRQL, the device driver allows interrupts that are set with a higher IRQL to occur. Although the operations that can be executed from a DPC are restricted (e.g., a DPC cannot block), a DPC is far less restrictive than an ISR.

The usage of threads to handle interrupts has also been used in micro-kernel architectures. Micro-kernels are minimalistic operating system kernels that implement only a small set of functionalities that require special privileges, such as address spaces, threads support and message-based Inter-Process Communication (IPC). All other functionalities, including device drivers, are left to be implemented in user-space by *servers*. The L4 micro-kernel[10][11], for example, abstracts the hardware itself as being a set of threads. These threads have special IDs and are responsible for sending a synchronous IPC message to a user-space thread which previously registered an interrupt handler for a particular interrupt. Interrupts are handled by the micro-kernel by simply masking the interrupt in the interrupt controller and signaling the user-space thread with an IPC message. The real handler is eventually executed in user-space with its interrupt masked, but with all other interrupts enabled. Any other interrupt can occur in the meantime, preempting the interrupt handler. Finally, when the handler finishes servicing the interrupt, it will signal the micro-kernel which will then unmask the interrupt controller.

3. PREEMPT-RT overview

The roadmap defined for PREEMPT-RT by its initial creator Ingo Molnar was to gradually add real-time (RT) capabilities to the Linux kernel[1]. This was definitely a long-term job and several functionalities that have been developed for the PREEMPT-RT patch have already made their way to the mainline kernel. However, several core characteristics of this patch, such as the threaded IRQs, continue to be developed outside of the main kernel tree.

The goals of the PREEMPT-RT kernel patch include:

- Fixed priority preemptive scheduling on the kernel. In POSIX, this is translated into using the scheduling policies `SCHED_FIFO` and `SCHED_RR` as opposed to `SCHED_OTHER`, which is the default policy on Linux.
- No impact on non-RT tasks running on the system, meaning that the system could have both RT and non-RT tasks running together.
- User control of the trade-off between latency and throughput by configuring the kernel in a proper way.

The major changes introduced in the Linux kernel by the PREEMPT-RT patch include:

- **Complete kernel preemption —** The *vanilla* kernel currently has three preemption configuration options: *No forced preemption*, *Voluntary kernel preemption* and *Preemptible kernel*. The PREEMPT-RT patch introduces a new configuration option, *complete preemption*, which further reduces scheduling latency by replacing most of the spinlocks with blocking mutexes.
- **High-resolution timers —** The introduction of high-resolution timers [12][13] includes the conversion of the old Linux timer API into two separate infrastructures: one for high-resolution kernel timers (e.g., `sleep`) and another to handle timeouts. These modifications lead to user-space POSIX *hrtimers*. High-resolution timers is a feature that was initially implemented in the PREEMPT-RT patch and has now been pushed to the mainline kernel.
- **Priority inheritance —** The solution to handle the priority inversion problem in the PREEMPT-RT kernel was to implement the Priority Inheritance protocol. Basically, this protocol defines that, if a high priority task blocks on a resource that is currently held by a lower priority task, this task temporarily inherits the priority of the blocking task. This priority inheritance change is temporary — as soon as the task releases the resource, its priority is restored. The priority inheritance

1. Whenever "Linux kernel" is mentioned, this is Linux kernel that can be obtained from [1], also called the "Vanilla kernel" or "mainline kernel".

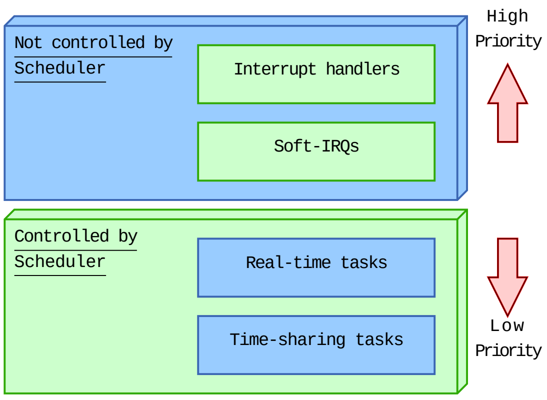


Figure 1. Execution entities

implementation is currently available in the *vanilla* kernel.

For further details on the PREEMPT-RT patch, please refer to [14], [4].

4. Threaded IRQs in PREEMPT-RT

Let us consider a hypothetical scenario: a low priority task doing heavy I/O operations (for example, writing huge amounts of data onto hard-disk). If there is a higher priority task that keeps preempting the low priority task, it will be disturbed by interrupts triggered by hardware due to the lower priority task activity. This is a realistic scenario that can be easily observed in a Linux system since interrupt handling routines have the highest priority amongst all execution entities in the system.

The PREEMPT-RT patch provides a solution for this problem. However, before describing the solution, it is necessary to describe how Linux handles the different execution entities that may be active in the system at each moment.

4.1. Linux execution entities

Fig. 1 shows several execution entities available on the mainline kernel[2]: higher priority entities on the top (Interrupt Service Routines, or ISRs) and lower priority entities on the bottom (regular user-space tasks). The only entities that are under the scheduler control are the regular Linux processes and the real-time processes; ISRs and the soft-IRQs are not under the scheduler control on the *vanilla* kernel.

This means that, whenever an interrupt occurs, no matter what task is being executed, the CPU is preempted and the ISR is executed. ISRs can be preempted only by other higher-priority interrupts that may occur. This way, a device driver is expected to do very little work in the IRQ handler (typically just confirm that IRQ is to be handled by the driver and acknowledge the hardware) and to postpone any time-consuming task to a soft-IRQ.

2. The figure does not represent all the execution entities that are available in Linux systems. For example, it does not make any distinction between soft-IRQs and tasklets and it does not show any kernel threads.

Soft-IRQs (and tasklets) are routines that are executed after ISRs have been processed and before the interrupted process is resumed. These routines are responsible for doing any task that has been postponed by ISRs (for example, copying data to/from buffers). Although soft-IRQs and tasklets are very similar, they have some differences:

- One tasklet cannot execute simultaneously on two different CPUs, while soft-IRQs can.
- It is guaranteed that a tasklet will be executed on the same processor that scheduled it. This is not guaranteed for soft-IRQs.

This means that tasklets are much easier to implement since they do not need to be reentrant.

4.2. PREEMPT-RT solution

The solution provided by the PREEMPT-RT patch to the problem described at the beginning of Section 4 is to have all the interrupt handlers running as regular kernel threads — this way, a high priority thread can avoid being preempted by the interrupt handler thread by setting its priority to be higher than the interrupt handler thread priority.

A device driver registers a new IRQ handler with a PREEMPT-RT patched kernel using the same interface as in the *vanilla* kernel, i.e., through routine `request_irq`. A new thread will be created by the kernel to handle the IRQ only if that IRQ does not have already one thread associated — there will only exist one thread for each IRQ. The device driver does not need to be designed to take advantage of threaded IRQs, i.e., device drivers from the *vanilla* kernel can be used with the PREEMPT-RT kernel running with threaded IRQs without modifications.

However, a device driver may not want to have its interrupt handler being executed in a threaded context. In this case, when invoking the `request_irq` routine, the device driver has to set the interrupt type flag with IRQ_NODELAY. This way, the kernel will not create an IRQ thread.

Shared IRQs are handled by the kernel by keeping a list, ordered in a FIFO maner, of all the ISRs registered per IRQ. Thus, when an interrupt occurs, all the registered ISRs for the corresponding IRQ are invoked from the IRQ thread. Note that the shared IRQ ISRs must both be either non-threaded or threaded, i.e., it is not possible to have a threaded ISR sharing the IRQ of a non-threaded ISR.

Fig. 2 is a simplified function call graph that shows what happens when an interrupt line is asserted. The entry point for all IRQs is routine `do_IRQ`. A more appropriate handler will then be invoked, depending on the type of interrupt being handled (for example, edge-triggered interrupt are

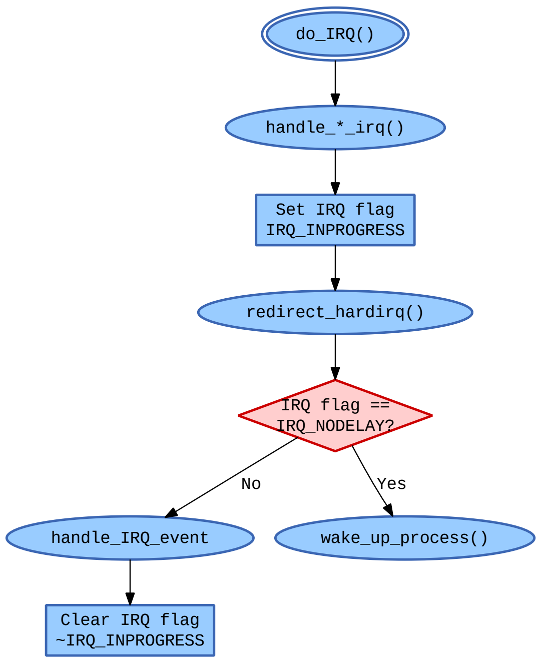


Figure 2. Threaded IRQs call graph

handled differently from level-triggered interrupts). This interrupt type specific handler is set by the kernel during its initialization and will be responsible for the *Programmable Interrupt Controller* (PIC) low-level operations such as masking the interrupt and signaling the *End-of-Interrupt* (EOI) to the PIC.

The status flag in the IRQ descriptor is set to IRQ_INPROGRESS and the `redirect_hardirq` routine is invoked. This routine is responsible for checking whether this IRQ is handled by a thread, i.e., the IRQ descriptor flag does not contains IRQ_NODELAY. If it is handled by a thread, the thread is woken up, otherwise the IRQ is handled immediately by sequentially invoking all the registered ISRS for the corresponding IRQ.

Since edge-triggered interrupts can occur in the falling and/or rising edge of an hardware signal, these interrupts need to be acknowledged in order to be re-enabled. If this is not done, there is a risk of losing an interrupt that occur while another one is being handled. Thus, for the edge-triggered interrupts, the interrupt is re-enabled during the interrupt handler so that the handler is able to recognize the situation where the same IRQ occurred while its handler was already executing.

A simplified version of the IRQ kernel thread body is shown in Listing 1.

The `set_current_state` function will change the state of the currently executing task to TASK_INTERRUPTIBLE. When invoking the `schedule` function with the task in this state, the task is removed from the run queue before any other task is scheduled. This

```
while (!kthread_should_stop()) {
  set_current_state(TASK_INTERRUPTIBLE);
  do_hardirq(desc);
  schedule();
}
```

Listing 1. IRQ kernel thread body

way, the task is set to sleep and is not scheduled again until another task wakes it again through `wake_up_process`. Invoking `wake_up_process` on a task that is in TASK_INTERRUPTIBLE state will result in setting its state again to TASK_RUNNING and inserting it back into the run queue.

The actual interrupt handling is done in a loop that will invoke all the handlers that have been previously registered through routine `request_irq` for the current IRQ . This loop is implemented in `handle_IRQ_event`.

4.3. Benefits of threaded IRQs

There are several advantages of having threaded IRQs in the kernel. One of these advantages has already been mentioned: it allows the user to set tasks with priorities higher than the IRQ handlers. This way, high priority (RT) tasks may be configured in a way so that they are not disturbed by lower priority (non-RT related) interrupts.

Other advantages of having threaded IRQs are:

- Since IRQ handlers are now regular kernel threads, it is possible to modify IRQ handler priorities and have IRQs with priorities different from those enforced by the hardware.
- System observability is increased since it is much easier to debug/instrument a thread than a traditional interrupt handler.
- Interaction between the hard-IRQ handler and the soft-IRQs/tasklets is simplified, without the usual locking complexity.

There are, however, some interrupt handlers that are not desirable to be executed as threads due to the fact that they execute critical operations to the overall system. The most obvious example of such interrupts is the timer interrupt. Thus, the ISR handler for this IRQ is set up using the IRQ_NODELAY flag so that it will force the interrupt handler to be executed in the interrupt context and not in threaded context.

The usage of threaded IRQs has also one major drawback: performance will decrease when interrupts are handled by threads. The main reason for this is that the number of context switches between threads is higher due to the IRQ scheduling. Thus, the overall throughput of a Linux system using threaded IRQs can be lower when compared with another system that does not use this feature.

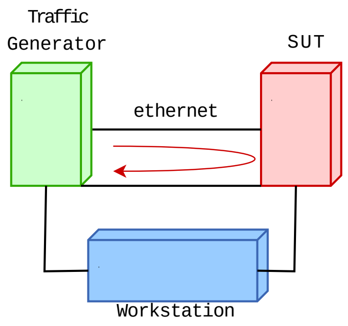


Figure 3. Test environment setup

5. Experiment description

In order to evaluate the effects of having threaded IRQs on the Linux kernel, a test environment was set up using a board with an Intel® EP80579 integrated processor. This processor is an Intel® Architecture (IA) based system-on-a-chip (SoC) processor, based on the Intel® Pentium® M processor. The SoC includes integrated memory controller hub, integrated I/O controller hub, and flexible integrated I/O support with three Ethernet MACs, two Controller Area Network (CAN) interfaces and a local expansion bus interface. The processor used in these experiments was an 800 MHz clock version, with 256 Kb cache and 769884 Kb of RAM.

The System Under Test (SUT) was configured to perform IP traffic forwarding between two different networks, associated with two ethernet cards (see Fig. 3). These two ethernet cards were connected to a traffic generator that injected traffic in one of the networks and received it in the other network.

The ethernet cards used in the experiments were two Intel® 82572EI Gigabit Ethernet Controller for PCI Express, using the `e1000e` Linux device driver.

This test environment was controlled from a workstation which was connected both to the SUT and to the traffic generator.

Several test experiments were defined in order to evaluate the effects of having threaded interrupts in the Linux kernel. These experiments were executed on three different configurations of Linux kernel version 2.6.29.3:

- *Vanilla* — This configuration is, basically, the mainline kernel without any PREEMPT-RT option. It was used as a reference for results comparison.
- *Threaded* IRQ*s* — This configuration uses a kernel patched with the PREEMPT-RT but only the configuration options needed to activate the threaded IRQs have been selected.
- *Fully* PREEMPT-RT — This configuration activates all the main features from the PREEMPT-RT patch, including threaded IRQs and the *Complete preemption* option.

Finally, a benchmark application was selected to be executed on the Linux box. This benchmark, *cyclictest*[15], was first written by Thomas Gleixner as a tool to investigate possible regressions while developing the PREEMPT-RT patches. It is now considered the state-of-the-art measurement tool to determine the internal worst-case latency of a Linux real-time system.

Basically, *cyclictest* is composed of one thread (or a set of threads) that sleeps for a certain period of time and measures the accuracy of this sleep system call. 500 μsecs was the interval selected for these experiments. Thus, *cyclictest* is able to measure a full chain of different types of latencies, namely:

- Timer interrupt
- Scheduler
- User space execution

The *cyclictest* application was executed in the following test scenarios:

- *No load* — The system was idle and the benchmark application was the only application being scheduled. This scenario was for reference only.
- *Low priority* — The system was under load with huge amounts of ethernet traffic being forwarded between the two networks. The *cyclictest* thread was executed with a priority lower than the IRQs associated with the ethernet cards.
- *High priority* — The system was under load with huge amounts of ethernet traffic being forwarded between the two networks. The *cyclictest* thread was executed with a priority higher than the IRQs associated with the ethernet cards.

Each of these scenarios was executed in the three kernel configurations, with the exception that, for the *vanilla* kernel configuration, only two scenarios were executed since it is not possible to configure an application to have higher priority than the IRQs. This means that a total of eight different experiments were executed.

To load the system, a traffic generator equipment was configured to inject traffic in one of the ethernet cards of the SUT (*eth0*). The SUT then forwarded the traffic to the other ethernet card (*eth1*). The ethernet packets injected were all the same size, 64 bytes. Also, different loads were defined — packets were injected with different frequencies, namely with 10, 9, 8, 7, 6 and 5 microseconds. Values smaller than 5 microseconds were discarded since at this point the SUT starts dropping too many packets.

All the tests were executed during a fixed period of time: 10 minutes each test.

6. Experiment results

This section provides the data collected during execution of the experiments described in the previous section. Each subsection contains the data collected for each of the defined kernel configurations: *vanilla* kernel, PREEMPT-RT kernel with threaded IRQs configuration option and PREEMPT-RT kernel with threaded IRQs and *Complete preemption* options.

6.1. Vanilla kernel results

This section provides the results obtained for the mainline kernel, i.e., the kernel without the PREEMPT-RT patch.

Table 1. Vanilla kernel cyclictest results

Tx Rate	cyclictest			# IRQs	
(μsecs)	Min	Avg	Max	eth0	eth1
10	5	48	585	18,076,863	13,882,919
9	4	48	1,180	13,912,623	11,093,282
8	4	52	1,051	10,644,941	9,791,348
7	4	76	1,435	10,350,929	9,970,528
6	5	94	1,893	7,808,480	8,708,121
5	3	193	6,553	4,556,491	5,172,545
0	3	5	89	-	-

Table 1 provides the data obtained from the execution of the *cyclictest* benchmark.

The first column in the table lists the rates (in microseconds) at which the 64 bytes packets were injected in the system. The next three columns show the minimum, average and maximum latencies, i.e., the minimum, average and maximum times the *cyclictest* benchmark took to resume execution after the sleep of 500 μsecs. Finally, the two last columns show the number of interrupts that occurred in the two PCIe network cards during the period of time the test was being executed, i.e. 10 minutes.

Note that the last row in this table contains the results from an execution of the *cyclictest* benchmark with a transmit rate of 0 (zero), i.e., the system was idle and no packets were being injected. The results in this row are for reference only.

As we can see from this data, the *vanilla* kernel has a high variation in the latencies — comparing, for example, the difference between the maximum latency value when the system is not receiving any packets (last row in table) and the maximum latency value when the system is receiving one packet every 10 μsecs, there is a difference of 496 μsecs (585 minus 89).

Another observation on the data in this table is that the number of interrupts on the ethernet cards decreases when the packet transmission frequency increases. This is probably due to the fact that the ethernet device drivers use a mechanism for disabling interrupts and start polling the device when interrupt frequency is too high. The impact of this mechanism was not in the scope of this analysis and has not been further investigated.

Fig. 4 shows a graphical representation of the *cyclictest* benchmark execution results.

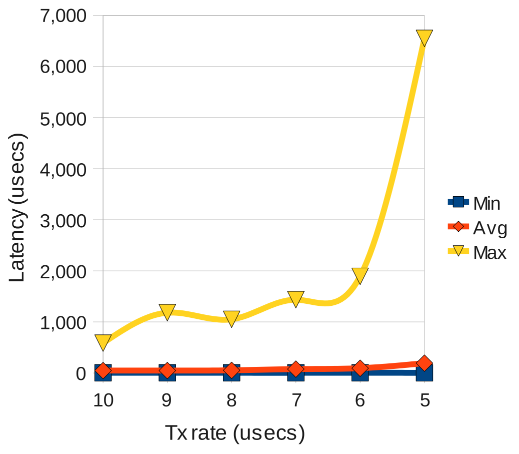


Figure 4. Vanilla kernel cyclictest graphic

Table 2. Vanilla 2.6.29.3 Tx/Rx

Tx Rate	Tx/Rx Frames			
(μsecs)	Tx	Rx	Lost	Lost (%)
10	57,716,382	57,716,382	0	0.00
9	63,260,305	63,260,305	0	0.00
8	70,782,330	70,782,330	0	0.00
7	80,704,937	80,704,937	0	0.00
6	91,873,761	91,873,761	0	0.00
5	124,042,888	124,042,888	0	0.00

Table 2 lists statistical data on the amount of traffic that was transmitted/received by the traffic generator. The first column shows the packet transmission rates in microseconds. The second and third columns represents the number of packets transmitted and packets received, respectively. The fourth and fifth columns contain the packet loss.

A note on the values on this table (and similar tables for other kernel configuration results) is that these values are not synchronized with the data in table 1. This is because there was no synchronization between the traffic generator and the SUT — the tests were manually executed and thus the values in this table are only approximated values.

As the table shows, the *vanilla* kernel is able to handle the amount of traffic generated without losing any packets. This is true even when injecting packets at a 5 μsec rate, where 124,042,888 ethernet frames of 64 bytes each have been sent, corresponding to more than 8 Gb of data.

6.2. PREEMPT-RT kernel with Threaded IRQs option results

This section provides the results obtained for the kernel patched with PREEMPT-RT, configured with threaded IRQs related options.

As described in Section 5, *Experiment description*, for this kernel configuration, the *cyclictest* benchmark was executed in two different scenarios: in the first scenario the benchmark is executed with a priority lower than the interrupt handlers for the ethernet cards, and in the second scenario the benchmark is executed with a priority higher than the interrupt handlers.

Since the interrupt handler threads are executed with priority 50 by default, in the first scenario the *cyclictest* was configured to run with priority 40 (lower than interrupt handler threads). Table 3 shows the results for this scenario.

Table 3. Threaded IRQs kernel cyclictest results (Low Priority)

Tx Rate	**cyclictest**			**# IRQs**	
(μsecs)	Min	Avg	Max	eth0	eth1
10	6	128,474,238	268,133,338	6,170,139	5,991,550
9	7	129,139,176	269,522,352	6,055,578	6,000,188
8	7	136,136,357	274,904,968	5,960,877	5,946,526
7	9	139,804,614	280,713,676	5,405,771	5,589,278
6	256	154,922,510	310,819,204	4,239,118	4,350,945
5	-	-	607,364,076	29,896	26,134

The first interesting conclusion that can be obtained from this data is that the number of interrupts in the ethernet cards is much lower than for the *vanilla* kernel. The overhead introduced by the context-switches to IRQ threads forces the ethernet cards to have their interrupts masked for a longer time. For this reason, the number of interrupts is actually lower in this configuration.

Also, the data in the table shows that this scenario has very high maximum latencies. For example, when transmission rate is 10 μsecs, the maximum latency obtained was greater than 4 minutes (268,133,338 μsecs) and when transmission rate is 5 μsecs, the benchmark is not even able to execute — maximum value in this row is around 10 minutes, corresponding to the total execution time for the test. This means that the CPU load is very high and no cycles are left for low priority applications.

Fig. 5 shows a graphical representation of the *cyclictest* execution results using threaded IRQs at a low priority.

Table 4. Threaded IRQs kernel Tx/Rx (Low Priority)

Tx Rate	**Tx/Rx Frames**			
(μsecs)	Tx	Rx	Lost	Lost (%)
10	57,001,190	57,001,190	0	0.00
9	63,118,069	63,118,069	0	0.00
8	70,603,675	70,603,675	0	0.00
7	79,494,929	79,494,929	0	0.00
6	91,920,546	91,417,773	502,773	0.55
5	112,119,077	103,828,807	8,290,270	7.39

Table 4 summarizes the amount of traffic transmitted/received by the traffic generator. This configuration is also not able to handle all the traffic that is injected on the system. Packets start to be dropped when high transmission rates are used (6 and 5 μsecs).

The scenario changes when the *cyclictest* benchmark is executed with higher priority. Table 5 shows the results of the benchmark execution when its priority is set to 80, which is a priority higher than the ISRs handlers (50, by default).

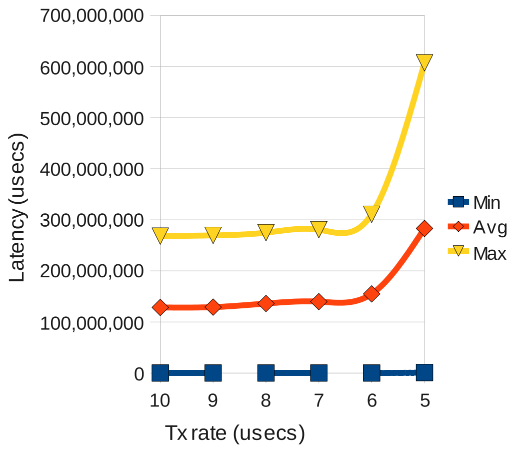


Figure 5. Threaded IRQs kernel cyclictest graphic (Low Priority)

Table 5. Threaded IRQs kernel cyclictest results (High Priority)

Tx Rate	**cyclictest**			**# IRQs**	
(μsecs)	Min	Avg	Max	eth0	eth1
10	4	8	65	6,223,553	5,994,708
9	4	8	479	5,780,033	5,836,146
8	5	9	74	5,123,027	5,277,829
7	6	135,580,508	272,452,238	4,467,440	4,591,824
6	6	140,860,392	282,329,933	2,830,294	2,808,786
5	8	156,278,465	313,641,236	16,384	14,435
0	4	5	93	-	-

Although the latencies for higher transmission rates are still very high, with lower transmission rates these values are significantly lower than for the *vanilla* kernel.

Fig. 6 shows a graphical representation of the *cyclictest* execution results using threaded IRQs at a high priority.

Table 6. Threaded IRQs kernel Tx/Rx (High Priority)

Tx Rate	**Tx/Rx Frames**			
(μsecs)	Tx	Rx	Lost	Lost (%)
10	56,920,922	56,920,922	0	0.00
9	62,747,678	62,664,039	83,639	0.13
8	70,389,695	70,389,695	0	0.00
7	79,638,318	79,637,232	1,086	0.00
6	91,039,343	89,535,226	1,504,117	1.65
5	108,608,754	98,436,714	10,172,040	9.37

Table 6 provides the statistical data collected in the traffic generator.

6.3. Fully PREEMPT-RT kernel results

This section provides the results for the Linux kernel patched with PREEMPT-RT, using both the threaded IRQs

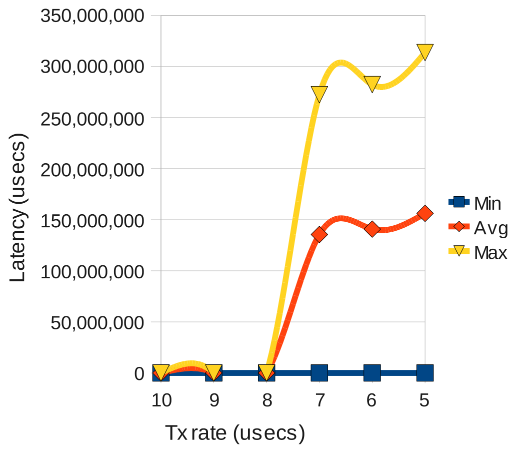


Figure 6. Threaded IRQs kernel cyclictest graphic (High Priority)

configuration options and the *Complete preemption* option.

Table 7 presents the results for the *cyclictest* benchmark executing with a priority lower than the priority of the IRQs threads.

The maximum latencies results for the lower transmit rates (7 to 10 μsecs) are similar to the results obtained for the *vanilla* kernel — around 1,400 μsecs. There is, however, still a very high variation between the minimum and the maximum values.

Similar to the previous configuration (PREEMPT-RT kernel with threaded IRQs only), the number of interrupts for the ethernet cards is lower than the values obtained for the *vanilla* kernel. Again, this is due to the overhead introduced by the context switching to ISRs threads.

Table 7. PREEMPT-RT kernel cyclictest results (Low Priority)

Tx Rate	**cyclictest**			**# IRQs**	
(μsecs)	Min	Avg	Max	eth0	eth1
10	2	60	1,417	6,094,390	5,887,783
9	2	74	1,294	6,053,376	5,858,603
8	7	129	1,466	5,553,851	5,596,537
7	2	152	1,546	4,810,329	5,044,578
6	1	1,610	75,579	3,644,729	3,765,491
5	370,627	252,284,884	496,880,098	1,724,908	1,629,418

Fig. 7 shows a graphical representation of the *cyclictest* execution results using threaded IRQs at a low priority.

In Table 8, the data collected from the traffic generator equipment shows that packet drops also occur in this configuration when transmission rates increase — with packets being transmitted every 6 μsec, around 1.51% of the packets are lost.

The last configuration used in the experiments was to set the *cyclictest* benchmark with priority 80, which is higher

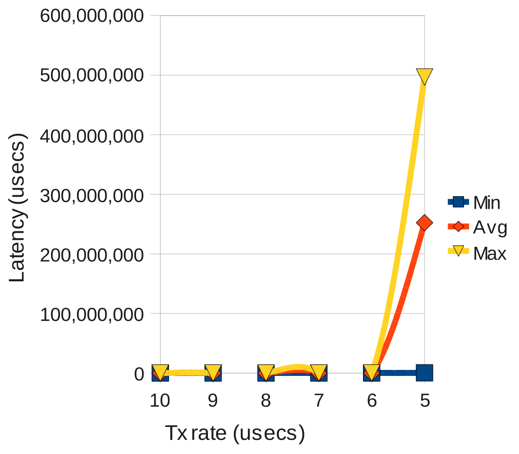


Figure 7. PREEMPT-RT kernel cyclictest graphic (Low Priority)

Table 8. PREEMPTRT 2.6.29.3 Tx/Rx (Low Priority)

Tx Rate	**Tx/Rx Frames**			
(μsecs)	Tx	Rx	Lost	Lost (%)
10	57,077,012	57,077,012	0	0.00
9	62,892,065	62,892,065	0	0.00
8	70,316,148	70,316,148	0	0.00
7	79,491,682	79,491,682	0	0.00
6	91,747,522	90,361,321	1,386,201	1.51
5	108,360,109	102,793,198	5,566,911	5.14

than the interrupt handlers. Table 9 presents the results of this configuration.

Table 9. PREEMPT-RT kernel cyclictest results (High Priority)

Tx Rate	**cyclictest**			**# IRQs**	
(μsecs)	Min	Avg	Max	eth0	eth1
10	5	9	27	6,158,486	5,835,824
9	5	9	29	5,792,293	5,617,035
8	5	9	32	5,558,052	5,573,145
7	5	9	30	4,774,998	4,898,368
6	1	42	25,880	3,398,359	3,431,860
5	5	1,249	50,153	454,401	407,200
0	4	5	25	-	-

In this configuration, the maximum values for the latencies reported by the *cyclictest* were around the 30 μsecs for transmission rates below 6 μsecs. The maximum latency value varies from 25 (when system is idle, i.e., no traffic is being injected) to 32 μsecs (when transmission rate was set to a packet every 8 μsecs). With a variation of 7 μsecs for these two scenarios, it is possible to state that the system is quite predictable. For higher transmission rates, values are again very high.

There is a huge latency of 25 milliseconds when moving from a packet rate of 7 to 6 μsecs. The overhead introduced by threaded IRQs does not explain such a high latency.

Further analysis would be required to identify the kernel code that is introducing this high latency.

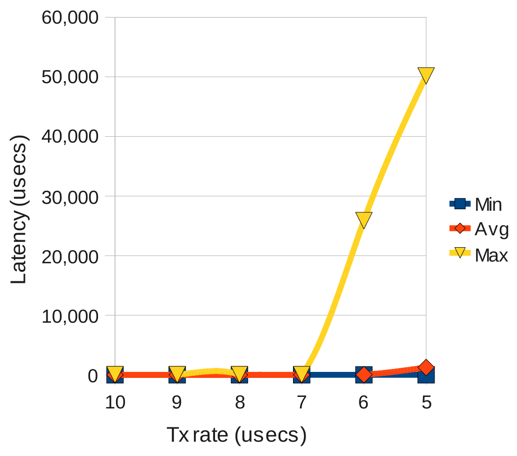


Figure 8. PREEMPT-RT kernel cyclictest graphic (High Priority)

Table 10 summarizes the data relative to the number of packets transmitted, received and lost during the experiment. Once again, packets start to be dropped at high transmission rates.

Table 10. PREEMPT-RT 2.6.29.3 Tx/Rx (High Priority)

Tx Rate	**Tx/Rx Frames**			
(μsecs)	Tx	Rx	Lost	Lost (%)
10	62,987,175	62,987,175	0	0.00
9	65,477,551	65,477,551	0	0.00
8	70,477,613	70,477,613	0	0.00
7	84,206,373	84,206,373	0	0.00
6	99,349,545	99,022,298	327,247	0.33
5	110,967,141	103,238,860	7,728,281	6.96

7. Conclusions

This paper provided an overview of one change introduced by the PREEMPT-RT patch-set to the Linux kernel: the threaded IRQs. A description of the implementation details of this modification has been provided, along with some experiments that have been conducted to measure the effectiveness of the threaded IRQs in the Linux kernel.

The implementation of threaded IRQs in PREEMPT-RT follows the same approach as the FreeBSD implementation. As described in section 2, *Related work*, FreeBSD implements threaded IRQs using full context-switching whenever an interrupt thread is to be scheduled. This was also the selected design for the PREEMPT-RT.

No special optimizations have been introduced by the PREEMPT-RT code in order to improve performance. Optimizations such as those used in the Solaris kernel (context-switch to interrupt threads is postponed until a new context is actually required, i.e., when interrupt thread blocks) could introduce some improvement in the system performance.

As the results of the experiments have shown, the *vanilla* kernel configuration is the most effective with respect to performance. None of the other configurations used in the conducted experiments achieved the same performance as the *vanilla* kernel. Also, the *vanilla* kernel was the only configuration where packet loss did not occur. In all the other configurations, packets started to be dropped at the higher transmission rates.

However, when the focus is predictability, the results obtained for the PREEMPT-RT kernel, configured with both threaded IRQs and the *Complete preemption* option, are the most interesting ones. Values for the *cyclictest* benchmark maximum latencies do not suffer variations from idle systems to moderately loaded system.

However, with higher packet transmission rates, the PREEMPT-RT kernel still has very high latencies. It is possible that the kernel network code is the responsible for these latencies but further analysis would need to be performed in order to confirm this and to identify the exact code. Other possibility is that the network cards device driver used in the experiments would require some additional modifications to take advantage of the improvements introduced to the kernel by the PREEMPT-RT patch.

The *cyclictest* benchmark itself can be configured to use an in-kernel trace tool, `ftrace`. This tool can be used to collect data for latencies tracing, which allows to identify the functions that are introducing the high latencies. This analysis would require to use a kernel compiled with the tracers and the re-execution of the tests. Although not in the scope of this article, this analysis is being performed and feedback to the PREEMPT-RT community will be provided. Eventually, the PREEMPT-RT patch will be modified with the results of this re-execution in order to improve these latencies.

Another topic that was out of the scope of this article, was the analysis of the impact of polling strategies in the ethernet device drivers used in the experiments. It would have been interesting to get a better understanding of the results collected by this analysis. The fact that the number of interrupts decrease with higher transmit rates indicates that the device driver uses the kernel NAPI interface (or similar technique) to poll the device instead of receiving one interrupt per packet. This technique is usually adaptive, i.e., traffic analysis is performed by the device driver and the number of interrupts per second is dynamically set based on the type of traffic being received.

Acknowledgments

I would like to thank all the people that have contributed to the Linux kernel, and in particular, to the PREEMPT-RT patch. This includes (but is not limited to): Andi Kleen, Arnaldo Carvalho de Melo, Daniel Walker, Gregory Haskins, Ingo Molnar, Luis Claudio R. Gonçalves, Peter Zijlstra, Steven Rostedt, Sven-Thorsten Dietrich, and Thomas Gleixner.

References

[1] "The linux kernel repository." [Online]. Available: http://www.kernel.org

[2] J. Corbet, "Approaches to realtime linux," *LWN.net*, October 2004. [Online]. Available: http://lwn.net/Articles/106010/

[3] "The preempt-rt patch website." [Online]. Available: http://www.kernel.org/pub/linux/kernel/projects/rt/

[4] "Wiki page for the preempt-rt patch." [Online]. Available: http://rt.wiki.kernel.org

[5] "The freebsd project website." [Online]. Available: http://www.freebsd.org

[6] T. F. D. Project, "Freebsd architecture handbook," 2006. [Online]. Available: http://www.freebsd.org/doc/en/books/arch-handbook/index.html

[7] S. Kleiman and J. Eykholt, "Interrupts as threads," *SIGOPS Oper. Syst. Rev.*, vol. 29, no. 2, pp. 21–26, 1995.

[8] M. Russinovich and D. A. Solomon, *Microsoft Windows Internals, Fourth Edition: Windows 2000, Windows XP, and Windows Server 2003*, 4th ed. Microsoft Press, 2004.

[9] M. Russinovich, "Advanced dpcs," 2006. [Online]. Available: http://technet.microsoft.com/en-us/sysinternals/bb963898.aspx

[10] "The l4 microkernel family website." [Online]. Available: http://os.inf.tu-dresden.de/L4/

[11] J. Liedtke, "On microkernel construction," in *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, Copper Mountain Resort, CO, December 1995. [Online]. Available: http://l4ka.org/publications/

[12] T. Gleixner and D. Niehaus, "Hrtimers and beyond: Transforming the linux time subsystems," in *Proc. Linux Symposium*, Ottawa, Ontario, Canada, July 2006.

[13] J. Corbet, "A new approach to kernel timers," *LWN.net*, September 2005. [Online]. Available: http://lwn.net/Articles/152436/

[14] S. Rostedt and D. V. Hart, "Internals of the rt patch," in *Proc. Linux Symposium*, Ottawa, Ontario, Canada, June 2007.

[15] "cyclictest website." [Online]. Available: http://www.kernel.org/pub/linux/kernel/people/tglx/rt-tests/

Towards Unit Testing Real-Time Schedulers in LITMUS$^{\mathrm{RT}}$

Malcolm S. Mollison, Björn B. Brandenburg, and James H. Anderson
Department of Computer Science, The University of North Carolina at Chapel Hill

Abstract

The problem of unit testing multiprocessor real-time schedulers in operating systems such as LITMUS$^{\mathrm{RT}}$ is discussed. A tool intended to aid debugging by identifying deviations from an intended scheduling policy and performance regressions is proposed. This paper gives a specification for the tool and also discusses ongoing work on a prototype implementation.

1 Introduction

The advent of multicore computing has led to renewed interest in real-time scheduling algorithms for multiprocessor systems. In research on this topic, the greatest emphasis has been on purely algorithmic issues. To impact the development of real systems, such work must be complemented by prototype development, so that scheduler-related overheads can be measured and the practicality of proposed scheduling algorithms assessed. The **LI**nux **T**estbed for **MU**ltiprocessor **S**cheduling in **R**eal-**T**ime Systems (LITMUS$^{\mathrm{RT}}$) project [2, 4, 8] was launched to enable such prototype-oriented research. LITMUS$^{\mathrm{RT}}$ provides a testbed for multiprocessor real-time schedulers by extending the Linux kernel to support the implementation of such schedulers as plugins.

Unfortunately, developers of real-time schedulers, including LITMUS$^{\mathrm{RT}}$ scheduler plugins, face two pressing challenges. First, it is very difficult to ascertain if a scheduler is actually making correct decisions, and thus is implemented correctly. For example, mistaken scheduling decisions may not result in deadline overruns for a particular benchmark task set, and thus may go unnoticed. Second, it is easy for scheduler developers to introduce extra, unwanted scheduling overhead. Such overhead will increase the degree to which the scheduler deviates from a desired scheduling policy.

These challenges are exacerbated by serious difficulties associated with multiprocessor operating system development. For example, concurrent programming issues such as synchronization, race conditions, etc., must be properly taken into account. Furthermore, collecting relevant data to aid debugging can be an obstacle in itself due to programming limitations commonly encountered in kernel environments.

Even if scheduler code is believed to produce correct schedules, each subsequent update to the code can introduce bugs. Therefore, there is a strong need for a tool that facilitates extensive testing of a scheduler against a number of developer-specified criteria after each update to the code. This requirement is captured by the notion of *unit testing* [6]—that is, programmatically testing the smallest possible units of a software system after each code revision. This is a widely-followed software-engineering practice [9]. In order for a testing tool to be of maximum use to developers, unit-test coverage should be as extensive as possible, and unit testing should be automated. The tool should also produce detailed unit-test results that can help developers identify and fix problems, including incorrect scheduling decisions, deadline overruns, and regressions in scheduling overhead.

In this paper, we present a specification for such a tool, and discuss both the current state of a prototype implementation and desired extensions.

Prior work. The Linux Test Project [7] has implemented a testing mechanism for the Linux scheduler, including the two POSIX-mandated real-time scheduling policies included in stock Linux, `SCHED_FIFO` and `SCHED_RR`. Like the proposed tool, the Linux Test Project performs scheduling overhead regression testing and conducts a series of checks that can alert developers to various problems. However, the Linux Test Project lacks a lightweight event tracing mechanism that would allow it to analyze individual scheduler decisions and provide very precise scheduling overhead analysis. (Our tool makes use of Feather-Trace [1, 5], a toolkit used by LITMUS$^{\mathrm{RT}}$ to record events.) Furthermore, from a real-time scheduling perspective, the Linux Test Project's scope is limited to only static-priority scheduling (`SCHED_FIFO` and `SCHED_RR`).

LITMUS$^{\text{RT}}$ currently supports a much wider range of real-time scheduling algorithms, including several global algorithms that have been the subject of much recent research. The development of LITMUS$^{\text{RT}}$ has thus far relied on purely manual techniques for debugging and testing of adherence to desired scheduling policies. Due to the scarcity of developer time, such testing is only conducted infrequently and involves running only a few simple, hand-crafted task sets. With the increasing complexity and maturity of LITMUS$^{\text{RT}}$, a more thorough testing effort is clearly warranted. Therefore, we desire a high-quality unit-testing tool for use with LITMUS$^{\text{RT}}$. To the best of our knowledge, the topic of testing of multiprocessor real-time scheduler *implementations* has not been considered in prior work.

The rest of this paper is organized as follows. In Section 2, we provide background on real-time scheduling on multiprocessor platforms, including LITMUS$^{\text{RT}}$ and associated tools. In Section 3, we provide a specification of the proposed tool. In Section 4, we describe progress to date on a prototype of the tool and planned future extensions. Finally, in Section 5, we conclude.

2 Background

The LITMUS$^{\text{RT}}$ operating system [2, 4, 8] executes task sets under a real-time scheduler selected by the user. Feather-Trace [1, 5] records scheduling decisions made by LITMUS$^{\text{RT}}$ in machine-readable, binary trace files. Our proposed tool, in turn, reads these files and performs analysis upon them under a unit-testing paradigm. This section provides necessary background on these topics.

2.1 Real-Time Task Model

A *real-time task set* consists of a number of tasks that are subject to real-time constraints. Tasks are invoked, or *released*, repeatedly during their lifetimes; each such invocation is known as a *job*. Under the *sporadic* task model, each release of a task must be separated by a minimum interval of time known as the task's *period*. Each task also has an associated *worst-case execution time* (WCET), which is an estimate of the execution time of any job of the task in the worst case. The ratio of each task's WCET and period defines its *utilization*. In this paper, we limit attention to sporadic tasks with *implicit deadlines*, *i.e*, each job is expected to complete before the end of its period; otherwise, it has exceeded its deadline and is considered *tardy*. A job is *eligible* if it has been released, is unfinished, and the previous job of the same task has completed execution.

2.2 Real-Time Scheduling Policies

A *scheduling policy* is an algorithm used by a *scheduler* at runtime to determine how to allocate available cores[1] to jobs.

Under the *global earliest-deadline-first* (G-EDF) policy, contending jobs are prioritized in order of non-decreasing deadlines. Hence, on an m-processor system, at any time, if there are more than m eligible jobs, then the m eligible jobs with the earliest deadlines should be executing, with ties broken consistently according to some (perhaps implementation-specific) policy. In order to meet this condition, the scheduler can preempt jobs or migrate them to different cores.

Our prototype unit-testing tool only supports the analysis of G-EDF, though support for other policies is planned. These include *partitioned earliest-deadline-first* (P-EDF) scheduling, under which each task is assigned to a particular core and only contends for execution time with other tasks assigned to the same core; *clustered earliest-deadline-first* (C-EDF) scheduling, under which tasks are assigned to user-defined clusters of cores; and the *PD*^{2} scheduling algorithm [10], under which tasks make progress at a rate proportional to their utilizations.

2.3 LITMUS$^{\text{RT}}$

The LITMUS$^{\text{RT}}$ operating system provides a platform for executing real-time task sets according to a real-time scheduling policy. LITMUS$^{\text{RT}}$ is implemented as a patch to the Linux kernel,[2] modifying its scheduling facilities to allow for the use of more specialized real-time scheduling policies than those that are included in stock Linux. LITMUS$^{\text{RT}}$ scheduler plugins have been written for the G-EDF, P-EDF, C-EDF, and PD^2 scheduling policies (among others).

2.4 Feather-Trace

Feather-Trace is a light-weight event tracing toolkit. It allows scheduler code in the kernel to buffer binary data and make it available for asynchronous export to userspace, from whence it can be written to disk. Feather-Trace is designed to incur very little overhead, so that time-critical code can record data without being disrupted.

In LITMUS$^{\text{RT}}$, this mechanism can be used to record scheduler events (for example, the release or completion of a job) in trace files. Each event record is stored in a special binary format and includes a precise timestamp.

[1]In this paper, we denote any logical CPU available for scheduling by the OS as a "core."

[2]The latest release patches kernel version 2.6.24.

These records can later be retrieved and examined by analysis tools, such as our unit-testing tool.

3 Specification

The goal of our proposed tool is to help real-time scheduler developers produce correct scheduler code for the LITMUS^{RT} system. Towards this end, the tool will conduct a series of tests to determine if desired criteria are met by a scheduler. An error will be reported when a criterion is not met, and details will be provided to help developers resolve the problem.

In typical unit testing, modules of source code are executed in a test framework. Each module is executed with pre-specified input and checked for expected output. If unexpected output is produced, then the test framework reports an error. This approach works well for source code that can be factored into small, modular components that accept well-defined inputs and give predictable outputs.

Scheduler code is not amenable to such piecemeal testing. This is due to three fundamental reasons. First, the absence of errors when testing individual components for *logical* correctness does not imply that the system as a whole exhibits *temporally* correct behavior (for example, scheduling errors due to race conditions cannot be found by piecewise testing). Second, scheduling code tends to rely heavily on side effects and external entities such as hardware timers; such dependencies are hard to emulate correctly in a traditional unit-testing environment. Third, scheduling code is invoked from many different code paths within the kernel since the scheduler constitutes the very core of the OS; it is infeasible to recreate appropriate system and task state for all possible invocations. In essence, traditional unit-testing techniques rely heavily on the tested system being well-structured and modular, whereas scheduling code tends to (implicitly) interact with large parts of the whole system. Furthermore, testing on a per-module basis would not allow for accurate sampling of scheduling overhead.

To work around these concerns, the proposed tool executes unit tests offline against records of scheduler events as they occurred on *real hardware* in the *actual system*. To automate the testing process, the generation and execution of task systems can also be automated.

This approach permits testing on any task system for a supported scheduling policy, alleviating the need to provide rigid, module-specific test input. It also allows for accurate measurement of scheduling overhead.

As with typical unit testing, errors will be reported when desired criteria are not met. Information to help developers correct bugs will be included with each error. If all tests complete without errors, developers can conclude that the scheduler behaved correctly for the criteria being tested.

3.1 Architecture

As depicted in Figure 1, the proposed tool consists of four components: the Driver, the Parser, the Validator, and the Report Generator.

Driver. The Driver automates the testing process, facilitating unit testing after each incremental update to the scheduler code. Each testing session begins with the Driver reading a developer-supplied configuration file that specifies task systems to execute and a suite of unit tests. Any unit-test tuning variables (for example, acceptable tardiness bounds) are also specified in the configuration file. The Driver executes the specified task system under LITMUS^{RT} with Feather-Trace recording enabled. When the task system completes execution, the Driver invokes the Parser, Validator, and Report Generator in sequence. For more thorough testing, the Driver can repeatedly generate and test random task systems (generated according to distributions specified in the configuration).

Parser. Feather-Trace exports scheduler event records to a buffer in a compact format to avoid incurring unnecessary overhead. The contents of this buffer are later written to a set of trace files. The Parser reads the trace files, extracting event records and storing them in an easy-to-use format in memory. The in-memory records are represented as a "stream" data structure, which the Validator and Report Generator iterate over in the course of performing their work.

Feather-Trace records the following types of events.

- Release: A job is released.
- Switch To: A job begins executing.
- Switch Away: A job stops executing.
- Completion: A job completes.
- Block: A job blocks. Even in task systems without synchronization requirements, blocking can be caused by the Linux I/O subsystem.
- Resume: A job resumes, after being blocked.

Each scheduling event record identifies the job and task it is associated with and includes a timestamp indicating the time of the event. Records for events associated with a particular core include the logical CPU number for that core.

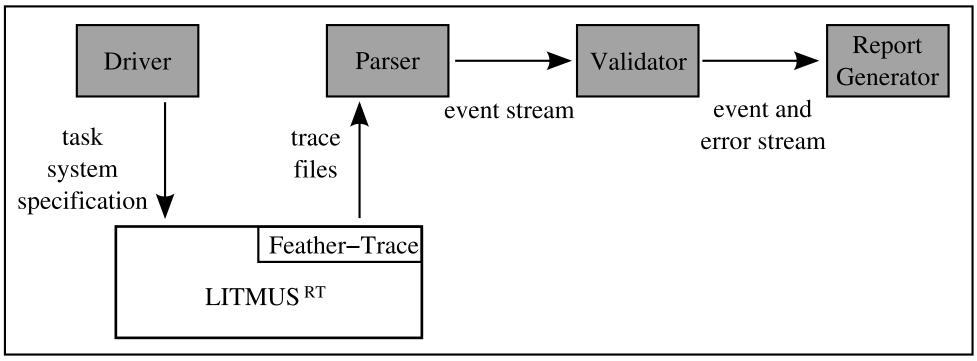


Figure 1: Data flow between components of the proposed tool (shaded) and LITMUS$^{\mathrm{RT}}$.

Additional Feather-Trace records specifying the WCET and period for each task are recorded at the beginning of task system execution.

Validator. The Validator executes each unit test, using as input the scheduler event stream. Each unit test produces an error record for each error it finds. Each error record includes the time of the scheduler event that triggered the error. This allows the Report Generator to display errors in the context of surrounding scheduling events. The Validator outputs a stream of all error records produced by the unit tests.

Report Generator. The Report Generator receives as input the event and error streams. It generates meaningful output to help developers correct bugs and detect problematic scheduling latency. A plugin system allows for output in various formats. For example, plugins to support HTML output or graphical output could be developed.

3.2 Unit Tests

The current needs of LITMUS$^{\mathrm{RT}}$ have motivated the creation of algorithms for a number of unit tests.

Completion Test. The Completion Test checks whether all released jobs actually complete. That is, for each Release record in the scheduler event stream, there should be a Completion record corresponding to the same job. An error is produced for any released jobs that do not complete.

Sporadic Task Model Test. The Sporadic Task Model Test checks whether all jobs of the same task are separated by at least the period of the task.

The algorithm for this unit test is straightforward: for each Release record in the event stream, if there is a previous Release record of the same task, then a check is made to ensure that the separation time between them is at least the corresponding task's period.

Some premature releases are expected in the case that periodic releases are desired, since scheduling overhead will delay releases by a variable, nonzero amount of time. To accommodate this, a "tolerance" tuning variable can be specified in the Driver configuration file. Releases that are premature by an amount of time less than the tolerance value will not cause an error to be generated.

Deadline Test. The Deadline Test checks to see if jobs meet their deadlines. Once again, the algorithm for the test is straightforward: for each Release record, there must be a Completion record for the same job, and the difference in their timestamps must be at most the period for the corresponding task.

A tolerance variable, similar to the one used for the Sporadic Task Model Test, allows developers to specify an acceptable tardiness bound.

G-EDF Decision Test. No scheduler can achieve true G-EDF scheduling in practice, due to scheduling overhead. For example, when a new job becomes eligible that has an earlier deadline than one of the jobs currently in execution, the new job should begin executing immediately. However, the overhead of halting the execution of the job to be preempted and starting execution of the new job takes a nonzero amount of time. Thus, testing for true adherence to the G-EDF policy would be an exercise in futility.

However, it still holds that whenever a scheduler does decide to switch execution to a particular job, that job should be one of the m earliest-deadline eligible jobs. The G-EDF Decision Test checks against this constraint. Successful validation of an event stream over both the Completion Test and G-EDF Decision Test assures developers that the scheduler allocated all eligible jobs in earliest-deadline-first order.

The G-EDF Decision Test algorithm models the state of the task system over the course of its execution. The

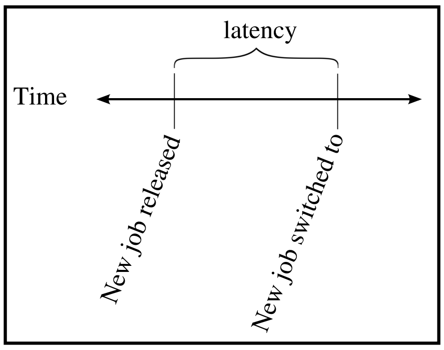


Figure 2: Context 1.

model consists of a list of executing jobs (up to m) and a list of released jobs that are eligible for execution. The algorithm progresses by iterating over the event stream, updating the model for each record. For example, a Switch To record indicates that a job began execution on a particular core; upon encountering such a record, the algorithm would add that job to the list of executing jobs.

The algorithm generates an error each time it determines that a job that did not have sufficiently high priority was scheduled for execution. A job has sufficiently high priority for execution if it is one of the m earliest-deadline eligible jobs.

G-EDF Latency Test. The G-EDF Latency Test helps developers locate areas of high overhead in scheduler code and quantitatively evaluate their efforts to optimize code to reduce overhead. It also may help developers discover bugs that do not violate the decision-making constraint checked by the G-EDF Decision Test, as these bugs often contribute to overhead.

Scheduler overhead causes departure from the G-EDF policy when a job becomes one of the m earliest-deadline eligible jobs but has not yet begun execution. Therefore, to measure G-EDF scheduling latency, we examine each such occurrence, and measure the specific latency components contributing to overhead. Note that this condition occurs prior to each switch in execution to a job, and thus once for each job release.

Which latency components are present depends on the context of the system when switching to a new job becomes necessary. There are three such contexts, explained in the following paragraphs.

In Context 1, as depicted in Figure 2, a job is released into a system with an idle core and is eligible (i.e., the previous job of the same task has completed). The only latency component—and, thus, the total scheduling overhead that occurs in this context—is the difference between the time of the release of the job and the time when it is switched to for execution on a core.

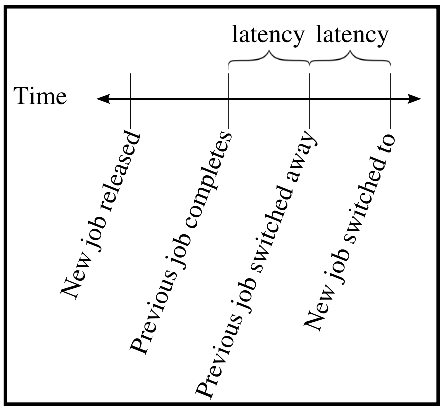


Figure 3: Context 2.

In Context 2, as depicted in Figure 3, a job is released into a system with no idle cores, but it is not one of the m highest-priority eligible jobs. When it becomes one of the m highest-priority eligible jobs due to the completion of another job, that job should be switched away from execution and the new job should commence execution. Scheduling overhead is the sum of two latency components. The first is the difference between the time of completion of the previously-executing job and the time when it is switched away from execution. The second is the difference between the time when the previously-executing job is switched away from execution, and the time when the new job is switched to for execution.

In Context 3, as depicted in Figure 4, a job is released into a system with no idle cores, but it is one of the m highest-priority eligible jobs. Another job should be switched away from a core and the new job should commence execution. Scheduling overhead is the sum of two latency components. The first is the difference between the time when the new job is released and the time when the preempted, previously-executing job is switched away from execution. The second is the difference between the time when the previously-executing job is switched away from execution, and the time when new job is switched to execution.

The goal of the G-EDF Latency Test is to measure each instance of overhead, classified as one of the latency components discussed previously. When latency greater than a desired threshold is discovered, it can be reported to the developer. Furthermore, statistical analysis can be performed. For example, a developer could use the latency measures to calculate the average amount of latency for a particular latency component over the course of execution of a task system. Historical data of this kind, accumulated from past testing, can be used to

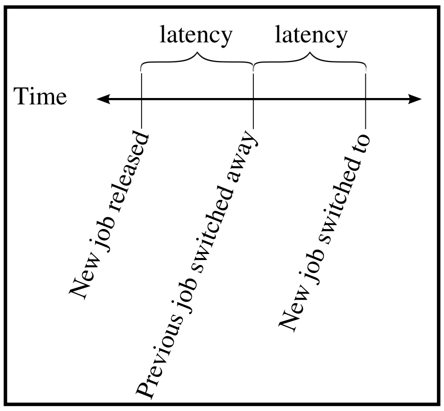


Figure 4: Context 3.

guard against regressions in scheduler performance.

Note that overhead potentially begins being accumulated when a job is released, and ends when the job is switched to. Thus, the G-EDF Latency Test algorithm iterates over the event stream, extracting latency information each time it encounters a release record.

The algorithm proceeds as follows. For each Release record, the corresponding Switch To record is found. This record is used to determine the CPU on which the events leading up to the beginning of execution of the job will occur. The events on that CPU from the time of the release to the time of the switch to execution match one of the three contexts. Once the context is identified, extracting the specific latency measures is straightforward.

Fortunately, each context represents a unique set of scheduler events, making it possible to identify the context for a particular release from the scheduler event stream.

- Context 1 is indicated by a Release record followed by a Switch To record.
- Context 2 is indicated by a Release record followed by a Completion record.
- Context 3 is indicated by a Release record followed by a Switch Away record.

The Driver configuration file allows developers to specify a large number of criteria for the G-EDF Latency Test. In the simplest case, acceptable thresholds for latency components can be provided. An error is generated each time these values are exceeded.

4 Prototype Description and Future Extensions

In this section, we provide a summary of ongoing work on a prototype of the tool, and discuss desired future extensions of the tool.

4.1 Prototype

Most of the functionality described in the specification has been successfully implemented in our prototype, including each of the unit tests discussed in this paper. The code has been made available online [3]. The prototype is implemented in the Python programming language, which was chosen for its ease of use and will be retained for the official release version of the tool.

The prototype does not yet support some of the more advanced proposed features, as discussed below, though support for them is planned for future releases.

The Driver does not automatically create and test randomized task systems; instead, a specific task system must be provided by the user. This feature would be useful for testing schedulers over a wide variety of task systems, increasing the probability of finding bugs that occur only in rare or specialized cases.

The Report Generator does not yet provide support for creating a graphical depiction of scheduling events, which would aid developers in understanding bugs.

The G-EDF Latency Test does not perform any automatic statistical validation or regression checking. For example, automatic calculation of average latency for each latency component over the course of execution of a task system is desired. These values could automatically be compared to ideal values provided by the user.

4.2 Future Extensions

Significant expansion of the tool beyond the specification described in this paper is desired. Most importantly, support for testing additional scheduling policies beyond G-EDF—in particular, C-EDF, P-EDF, and PD^2—is planned.

Supporting C-EDF and P-EDF will be straightforward. C-EDF unit-test algorithms will not be notably more complex than the G-EDF algorithms provided in this paper. As P-EDF is a special case of C-EDF, a separate unit tests for P-EDF will not be needed.

However, PD^2 will present a significant challenge, and will require the development of a large number of additional and complex unit tests. The need to debug PD^2 schedulers was the most pressing motivation for the development of the tool, but G-EDF was chosen as a more easily achievable initial target. Lessons learned in developing the overall framework for the unit tester and the G-EDF unit tests will likely prove invaluable in the extension of the tool to allow for testing of PD^2 schedulers.

In addition, the current specification for the tool does not account for jobs that block. A tool for testing real-time schedulers running under LITMUS$^{\mathrm{RT}}$ needs to account for blocking, since it can be caused by some synchronization protocols as well as the Linux I/O subsystem. Fortunately, extending the specification to account for blocking will be straightforward. Most of the unit tests remain the same, though a slight change to the model used in the G-EDF Decision Test is necessary. The most significant change will be the need to analyze new latency components in the G-EDF Latency Test.

5 Conclusion

We have established the need for an automated tool for testing multiprocessor real-time schedulers in LITMUS$^{\mathrm{RT}}$. The tool should uncover incorrect scheduling decisions. It should also allow for rigorous analysis of scheduling overhead. We have presented a specification of such a tool, and have discussed ongoing work on a prototype. Finally, we have described significant desired extensions of the tool beyond the features provided for in our current specification.

References

[1] B. Brandenburg and J. Anderson. Feather-Trace: A Light-Weight Event Tracing Toolkit. In *Proceedings of the Third International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 20-27, 2007.

[2] B. Brandenburg, A. Block, J. Calandrino, U. Devi, H. Leontyev, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A Status Report. In *Proceedings of the 9th Real-Time Linux Workshop*, pages 107-126, 2006.

[3] LITMUS$^{\mathrm{RT}}$ Scheduler Testing.
Homepage. http://cs.unc.edu/˜mollison/unit-trace/.

[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*, pages 111-123, 2006.

[5] Feather-Trace.
Homepage. http://www.cs.unc.edu/˜bbb/feathertrace/.

[6] IEEE Standard for Software Unit Testing. 1986.

[7] The Linux Test Project.
Homepage. http://ltp.sourceforge.net/.

[8] The LITMUS$^{\mathrm{RT}}$ Project.
Homepage. http://www.cs.unc.edu/˜anderson/litmus-rt/.

[9] P. Runeson. A Survey of Unit Testing Practices. In *IEEE Software*, 23(4):22-29, 2006.

[10] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. *Journal of Computer and System Sciences*, 72(6):1094-1117, September 2006.

Exception-Based Management of Timing Constraints Violations for Soft Real-Time Applications. [*]

Tommaso Cucinotta, Dario Faggioli
Scuola Superiore Sant'Anna, Pisa (Italy)
`{t.cucinotta, d.faggioli}@sssup.it`
Alessandro Evangelista
`mail@evangelista.tv`

Abstract

This paper presents an open-source library for the C language supporting the specification and management of timing constraints within embedded soft real-time applications. The library provides a set of well-designed C macros that allow developers to associate timing constraints to code segments, and to deal with their violations through the well-established practise of exception-based management.

After a brief overview of the requirements motivating the work, the exceptions library is presented. Then, the paper focuses on the specific macros that deal with the specification of deadline and execution-time constraints, with a few notes on how the library has been implemented.

Finally, a few experimental results are shown in order to discuss the features and limitations of this approach, with the current implementation (on Linux) that relies almost completely on POSIX-compliant system calls.

1 Introduction

General Purpose Operating Systems (GOPSes) are being enriched with more and more support for soft real-time applications, allowing for an easier development of applications with stringent timing requirements, such as multimedia and interactive ones. Still, one of the challenges for developers is how to specify timing constraints within the application, and how to properly design the application so as to respect them.

Furthermore, this kind of systems differ from traditional hard real-time ones, on a number of different points. First, a GPOS with a monolithic kernel cannot provide a precise scheduling of processes. Second, the typical knowledge, by developers/designers, of the main timing parameters of the application, such as the execution time of a code segment, is somewhat limited. In fact, it is not worth to recur to precise worst-case analysis techniques, and there is a need for using general-purpose hardware architectures (that are optimised for average-case performance, penalising predictability) and compression technologies (which cause the execution times to heavily vary from job to job, depending on the actual application data). Furthermore, in order to scale down production costs, a good resources saturation level is needed. Finally, timing requirements in this context may be stringent, but they are definitely not safe-critical, therefore it may be sufficient to fulfil them with a high probability.

Therefore, in such context, timing constraints violations should be expected to occur at run-time, and developers need to deal with these events by embedding appropriate recovery logic. This usually involves the correct use of timers and signals, something not always immediate.

This paper presents a framework that allows for adoption of the well-known *exception-based management* approach for dealing with timing constraints violations in C applications. The framework makes it possible to handle these events similarly to how exceptions are managed in languages that support such programming paradigm, e.g., C++, Java and Ada.

Specifically, two main forms of timing constraints can be specified: *deadline constraints*, i.e., a software component needs to complete within a certain (wall-clock) time, and *WCET constraints*, i.e., a software component needs to exhibit an execution time that is bounded. Also, the proposed solution allows for an arbitrary nesting of timing constraints. In fact, in the expected typical scenario, it is foreseen to have one deadline constraint at the outermost level, and one or more nested WCET constraints.

To the best of the authors' knowledge, no similar mechanism has been previously presented for the C language, with the same completeness of the one presented here, with no need to modify the C compiler, and only relying on standard POSIX features.

Paper outline After a brief overview of the related work in Section 2, Section 3 identifies the main technical requirements that need to be supported by the mechanism, then

[*]The research leading to these results has been supported by the European Commission under grant agreement n.214777, in the context of the IRMOS Project. More information at: http://www.irmosproject.eu.

Section 5 describes the POSIX-based implementation realised for the Linux OS, finally a few experimental results are presented in Section 6 highlighting the impact of the Linux kernel configuration on the mechanism precision. Finally, conclusions are drawn in Section 7 along with directions for future work.

2 Related Work

The need for having more and more predictable timing behaviour of system components is well-known within the real-time community, to the point that modern general-purpose (GP) hardware architectures are deemed as inappropriate for dealing with applications with critical real-time constraints. In fact, there exist such approaches as Predictable Timed Architecture [3], a paradigm for designing hardware systems that provide a high degree of predictability of the software behaviour. However, such approaches are appropriate for hard real-time applications, but cannot be applied for predictable computing in the domain of soft real-time systems running GP hardware. Yet, the concept of deadline exception has been actually inspired by the concept of deadline instruction as presented in [9].

Coming to software approaches relying on the services of the Operating System (OS) and standard libraries, the POSIX.1b standard [5] exhibits a set of real-time extensions that suffice to the enforcement of real-time constraints, as well as to the development of software components exhibiting a predictable timing behaviour. However, working directly with these very basic building blocks is definitely non-trivial. The code for handling timing constraints violations, as well as other types of error conditions, needs to be intermixed with regular application code, making the development and maintenance of the code overly complex. As it will be more clear later, the proposed framework improves usability of these building blocks, by enabling the adoption of an exception-based management of these conditions.

Such an approach is not new, in fact it is used in other higher-level programming languages, such as Java, with the Real-Time Specification for Java (RTSJ) [1] extensions. These, beyond overcoming the traditional issue of the unpredictable interferences of the Garbage Collector with normal application code, also include a set of constructs and specialised exceptions in order to deal with timing constraints specification, enforcement and violation.

Also, the Ada 2005 language [2] has a mechanism that is very similar to the one presented in this paper, namely the Asynchronous Transfer of Control (ATC), that allows for raising an exception in case of an absolute or relative deadline miss, and/or of a task WCET violation, that cause a jump to a recovery code segment.

However, the focus of this paper is on the C language, probably still the most widely used language for embedded applications with high performance and scarce resource availability constraints. By making such a mechanism easily and safely available in C, the work presented in this paper contributes in enriching the C language with an essential feature useful for the development of real-time systems.

Focusing on the C language, the RTC approach proposed by Lee et al. [7] is very similar to the one that is introduced in this paper. They theorised and implemented a set of extensions to the C language allowing one to express typical real-time concurrency constraints at the language level, and deal with the possible run-time violations of them, and treat these events as exceptions. However, while RTC introduces new syntactic constructs into the C language, requiring a non-standard compiler, this paper presents a solution based on a set of well-designed macros that are C compliant and may be portable across a wide range of Operating Systems. Furthermore, RTC explicitly forbids nesting of timing constraints, while the approach presented in this paper does not suffer of such a limitation.

Finally, the concept of *time-scope* introduced in [8] is also similar to the "try_within" code block that is presented in this paper. However, that work is merely theoretic and language-independent, and it does not present any concrete implementation of the mechanism.

3 Requirements Definition

The basic requirements that drive the work of this paper are presented here as drawn out by a simple example: a multimedia, component based application, designed as a single thread of execution[1] activated periodically or sporadically. For example, consider the *Video Decoder* application, whose behaviour is outlined in the UML Activity Diagram of Figure 1.

From a design level perspective, as *Video Decoder* will be co-scheduled with other applications, it would be highly desirable to characterise each component with such typical information: (1) WCET (or an appropriate statistic of execution time distribution); (2) relative or absolute deadline; (3) minimum period of activation. Also, it might be desirable that *Video Decoder* actually respects both the declared WCET and the deadline constraint, also in cases of overload, e.g., when a frame is particularly difficult to decode.

Now, assume that a *Frame Decoder* is used in the main loop of *Video Decoder*. Due to the in-place timing requirements, it would be useful to characterise *Frame Decoder* invocations with the WCET to be expected at run-time. In fact, as shown in Figure 1, such information, plus the WCETs of the *Stream Parser*, *Filtering* and *Visualization* components, sum up to the WCET of the *Video Decoder* itself. However, video decoding architectures are highly modular, and make heavy use of third-party video and audio decoding plug-ins, e.g., depending on the stream format.

[1]For example, the `fflay` player, part of the widely used open-source `ffmpeg` project, is designed as a single threaded application.

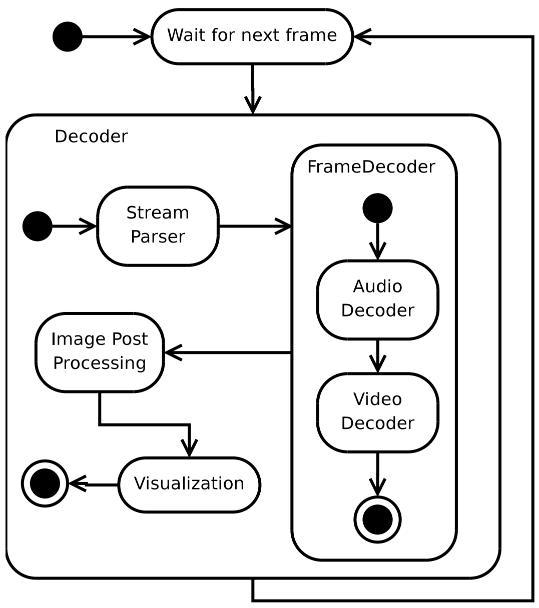


Figure 1. UML Activity diagram for the example video decoder thread.

Thus, in order to allow for an appropriate use of *Frame Decoder* within real-time applications, it would be highly desirable for libraries developers to have a WCET estimation such that either: (1) the decoding operation terminates within the WCET limit, or (2) it is aborted.

The approach that is envisioned in this paper is aimed at simplifying design of such a complex software, and it is based on the adoption of an exception-based programming paradigm. A timing constraint violation is seen as an exceptional situation whose occurrence must be foreseen by the programmer, without necessarily subverting the flow of control that is normally realised.

However, it is clear that the possibility for the program to jump asynchronously to exception handling code segments is not something that may be seamlessly incorporated within an application. The latter should be designed so as to tolerate this kind of operation abortion, so as to not introduce memory leaks, and to properly cleanup any resources that might be associated with the aborting code segment. For example, a multimedia encoding/decoding library in which all the needed buffers are allocated at initialisation time, and the encoding/decoding functions only operate on these buffers (without any memory allocation nor mutex locks acquisition), the encoding/decoding functions may probably be safely asynchronously aborted. If this is not the case, then one should generally modify the code so as to catch the deadline exception at appropriate points in the code, so as to trigger the proper cleanup logic.

From the above sketched example, the following set of high-level requirements may be identified for the proposed mechanism.

Requirement 1 *it should be possible to associate a deadline constraint to a code segment, either specifying relative or absolute time values;*

Requirement 2 *it should be possible to associate a WCET constraint to a code segment;*

Requirement 3 *when a timing constraint is violated, it should be possible to activate appropriate recovery logic that allows for a gracefully abort of the monitored code segment; also, it should be possible for the recovery code to either be associated to a generic timing constraint violation, or more specifically to a particular type of violation (deadline or WCET);*

Requirement 4 *it should be possible to use the mechanism at the same time in multiple applications, as well as in multiple threads of the same application;*

Requirement 5 *nesting of timing constraints should be allowed, at least up to a certain (configurable) nesting level. In fact, this is a key feature for component based design of real-time applications. For example, not only* Video Decoder *should be associated with overall deadline and WCET constraints, but also* Frame Decoder *should be associated with its own WCET constraint;*

Requirement 6 *it should be possible to cancel a timing constraint violation enforcement if the program flow runs out of the boundary of the associated code segment, e.g., when it ends normally or when another kind of exception requests abort of the code segment;*

Requirement 7 *the latency between the occurrence of the timing constraint violation and the activation of the application recovery code (from here on referred to as* handler activation latency*) should be known to the designer/developer, and it should be possibly negligible with respect to the task execution time;*

Requirement 8 *the mechanism should allow the programmer to specify some "protected" section of a code segment that will never be interrupted by a timing constraint violation notification. Thus, if that happens, the execution of recovery code would be delayed while inside such a section;*

Requirement 9 *the mechanism could provide support for gathering benchmarking data of the code segments, instead of enforcing their timing-constraints. This operational mode could be enabled at compile time, and used for tuning the actual parameters used as timing constraints for the various code segments;*

Requirement 10 *the mechanism could be portable to as many Operating Systems as possible.*

4 Proposed approach

Here a mechanism complying with the above enumerated requirements is presented, with a focus on the programming paradigm and syntax. First, the generic framework for exceptions handling for the C language is presented. Then, the extensions for dealing with timing constraints violations are presented. Finally, for the sake of completeness, a few implementation details are discussed.

4.1 Exceptions for the C language

The framework for exception-management for the C language is distributed as part of the open-source project Open Macro Library (OML) [2], whose description is out of the scope of this paper. OML Exceptions supports hierarchical arrangement of exceptions, where all exceptions must derive from the common "type" `exception`. The syntax of such framework (from here on referred to as OML Exceptions) comprises the following macros:

`define_exception...extends`: this macro may be used to define new application-specific exceptions;

`try`: this macro delimits the code segment subject to exception handling;

`finally`: this macro identifies the code segment that will be executed both in case of exception and normal `try` termination;

`handle...handle_end`: these two macros must enclose the (optional) `when` clauses;

`when`: this macro identifies the code segment that is executed in reaction to each particular type of exception (or sub-type); the first matching `when` clause is the one that catches the exception; if no `when` clause catches the exception, then it is automatically re-thrown;

`re-throw`: this macro may be used to explicitly re-throw an exception from within a `when` clause, after it has been caught.

For example, Figure 2 shows an application that defines a custom exception, `ENotReady`, extending the `exception` basic "type", which is raised by using the `throw()` macro within the `foo()` function. Finally, the exception is caught by means of the `when()` macro within a `handle...end` block. The hierarchical arrangement of exceptions is useful for allowing a single `when` clause to specify a generic type and catch any exception descending from the specified one. As all exceptions derive from `exception`, a clause `when(exception)` may be used for catching any type of exception (similarly to the `catch(...)` syntax of C++).

[2]More information at: http://oml.sourceforge.net.

```
define_exception(ENotReady) extends(exception);

void foo() {
  if (cond)
    throw(ENotReady);
}

void bar() {
  try {
    /* Potentially faulty code segment */
    foo();
  } finally {
    /* Clean-up code executed both on normal
     * termination _and_ on exception */
  }
  handle
    when (ENotReady) {
      /* Handle the ENotReady exception */
    }
    when (exception) {
      /* Handle any exception that is not
         of ENotReady type nor sub-type */
    }
  handle_end;
}
```

Figure 2. Example code segment

Notice that OML Exceptions is both process and thread safe. Moreover, it allows exception throwing/catching code to be nested. However, due to how the macros are defined, there are a set of limitations in the use of them. For example, within `try` blocks, it is forbidden to use any C mechanism that would cause an attempt to cross the block boundary, such as `return` statements, `goto` statements (and also `longjmp` calls) with destinations outside the block, and `continue` and `break` statements referring to iterative loops enclosing the `try` block. The full discussion of these aspects is omitted for the sake of brevity.

4.2 Timing Constraints Based Exceptions

OML Exceptions includes a support for notifying timing constraints violations by means of the following constructs[3]:

`try_within_abs`: this macro allows for starting a `try` block with an absolute deadline constraint;

`try_within_rel`: convenience macro macro useful for specifying a relative deadline constraint, however the effect of a relative deadline expiring is not distinguishable from the one of an absolute deadline expiring (see note at end of section);

`try_wcet`: this macro allows for starting a `try` block with a maximum allowed execution time (WCET);

[3]This support is available within the `dlexception` branch on the CVS repository of the OML project.

```
#include <oml_exceptions.h>

void Decoder() {
  next_deadline = current_time();
  for (;;) {
    next_deadline += period;

    /* absolute deadline constrained code */
    try_within_abs(next_deadline) {
      StreamParser();
      if (FrameDecoder() == 0)
        ImagePostProcessing();
      Visualization();
    }
    handle
      when (ex_deadline_violation) {
        /* e.g., re-use last decoded frame */
      }
    handle_end;
    /* Wait for next activation */
  }
}

int FrameDecoder() {
  int rv = 0;   /* Normal return code */

  try_wcet(12000) {
    DecodeAudioFrame();
    DecodeVideoFrame();
  }
  handle
    when (ex_wcet_violation) {
      /* Notify caller of incomplete decoding */
      rv = -1;
    }
  handle_end;

  return rv;
}
```

Figure 3. Example code of an video/audio player using the proposed mechanism.

ex_timing_constraint_violation: this is the common basic type for timing constraint violation exceptions; it may be used for the purpose of catching a generic timing constraint violation, without distinguishing between them;

ex_deadline_violation: this exception occurs as a result of a try_within_rel or try_within_abs segment not terminating within the specified relative or absolute deadline;

ex_wcet_violation: this exception occurs as a result of a try_wcet segment not terminating within the specified execution time constraint.

A simple example of how to use these macros is shown in Figure 3. The Decoder main body is a typical periodic thread where each activation has the next activation time as absolute deadline. Also, the FrameDecoder() function has a nested WCET constraint of $12ms$.

As a final remark, consider the particular erroneous usage of the framework shown in Figure 4. If the try_within statements are used in different components, then such a situation may occur during development. OML Exceptions includes a special exception matching rule for the when clauses involving timing constraint exceptions: if the raised exception is associated to a try block that is external (in the run-time sense) to the current try block, then the exception is propagated instead of being stopped. In the example, this mechanism allows the outer handler to correctly detect the deadline violation, because it is not stopped by the nested handler.

```
try_within(10) {
  ...
  try_within(50) {
    ...
  }
  handle
    when (ex_deadline_violation) {
      /* handle violation of try_within(50)  *
       *  and not the one of try_within(10)  *
       *  which is the first that is raised. */
    }
  handle_end;
}
handle
  when (ex_deadline_violation) {
    /* handle the violation of try_within(10) *
     *  which has to be captured here and not *
     *  in the previous when clause.          */
  }
```

Figure 4. Typical example where relating each try clause with its when is needed.

OML Exceptions complies with all of the requirements introduced in Section 3, with the few notes outlined in the following section.

5 Implementation

This section provides an overview of how the proposed mechanism has been implemented, always bearing the outlined requirements in mind.

5.1 Time-Scoped Segment Implementation

OML Exceptions has been realized by means of the POSIX sigsetjmp() and siglongjmp() functions. The former saves the execution context such that the latter is able to restore it, and continue program execution from that point.

For the try_within_abs and try_within_rel constructs, the time reference is the POSIX

`CLOCK_MONOTONIC` clock. For the `try_within_wcet` macro, the time reference is the POSIX `CLOCK_THREAD_CPUTIME_ID` clock. Events are posted using interval timers (POSIX `itimer`).

Notification of asynchronous constraint violations is done by delivering to the faulting thread a real-time signal (i.e., a POSIX signal with the property of being queued and guaranteed not to be lost). The OML Exceptions signal handler performs a `siglongjmp` to the appropriate context, jumping to the `handle...handle_end` block for the check of the exception type.

This implementation is portable to any Operating System providing support for POSIX real-time extensions.

5.2 Deadline and WCET Signal Handling

In case one (or more) specified constraint is violated, a signal has to be sent to the *correct* thread, in order to fulfil Requirement 4. However, signal delivery to a specific thread is not covered by POSIX. In fact, when a signal is sent, it reaches the whole process, and it is not possible to determine in advance which thread will receive and handle it. Therefore, the standard suggests to have one only thread receiving the signal, and all the others ignoring it, so that the receiving thread may notify the correct thread by means of other inter-thread synchronization primitives. However, such an approach would imply that every time a timing constraint is violated, the CPU incurs additional context switches, not to mention the additional overheads of managing (creating and destroying) the "signal router" thread.

On the other hand, Linux supports delivery of signals to specific threads thanks to an extension of the POSIX semantics built into the kernel. Therefore, OML Exceptions is implemented by using this extension, which, at the cost of sacrificing Requirements 10, allows for a much more efficient implementation of the mechanism on Linux (see also Figure 5). However, a version of OML Exceptions perfectly compliant with POSIX is being implemented as well, so that the framework will be capable of choosing the best implementation at compile-time.

5.3 Benchmarking Operational Mode

In order to cope with requirement 9, a compile-time switch has been provided that, when enabled, gathers information on the duration of all the `try...handle` code segments. This allows developers to easily obtain statistics about execution times of the time-scoped sections.

5.4 Non-Interruptible Code Sections

Requirement 8 is achieved by providing two additional macros, `oml_within_disable` and `oml_within_enable`, within which developers may enclose *atomic* code segments that cannot be asynchronously interrupted by a timing constraint violation. These two macros simply disable and enable, respectively, delivery of the time constraint violation real-time signals. If a signal occurs in the middle of such a protected code region, then it is enqueued by the OS, and delivered immediately at the end of the section.

5.5 Precision Limitations and Latency Issues

With respect to the maximum precision with which timing constraints are checked and enforced, this is limited by the time-keeping precision of the underlying Operating System. This is true also for the preliminary implementation on Linux, and thus a description of how timers and task execution time accounting are dealt with in the Linux kernel follows.

From mainstream kernel version 2.6.21, the kernel has been enriched by the high resolution timers. Thanks to them, timers are no longer coupled with the periodic system tick, and thus they can achieve as high resolution as permitted by the hardware platform. Nowadays, large number of microprocessors, either designed for general purpose or embedded systems, are provided with precise timer hardware that the OS can exploit, e.g., the TSC cycle counter register of the CPU. Therefore, if a Linux process or thread posts a timer to fire at a certain instant, it could expect to be woken up quite close to that point in time.

Despite this, there still exist Linux kernel subsystems depending on the periodic system tick. With respect to the presented work, the most relevant one is the time accounting mechanism, i.e., how the system tracks how much a thread is executing. In fact, this is done by the kernel at each occurrence of the following events:

- at each periodic system tick;
- at each task scheduling event, i.e., enqueue, dequeue or preemption.

Thus, the time accounting resolution is limited by the system tick frequency, which can be configured by the user at kernel compile time. Typical values are 100, 250 and 1000 Hertz, which means, respectively, 10, 4 and 1 millisecond resolution. This is also important, since the CPU-clock based timers used to implement the WCET timing constraints are not based on high resolution timers, and rely only on standard Linux accounting.

6 Experimental Evaluation

The proposed mechanism is effective and useful only if the latency between the occurrence of the timing constraint violation and the activation of application recovery logic (handler activation latency) is relatively small (with respect to the job execution times of the application), and if its value is known to the designer (Requirement 7).

In Figure 5 the various components contributing to the total amount of latency introduced between an actual con-

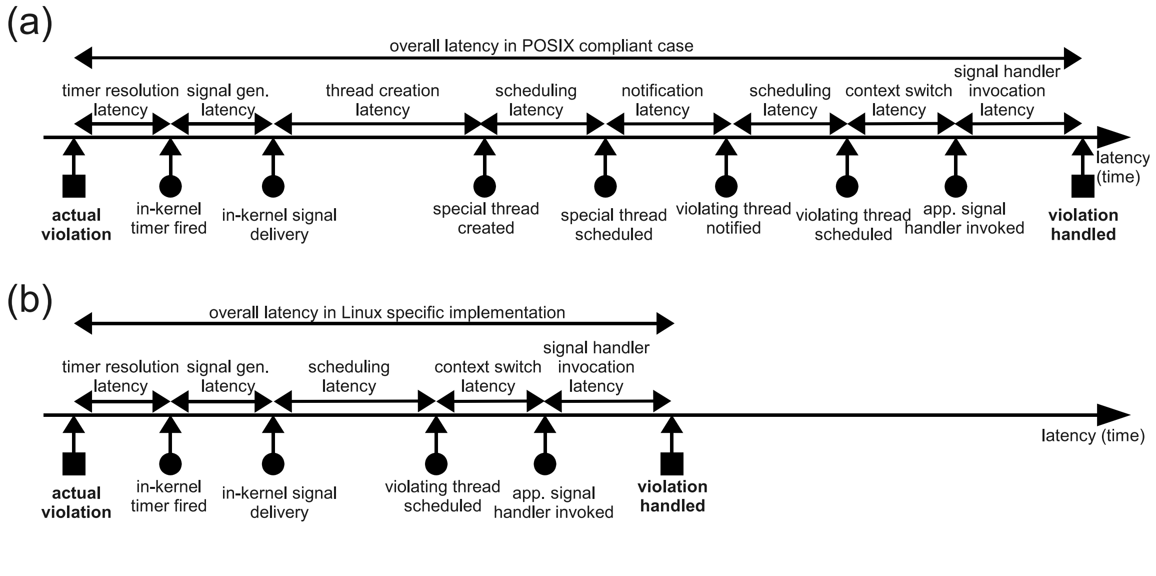


Figure 5. Various components contributing to the handler activation latency for the POSIX compliant (a) and Linux specific (b) implementations.

straint violation and its notification to the application are shown.

The handler activation latency has been measured by means of two experiments, made on the Linux Operating System (OS), that also highlight how the latency is affected by the kernel configuration.

6.1 Experiments Set-Up

A simple test application, built as a single thread of execution, has been used in both experiments, and no other applications have been launched concurrently. This way, the measurements were not affected by other components of the handler activation latency, such as the scheduling latency (the latter should be addressed by adopting a real-time design strategy). Thus, a task τ with WCET, relative deadline and period equal to $(C, D, T) = (50, 50, 100)\ msec$ is used, and run for 1000 consecutive instances. Experiments have been performed on commonly available desktop PC hardware, with 3.0 GHz Intel CPU and 2 GB of RAM. Debian GNU/Linux version 4.0, with hand-tailored kernel version 2.6.28 was the Operating System used. Kernel configuration includes the high-resolution timer subsystem, with support for high precision hardware timing sources. The one used by the system while running the experiments was the HPET [6].

In the first experiment, the latency of a deadline violation is measured 1000 times. This is done by forcing τ to execute more than 50 $msec$ inside a `try_within` block, and then subtracting the ideal deadline violation instance — $i \cdot T + D$ — from the actual time instant $\dot{D}$ at which the deadline miss signal handler is invoked.

In the second experiment, the task again executes more than 50 $msec$, this time from inside a `try_wcet` block, so to cause a WCET violation and measure its latency as well.

Both experiments have been run on three different configurations of the Linux kernel, i.e., with 100, 250 and 1000 Hertz as the periodic tick frequency, to study if and how this affects the latency.

Common statistics on the measured latency figures have been computed for both experiments on each configuration, and the corresponding cumulative distribution functions (CDF) are reported below. Minimum achieved latency values have not been reported since they are highly dependant on how close to a system tick (or, in general, an accounting event) a timing violation event occurs. Thus, since they depend on the actual alignment of the task and the OS events, they turn out to be unrelated to the system configuration, thus they provide few information about the performance of the mechanism.

6.2 Experiments Results

Results of the experiments are shown in Tables 1 and 2 and in Figures 6 and 7. They show that the latency of the notification of a deadline violation is both small and independent from the system tick frequency. In fact, values in Table 1 are comparable, and the three CDF in Figure 6 are completely superimposed. The measured latency values are in the order of the μs, what constitutes a more than acceptable performance.

Situation is different for WCET violation results. In fact,

	max	mean	std. dev.
HZ=100	28610	1724.418	1187.854
HZ=250	17202	1595.095	711.1304
HZ=1000	33394	1602.544	1023.255

Table 1. Deadline latency in ns

	max	mean	std. dev.
HZ=100	18727747	5748948.344	4474771.769
HZ=250	4423164	1233955.255	844593.486
HZ=1000	1999752	522228.673	390837.341

Table 2. WCET latency in ns

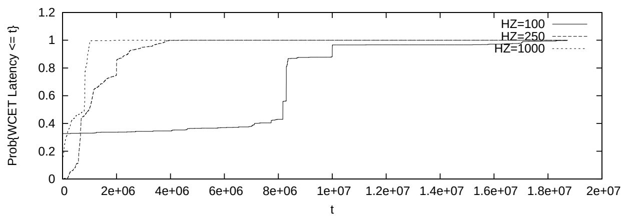


Figure 7. Cumulative Distribution Function of the WCET violation latencies. Time on x-axis in ns

as shown by both the values in Table 2 and the three CDF of Figure 7, the precision achieved in case of a WCET violation is tightly coupled with the system tick frequency HZ. Table 2 also shows how the mean WCET latency is close to $\frac{HZ}{2}$, which is exactly what was expected. As it can be easily seen, for the mechanism to be useful, the value of $HZ = 1000$ is strongly suggested.

7 Conclusions

In this paper, an open-source library has been presented for the management of timing constraints violations according to the well-known exception-based paradigm. This constitutes a valuable support for developers of embedded soft real-time applications.

A set of basic requirements have been identified, and a mechanism has been presented fulfilling them. The result is a framework designed as a set of macros for the C language, integrated in an open-source project that enables exception management.

Thanks to the proposed framework, developers may focus on one hand on the main application flow of control, which will be executed most of the times (or at least with a high probability, if the system is properly designed). On the other hand, the framework allows to catch dynamically possible violations of the timing constraints associated to critical parts of the code, due either to particular input data, or to the non-perfectly predictable behavior of applications on a soft real-time platform, such as Linux. The code in compensation or recovery of this is provided in the form of an exception handler.

An implementation of the proposed mechanism has been presented for the Linux Operating System, based on standard POSIX-class primitives. A few experimental results have been presented highlighting latencies that the applications using the framework experience in the activation of the exception management code, compared to the ideal time of fire of the exception, and the limitations on the precision of the current solution have been discussed.

Of course, such a framework should be used in combination with a real-time design methodology, and advanced real-time scheduling features, available in various forms for the Linux kernel, such as the POSIX Sporadic Server [4] or the Adaptive QoS Architecture for Linux [10].

Concerning possible directions for future work, a kernel-level mechanism is being investigated for the Linux OS, leading to a reduction in the handler activation latency. Furthermore, a more ambitious macro-based framework for C is under design that will enrich OML with generic constructs for threads management, synchronization and real-time scheduling.

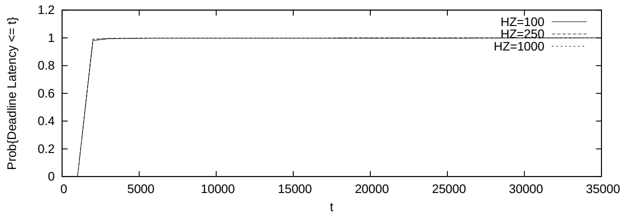


Figure 6. Cumulative Distribution Function of the deadline violation latencies. Time on x-axis is in ns.

References

[1] G. Bollella and J. Gosling. The real-time specification for java. *Computer*, 33(6):47–54, 2000.

[2] A. Burns and A. Wellings. *Concurrent and Real-Time Programming in Ada 2005*. Cambridge University Press, 2007.

[3] S. A. Edwards and E. A. Lee. The case for the precision timed (pret) machine. In *Proceedings of the* 44^{th} *annual conference on Design automation (DAC'07)*, pages 264–265, New York, NY, USA, 2007. ACM.

[4] D. Faggioli, G. Lipari, and T. Cucinotta. An efficient implementation of the bandwidth inheritance protocol for handling hard and soft real-time applications in the linux kernel. In *Proceedings of the* 4^{th} *International Workshop on Operat-*

ing Systems Platforms for Embedded Real-Time Applications (OSPERT 2008), Prague, Czech Republic, July 2008.

[5] IEEE. *Information Technology -Portable Operating System Interface (POSIX)- Part 1: System Application Program Interface (API) Amendment: Additional Realtime Extensions.* 2004.

[6] Intel. *IA-PC HPET (High Precision Event Timers) Specification (revision 1.0a).* October 2004.

[7] I. Lee, S. Davidson, and V. Wolfe. RTC: language support for real-time concurrency. In *Proceedings of the IEEE Real-Time Systems Symposium (RTSS 91)*, San Antonio, TX, USA, December 1991. IEEE *** FIXME ***.

[8] I. Lee and V. Gehlot. Language Constructs for Distributed Real-Time Programming . Technical report, University of Pennsylvania, May 1985.

[9] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee. Predictabl ision timed architecture. In *Proceedings of the International Conference on Compilers, Architecture, Synthesis for Embedded Systems (CASES)*, pages 137–146, Atlanta, Georgia, United States, October 2008.

[10] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari. AQuoSA — adaptive quality of service architecture. *Software – Practice and Experience*, 39(1):1–31, 2009.

Hardware Microkernels for Heterogeneous Manycore Systems

Jason Agron, David Andrews
Computer Science and Computer Engineering
The University of Arkansas
504 J.B. Hunt Building, Fayetteville, AR
{jagron,dandrews}@uark.edu

Abstract— **The migration away from power-hungry, speculative execution procesors towards manycore architectures is good news for the embedded and real-time systems community. Commodity platforms with large numbers of heterogeneous processors can be leveraged to meet the stringent real-time requirements of next-generation embedded systems. Although promising, the large numbers of heterogeneous cores that can bring about new levels of performance also bring with them new challenges for designers of real-time operating systems. Researchers are already conjecturing that the scalability and heterogeneity of manycores will result in the retirement of our existing monolithic operating system structure. In this paper we first discuss some of the challenges that these manycore chips will present. We then discuss how hardware-based microkernels can provide the low latency, low jitter performance envelopes that are needed within real-time systems. We show how our hthreads hardware microkernel provides these characteristics for scalable numbers of threads and processors. A hardware-based microkernel approach also provides ISA-neutral OS services, such as basic atomic operations, upon which higher-level programming models can be built.**

I. Introduction

In 2005, Paul Otellini announced Intel's commitment to dedicate all future product development to multicore designs. This announcement represented a key inflection point for the industry as it acknowledged the migration away from power hungry speculative execution processors, and the ushering in of the manycore era. The manycore era is good news for the embedded and real-time systems community. The manycore technology infrastructure will be providing new platforms with multiple, heterogeneous computational components that can be leveraged to meet real-time requirements. In addition to performance, power issues which have long been a first class design constraint for embedded and real-time systems, are now a driving factor for modern mobile Internet computing device architectures. The manycore philosophy is even effecting digital signal processing (DSP) architectures where vendors are integrating ever larger number of DSP cores within a chip.

Thus the manycore era should yield important economic advantages for embedded systems designers as it lessens the need for creating expensive application specific custom hardware solutions. Embedded systems designers will now have access to commodity chips with 100's to potentially 1,000's of heterogeneous components that can be used across wide application domains. While exciting, the entry of these components raises the challenge for designers of real-time operating systems. The real-time community has long been migrating away from custom towards more general purpose operating systems to enable portability and reuse. Researchers and practioners have been modifying general purpose operating systems to provide the scheduling and latency guarantees critical for real time systems. Interestingly, manycore architectures are now posing the same scheduling and latency issues for general purpose operating systems that have concerned the real-time systems community. Operating system scheduling and latency issues are being magnified by the very scalability and heterogenity benefits of the manycore architecture itself. Experts in both industry and academia are acknowledging that the manycore era may well spell the retirement of our historical desktop-based operating systems along with power-hungry super-scalar processors. The specific rationale for this viewpoint varies, but does converge on the limitations of our current operating system's monolithic structure. This monolithic structure poses capacity, scalability, and performance challenges that cannot be solved by simply adding yet more middleware and virtualization layers on top.

Manycore architectures can only bring about higher performance if parallelism is increased through the creation of more (and potentially heterogeneous) threads. New and more efficient implementations of OS services will be *fundamental* in enabling these new systems to transition into the real-time domain. Systems with heterogeneous components will also require new methods in enabling real-time operating systems to form a unifying virtual machine representation for application developers. Unfortunately, many of the proposed approaches from the general purpose computing domain, such as remote procedure calls (RPC) for resolving heterogeneity issues within existing monolithic operatings systems, introduce additional latency and jitter; two critical issues for real-time systems. In this paper we show how hardware-based microkernels can resolve the issues that result from our existing monolithic structure. While it may appear unorthodox, shifting the hardware/software boundary is a fairly common historical occurrence. Specialized hardware has replaced less efficient software emulation to better support virtual memory and memory protection (MMUs), machine virtualization, graphics and I/O, transactional memory, and context switching (simultaneous multithreading, partitioned register files). The multicore era has already created an environment where even basic systems now contain multiple processors. The manycore era will accelerate this trend, with each new system promising a doubling of the numbers of processors. Real-time system developers will not just want, but *need* efficient multithreading support for these systems. History suggests that an increased demand for low latency and low jitter thread-based operations can result in their evolution from relatively inefficient software routines, into more efficient hardware-based primitives.

II. Manycore Challenges for Real-Time Systems

Manycores will bring about performance improvements through parallelism instead of increased clock speeds and energy hungry speculative execution. As researchers conjecture that Moore's law will track processor densities in the manycore era, Amdahl's law provides a cautionary lesson that speedup does not simply track the numbers of parallel processors. Amdahl's law shows that speedup is confined to sections of code that can be parallelized. For thread-level parallelism, a program with 100 threads running on a system with 1,000 processors simply cannot achieve a 1,000x speedup. Perhaps Moore's law for manycores should be applied to the number of threads created; the more threads, the more potential for parallelism. Unfortunately there is no free lunch. As the number of threads increases, the operating system must expend more cycles to create, manage, and control the increased number of threads. Additionally, if a basic quantum of work expressed in a single thread is split into multiple, smaller threads to exploit more parallelism; then the ratio of cycles of fixed overhead costs for the operating system versus cycles available for application processing worsens. The growth in the number of threads can also introduce more jitter into operating systems with shared data structures. Operations such as releasing semaphores will cause the scheduler to be invoked to update thread contexts, even if no scheduling decision results.

Moore's law, when applied to threads, will clearly affect the efficiency and overhead incurred by the operating system. Operating system overhead can certainly be reduced by flattening the hierarchical software protocol stack. However latency and overhead is still incurred when sharing data structures within a monolithic kernel across multiple processors. Contended access to common structures in systems with 100's to 1,000's of processors will not be helped by flattening the protocol stack as this does not eliminate the potential for starvation and high contention-induced overhead.

A. Introducing Heterogeneity

Real-time embedded systems routinely integrate different cores to handle mixes of streaming real-time signal, image, and audio data along with general-purpose data processing requirements. Researchers within the manycore domain are following this approach calling for heterogeneous mixes of cores to better exploit the parallelism of next-generation applications.

At run-time, applications call operating system library code. This code is typically assembly language for a specific ISA. This poses no compatibility issues for homogeneous systems. However this does prevent, a processor with a *different* IS from directly invoking this library code. Inter-processor sy chronization implemented using ISA-specific atomic ope tions is particularly problematic, as atomic operations are not uniform across different processor families [1]. This mismatch is further compounded on modern SMP systems that rely on the underlying snoopy cache coherency protocols to guarantee atomicity of the operations. The Cell as well as Intel's EXOCHI [2], [3] resolve these differences using remote-procedure call (RPC) mechanisms. Slave processors request operating system services that run on a host processor. While flexible, the mechanisms used to implement RPCs usually involve interrupt/exception processing routines which introduce considerable overhead and jitter; 2 very undesirable traits.

A second popular approach to solving processor heterogeneity is to use interpretation techniques and virtual machines. Similar to RPC mechanisms, interpretation and virtualization techniques come at the cost of additional overhead. History has shown that these software emulation techniques become part of a more efficient hardware infrastructure when the demand for such services go from niche to mainstream. As an example, the demand for x86-based virtualization for servers has rapidly accelerated in recent years. However, the x86 was simply not designed with virtualization in mind, thus making it very difficult to virtualize in the *classical* sense [4]. The lack of proper virtualization abstractions at the hardware level forced developers to rely on a complex software run-time system; responsible for binary translation, code-caching, and just-in-time compilation [5].

Chip-makers soon realized that they could eliminate the complex run-time system by making small changes to the underlying hardware platform. Within a short period of time, chip-makers and virtualization experts developed a hardware/software co-designed solution that provided full x86 virtualization, and a vastly simpler run-time system. The solution was achieved by intelligently considering the hardware/software interface from the perspective of *both* hardware designers and software programmers. We began exploring the same HW/SW co-design approach for real time operating systems for embedded systems implemented within hybrid CPU/FPGA platforms.

III. Hthreads Microkernel

We began our investigation of hardware based microkernels by exploring how to bring modern programming model abstractions into the domain of real-time systems using hybrid CPU/FPGA components. Our goal was to bring custom accelerators created within the FPGA fabric under the control of a real-time Linux kernel. We rationalized that starting with an existing Linux kernel, and then modifying it as necessary, would be expedient and foster portability. Our first goal was to enable the FPGA-based co-processors to synchronize with software tasks running on the CPU using semaphores. Unfortunately, the CPU that was chosen, a stripped down PowerPC processor that was embedded in a Xilinx Platform FPGA, did not expose the pins necessary for cache

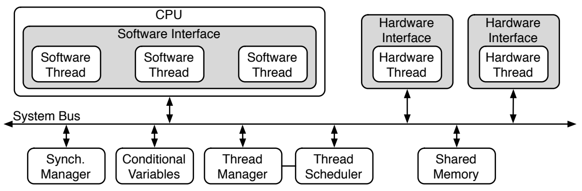


Fig. 1. Hthreads System Block Diagram

coherence bus protocols. This eliminated the ability to synchronize CPU-and FPGA-based tasks using load-linked/store-conditional (LL/SC) operations as the coherence information was not accessible within the FPGA.

To resolve this we replaced the use of atomic assembly instructions with a simple, hardware-based, mutex state machine. Although the implementation of the mutex hardware was quite simple, integration into Linux became slightly more complex. The Linux scheduler needed to be notified when threads became un-blocked. This required the use of an external interrupt in order to invoke the Linux scheduler from the mutex hardware. The use of interrupts produced unpleasant results as any change in state of a mutex resulted in the invocation of a CPU interrupt even when this state change did not result in a new scheduling decision. This introduced additional overhead, latency, and jitter to the application program running on the CPU. Additionally, synchronization events between dedicated hardware tasks would interrupt the CPU for bookkeeping reasons, even though these tasks did not utilize the CPU for execution purposes.

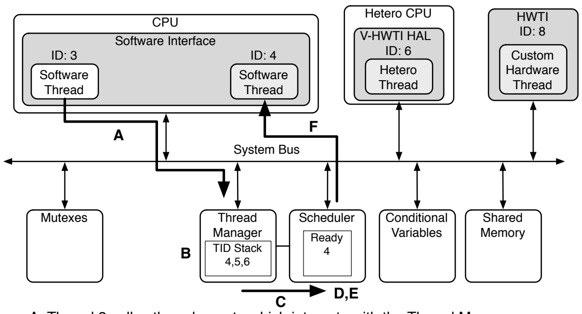
To reduce the latency and jitter, we then decided to migrate the operating system scheduler into hardware. It became obvious that attempting to break up the monolithic Linux kernel would require significant modifications to the kernel itself. As we continued down this path, it quickly became clear that we were essentially peeling away the layers of an onion. The presence of global data structures made it very difficult to discern which pieces of kernel state were *owned* solely by the scheduler. We then switched tactics, and set out to create a stand-alone microkernel composed of independent, low-overhead services, equally accessible by threads running on a CPU or in hardware. We focused on flattening the software protocol stack and the structure of the operating system, instead of only targeting the operating system internals. We adopted a POSIX threads (PThreads) compatible model [6], and created a HW/SW co-designed set of system services to support this model.

The resulting hthreads operating system consists of 4 major hardware cores that provide OS services: the Thread Manager, Scheduler, Synchronization Manager, and Condition Variables as shown in Figure 1 [7], [8]. This modular partitioning broke up the traditional monolithic kernel structure allowing us to separate concerns between different OS service cores. Each OS IP core fully encapsulates its internal data structures and serves as the sole interface to its internal data. This fosters explicit inter-service communications and eliminates shared data structures within the operating system itself. The basic control structure of each OS core is independent of the numbers and types of processor resources and acti threads in the system. This decoupling is advantageous each generation of manycore chips will provide performan increases through the addition of cores. The operating system must provide a framework that allows application programs to be seamlessly ported between generations as well as vendor platforms.

Each core has a simple memory-mapped interface that is accessed via traditional load/store instructions. This allows any processor that can master the bus to *directly* request services. This circumvents the need for slower remote-procedure call (RPC) mechanisms to provide a uniform set of efficient system services to all heterogeneous cores. Allowing each core to operate independently enables different processors to simultaneously request system services. This reduces both latency and jitter as simultaneous requests for different services do not compete for centralized services. The internal functions of each OS IP core have been parallelized, also providing low latency system calls with minimal jitter through efficient hardware implementations. More detailed analysis is provided in the following sections of each hthreads component. Timing results are provided that were obtained on two development platforms: a Xilinx Virtex-II Pro XC2VP30 FPGA, and a Xilinx Virtex-5 FXT 70 FPGA. All timing results were obtained using built in free running counters for cycle accurate measurements during run time.

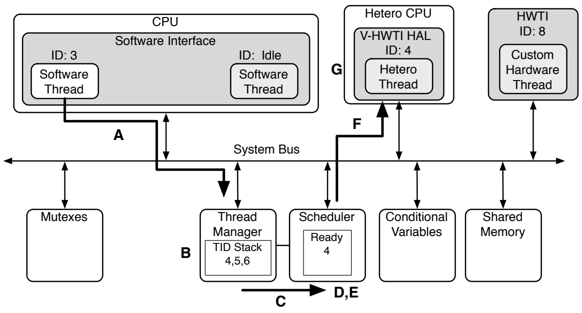
IV. Thread Management

A block diagram of the thread manager is shown in Figure 4. The thread manager core serves as a centralized repository of thread state, regardless of the location of the thread in the system. The state of a thread indicates if the thread is used, exited, running, suspended, or ready-to-run. Thread state also includes if the thread was created as a separate detached thread, or has a parent/child relationship with another active thread. This information is used in scheduling decisions for both the parent and child threads across heterogeneous processors. The thread manager serves as the interface for creation and allocation of thread IDs within the OS. The steps taken to create both a native and a heterogeneous thread within hthreads is shown in Figures 2 and 3 respectively. Important for scalability and processor heterogeneity, the thread manager separates the concern of thread state information from the physical numbers and types of processors. The state information is maintained and updated without needing to know where the thread is currently in the system. This includes threads that become ready-to-run after being unblocked from mutual exclusion primitives. Important for real-time systems, the thread manager reduces system jitter by fielding external interrupts. In a typical system, the CPU will receive interrupt requests from sources such as timers or external devices. The arrivals of these requests are outside of the control of the scheduler and interrupt the normal execution of applications. In hthreads, the thread manager fields requests, and the appropriate interrupt handler is scheduled as another thread. This puts the overhead of interrupt handling under scheduler control. Also, it makes the interrupt service routine running on the CPU very lightweight,



A: Thread 3 calls pthread_create which interacts with the Thread Manager.
B: Thread manager allocates a fresh thread identifier (TID 4).
C: Thread manager submits an ENQ request of TID 4 to the scheduler
D: Scheduler handles ENQ request by querying the attributes for TID 4.
E: Scheduler finds that ID 4 is a native thread, and adds it to the R2RQ.
F: If ID 4's priority is better than that of the currently running thread (ID 3):
* Then a preemption interrupt will be fired by the scheduler.
* Otherwise, TID 4 will remain in the R2RQ.

Fig. 2. Hthreads Native Thread Creation Sequence



A: Thread 3 calls pthread_create which interacts with the Thread Manager.
B: Thread manager allocates a fresh thread identifier (TID 4).
C: Thread manager submits an ENQ request of TID 4 to the scheduler
D: Scheduler handles ENQ request by querying the attributes for TID 4.
E: Scheduler finds that ID 4 is a heterogeneous thread, and prepares to signal
F: Scheduler sends a SIGNAL to TID 4's V-HWTI.
G: Heterogeneous thread begins execution

Fig. 3. Hthreads Heterogeneous Thread Creation Sequence

requiring only a register read to determine the next thread and a context switch.

A. *Performance Analysis*

Raw timing of the hardware execution times of thread management operations are provided in Table I. Several thread management operations directly result in scheduling operations, namely `add-thread`, `next-thread`, and `yield-thread`. The times shown for these operations acknowledge that additional latencies are incurred when the scheduler enqueues and dequeues thread_ids. The remaining operations deal solely with allocation, creation, and status of threads. All timings show the efficiency of a hardware resident microkernel. Table II also shows how end-to-end execution times can vary between heterogeneous processor families. These on-chip tests were measured at the application (user-thread) level on an Xilinx Virtex-5 FXT 70 FPGA clocked at 125 MHz with all data caches turned off. This variance is due to differences in the operating system kernel implementation on each processor family as well as slight

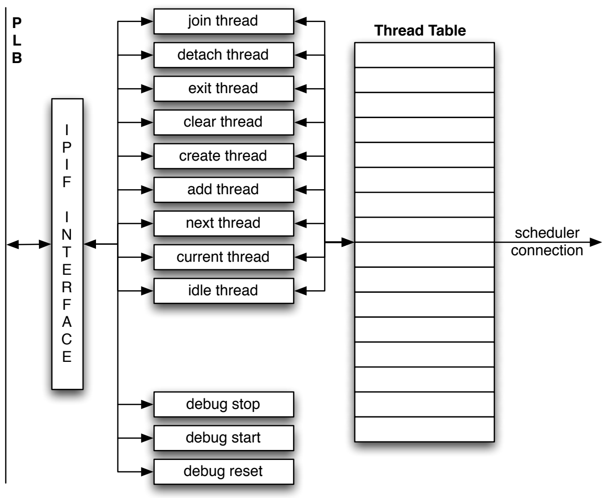


Fig. 4. Thread Manager Block Diagram

TABLE I
RAW (HW-ONLY) TIMING RESULTS OF THREAD MANAGEMENT OPERATIONS

Operation	Time (clock cycles)
Add_Thread	10 + ENQ
Clear_Thread	10
Create_Thread_Joinable	8
Create_Thread_Detached	8
Current_Thread	3
Detach_Thread	10
Exit_Thread	17
Join_Thread	10
Next_Thread	10 + DEQ
Yield_Thread	10 + ENQ + DEQ

differences in how the scheduler manages heterogeneous cores. Although execution times vary between processor families, system call overhead is still quite low within each family. We also show the performance degradations that can occur when calls, such as `create` and `join`, are implemented with RPC mechanisms. To illustrate this, we implemented a custom hardware component that requests OS services from a delegate software thread running on the PowerPC processor clocked at 300 MHz (approximately 2x faster than the 125 MHz tests). Even on a faster CPU, the RPC approach clearly suffers a significant performance degradation. This is due to a dependence on high overhead interrupts and additional communications requirements. Although not shown, these same RPC approaches can and do introduce unwanted and unpredictable jitter to a user's application.

V. SCHEDULER

The main function of the scheduler is to determine if a new scheduling decision is required for any processor in the system. A scheduling decision can result from both implicit and explicit events. Explicitly, a thread running on a processor may exit or yield. This frees up the processor, allowing another thread that is ready-to-run to execute. Implicitly, any thread in the system can release a resource that results in the unblocking of a different thread that may preempt a currently

System Call	Execution Time (from HW)	Execution Time (from PPC)	Execution Time (μBlaze)
`create`	160μs *	40.8μs	12.5μs
`join`	130μs *	65.7μs	13.9μs
`mutex_lock`	0.36μs	12.0μs	2.7μs
`mutex_unlock`	0.36μs	11.9μs	3.0μs

TABLE II
SYSTEM CALL PERFORMANCE. (*) INDICATES CALL IS IMPLEMENTED USING RPC MODEL.

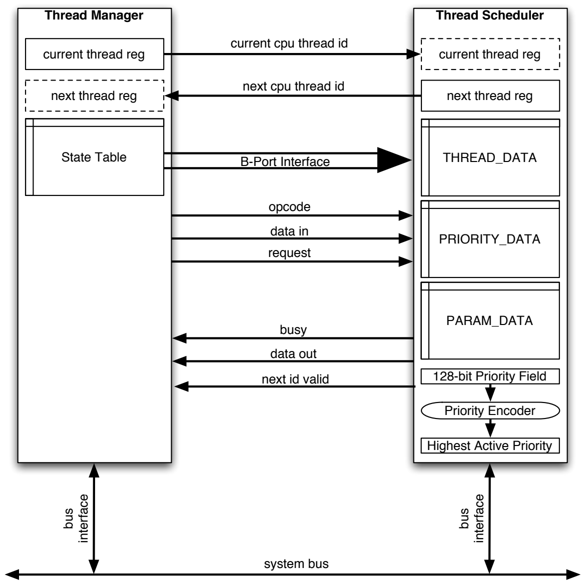


Fig. 5. Block Diagram of Scheduler Module

running thread. In general, a classic monolithic operating system scheduler is invoked for any potential scheduling decision regardless of the outcome. This can occur even when the priority of an unblocked thread will not result in a new scheduling decision. In this case, the CPU running the scheduler is interrupted so that the unblocked thread can be added back to the ready-to-run queue. This is a classic source of jitter in software-based systems as the scheduler must be invoked to *determine* if the release of the unblocked thread should trigger a new scheduling decision. An independent, hardware-based scheduler can evaluate the complete system state and determine if any new scheduling decisions are needed for all system resources without interrupting any particular application. Our hardware based scheduler makes all scheduling decisions a priori, in parallel with the application programs running on the CPUs. A processor is only interrupted when a thread entering the ready-to-run queue has a higher priority than both the thread currently running on that processor, as well as any other threads within the ready-to-run queue.

The scheduler is implemented as a single finite-state machine along with a set of three Block RAMs and a 128-bit priority encoder as shown in Figure 5.

The 128-bit priority encoder is used to calculate the highest priority level with active threads in the system at any given

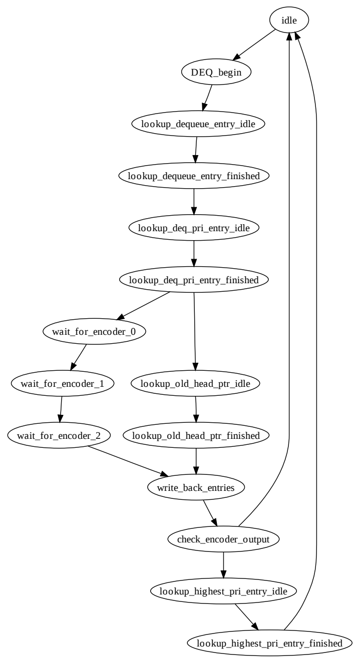

idle
DEQ_begin
lookup_dequeue_entry_idle
lookup_dequeue_entry_finished
lookup_deq_pri_entry_idle
lookup_deq_pri_entry_finished
wait_for_encoder_0
wait_for_encoder_1
lookup_old_head_ptr_idle
wait_for_encoder_2
lookup_old_head_ptr_finished
write_back_entries
check_encoder_output
lookup_highest_pri_entry_idle
lookup_highest_pri_entry_finished

Fig. 6. Scheduler Dequeue Operation

time. Its input is a 128-bit register in which each bit signifies whether or not active threads exist at a given priority level. The priority encoder has been implemented as a small FSM that multiplexes each 32-bit section of the 128-bit input field onto a 32-bit priority encoder. The output of the 128-bit priority encoder is a 7-bit value that represents the highest priority level with active threads in the system. In hthreads the best priority level is 0, while the worst is 127. The priority encoder always generates a new output in four clock-cycles.

Together, the priority encoder and the partitioned ready-to-run queue structure allow for all of the scheduler's operations to complete in constant time. The priority encoder allows the scheduler to find the highest priority thread without having to traverse the ready-to-run queue. The partitioned ready-to-run queue structure is logically a set of 128 FIFOs, one FIFO for each priority level, on which constant time enqueue and dequeue operations occur. It is important to note that the scheduler will recalculate the next scheduling decision at any time the ready-to-run queue is changed; i.e., when threads are added or removed from the queue, or when scheduling parameters change for individual threads.

The dequeue operation of the scheduler is used to consume the current scheduling decision by removing the thread that is scheduled to run next from the ready-to-run queue, followed by the calculation of the next scheduling decision. The next scheduling decision can be calculated by first examining the output of the priority encoder to find the highest active priority level in the system. Once this priority level is found, the next scheduling decision, or next thread to run, is the head of this priority level's queue. The control flow graph of the dequeue operation is shown in Figure 6.

The enqueue operation of the scheduler is used to add a

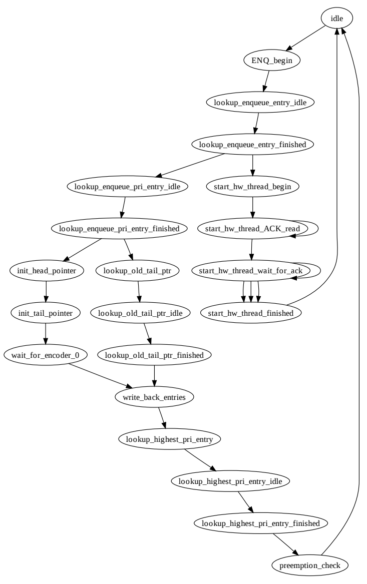


Fig. 7. Scheduler Enqueue Operation

thread to the ready-to-run queue, followed by a calculation of the next scheduling decision. The newly-scheduled thread is added to the end of its priority level's queue, then the priority encoder recalculates the highest active priority level in the system. The next scheduling decision is chosen and its priority is compared to that of the currently-running thread (or threads, in SMP mode) in order to determine if preemption should occur. If it is determined that the thread to be scheduled next is of better priority than the currently-running thread, then the scheduler will assert the preemption interrupt. Otherwise, the system will continue to run uninterrupted until the next scheduling event occurs, such as a blocking system call or another change in the ready-to-run queue. The control flow graph of the enqueue operation is shown in Figure 7.

The adjustment of a thread's scheduling parameter is the composition of the enqueue and dequeue operations. If a thread is not currently queued, adjusting the thread's scheduling parameter is merely an adjustment of a single thread's state. However, if the thread is currently queued this operation could result in the change of that thread's location in the queue; i.e. a different priority level. Essentially, the scheduler first performs a dequeue of the selected thread, followed by an enqueue of that thread with its updated scheduling parameter.

A. Performance Analysis

The raw (HW-only) performance of the scheduler's operations are shown in Table III. One can see that all of the scheduling operations execute in 500 ns (50 clock cycles) or less. These execution times are extremely fast, especially when considering how many cycles many of these operations would take within software.

It is also important to note that the scheduler has been implemented as an FSM in which all internal data structures have constant access time, and as such, has completely deterministic behavior. The predictable behavior of this FSM paired

TABLE III
RAW (HW-ONLY) TIMING RESULTS OF SCHEDULING OPERATIONS

Operation	**Time (clock cycles)**
Enqueue(SWthread)	28
Enqueue(HWthread)	20 + (1 Bus Transaction)
Dequeue	24
Get_Entry	10
Is_Queued	10
Is_Empty	10
Set_Idle_Thread	10
Get_Sched_Param	10
Check_Sched_Param	10
Set_Sched_Param(NotQueued)	10
Set_Sched_Param(Queued)	50

with the constant-time execution behavior of the partitioned ready-to-run queue makes for a scheduling subsystem whose performance does not vary with the number of active (queued) threads. This in turn makes for jitter-free scheduling operations at the base hardware-level, and thus more predictable and precise scheduling of threads within the OS.

The only exception, which can be seen in Table III is the amount of time it takes to perform the master bus transaction when a HW thread is being enqueued. The jitter in this operation is inherent to any bus architecture, as bus arbitration times vary depending on the instantaneous contention for the shared bus.

Integrated performance testing of the scheduler focuses on two major metrics: end-to-end scheduling delay, and raw interrupt delay. End-to-end scheduling delay is defined as the time delay between the firing of a timer interrupt to when the context switch to a new thread is about to complete (old context saved, new context is about to be switched to). The end-to-end scheduling delay is measured using a dedicated hardware counter that begins counting immediately after a timer interrupt fires, and stops counting as soon as measures this delay in the context switch is complete. This measurement is collected in a non-invasive way, as the hardware counter directly monitors the interrupt lines attached to the CPU; allowing for clock-cycle accurate measurements to be made without bias. Raw interrupt delay is defined as the time delay from when an interrupt signal fires to when the CPU actually enters its ISR. The raw interrupt delay is also measured using a dedicated hardware counter that begins timing immediately after a timer interrupt goes off and would stop timing when the CPU sent a signal to the hardware timer from its ISR (a single memory-mapped read).

On-board testing of the raw interrupt delay of this system with 250 active threads is shown in Figure 9. After being synthesized and fully integrated into the hthreads system, the scheduler has an average end-to-end scheduling delay of 1.9 μs with 1.4 μs of jitter with 250 active threads as shown in Figure 8. Where the jitter is defined as being the difference between the maximum and mean delays, making the worst-case end-to-end scheduling delay 3.4 μs. The results of both the worst-case end-to-end scheduling delay and jitter are quite low when compared to the 1.3 ms worst-case scheduling delay and 200 ms scheduling jitter of Linux [9], the 40 μs scheduling delay of Malardalen University's RTU [10]. Even RTLinux, a

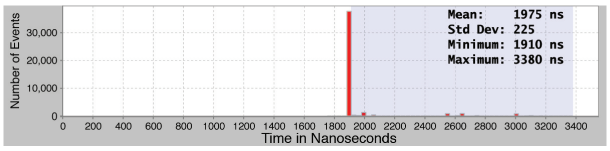


Fig. 8. Histogram of Integrated End-To-End Scheduling Delay, (250 Active SW Threads, 100 MHz)

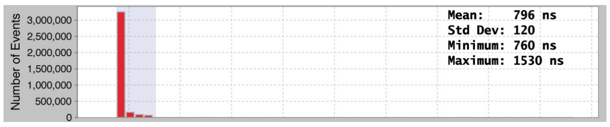


Fig. 9. Histogram of Raw Interrupt Delay, (250 Active SW Threads, 100 MHz)

commercial real-time OS, has an order of magnitude higher scheduling jitter: 12 μs of scheduling jitter for a 1.2 GHz desktop CPU, and 32 μs of jitter for a 200 MHz embedded CPU [11], [12]. Further tests have shown that the scheduling jitter within hthreads is primarily caused by cache misses during context switching.

During a context switch, the CPU does not interact with the scheduler module, which gives the scheduler module the perfect opportunity to calculate the next scheduling decision. This scheduling decision is calculated in parallel with CPU execution, thus eliminating much of the processing delays normally incurred by calculating a new scheduling decision. Also, the scheduling decision being calculated is for the *next* scheduling event so that when a timer interrupt goes off, the next thread to run has already been calculated. This pre-calculation of the scheduling decision allows the system to react very quickly to scheduling events because when a scheduling event occurs, the OS simply reads the next thread to run out of a register and performs a context switch, and then during this context switch, the register is refreshed with the thread that should run at the next scheduling event by the hardware scheduler module. The overhead of making a scheduling decision is hidden because it occurs during a context switch and it is able to complete before a context switch completes.

VI. Synchronization

Modern processors use ISA-specific atomic instructions to perform mutual exclusion. Originally, atomic instructions such as test and set, required locking the system bus to ensure operation atomicity. This resulted in performance degradations for other processors attempting to access global memory. Bus locking can be circumvented by using a two-phase atomic operation such as the load-linked/store conditional (LL/SC, or more specifically lwarx and stwcx) instructions found with IBM's PowerPC architecture. These instructions rely on the snoopy cache protocol to maintain atomicity between processors. The reliance on processor-specific atomic operations and the snoopy cache protocol make it difficult to perform atomic operations between different processor types [1], [13].

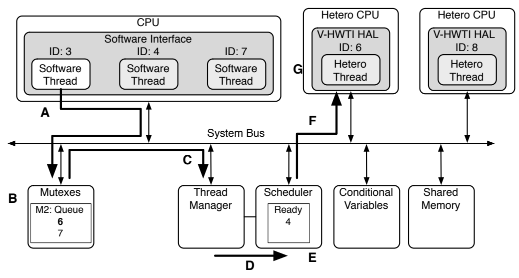
Our hardware synchronization core is designed to allow any type of processor to synchronize in a heterogeneous syste using a single, traditional load instruction. The synchronizati manager hosts a set of mutexes, in which each mutex is associated with an address offset from the base address of the IP core. The offsets of each mutex are separated by a stride indicated by the number of bits used to represent thread identifiers in hthreads. This allows an entire mutex lock/unlock command to be encoded in a single load instruction. This structure allows a processor to use a single load instruction to pass parameters to the synchronization manager in the address field, and to receive the return value of the call in the loaded value. The command contains the thread's identifier, the mutex number, as well as the type of command. The loaded value, or return value of the command, tells the requester the result of the mutex lock/unlock operation. For a mutex lock command, this return value states whether the lock is now owned by the requester or not. If the thread did not become the owner, then the synchronization manager adds this thread to its internal blocked queue. Upon unlock operations, the synchronization manager sends the identifier of the new owner thread to the thread manager to become runnable again.

This simple protocol allows any processor that can master the bus to engage in synchronization with any other processor, regardless of ISA differences, thus supporting processor heterogeneity. Reducing mutex operations down to a single load instruction provides very fast synchronization primitives without the need for a snoopy cache coherence protocol. A mutex unlock operation is illustrated in Figure 10. This figure shows the processing steps performed to release a mutex, make a scheduling decision, and resume the execution of a thread. In a traditional operating system, steps A through E are performed completely in software on the CPU. These steps would require a context switch from the application thread to the system services, and must be performed *before* the scheduler considers if a new scheduling decision is required based on the queuing of the unblocked thread. In our hthreads system, steps B through G are performed in hardware, allowing the CPU to continue executing the application thread. For systems with heterogeneous processors, migrating this processing off a single CPU is advantageous, as significant overhead and jitter are introduced when a CPU performs speculative processing for threads being unblocked on other processors. Another example of mutex operations is shown in Figure 11. This example demonstrates the OS-level processing that occurs when a native software thread attempts to lock a mutex that is already owned by a heterogeneous thread. The calling thread ends up blocked within the synchronization manager's wait queue, and will become unblocked following the steps in Figure 10 when the heterogeneous thread releases the mutex.

The synchronization manager provides support for the three standard PThreads lock types: fast, error checking, and recursive. Additionally, the synchronization managers maintains mutex waiting lists for blocked threads waiting for access to a mutex. The basic operations for the synchronization manager are listed in Table IV. Figure 12 shows the top-level design of the synchronziation manager. The design consists of six finite state machines, and two Block RAMs.

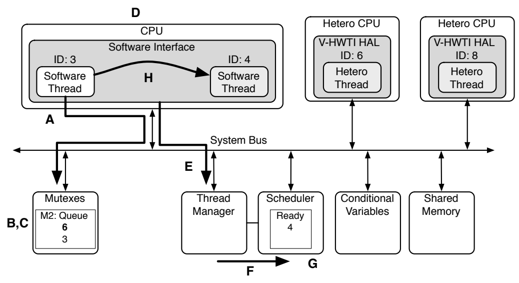
TABLE IV
SM OPERATIONS AVAILABLE THROUGH BUS INTERFACE

Operation	Type	Description
Lock	Read-only, depth = MxT	Attempts to lock a mutex for a given thread.
Try-lock	Read-only, depth = MxT	Attempts to try-lock a mutex for a given thread.
Unlock	Read-only, depth = M	Unlocks a mutex.
Owner	Read-only, depth = M	Returns the current owner's TID of a given mutex.
Kind	Read/Write, depth = M	Returns or updates the kind of a given mutex. Where kind is one of the following: FAST, RECURSIVE, or ERROR.
Count	Read-only, depth = M	Returns the current lock count of a given mutex. Only valid for RECURSIVE mutexes.



A: Thread 3 unlocks mutex M2 by calling hthread_mutex_unlock(M2), which sends signal to Mutex Manager.
B: Mutex Manager inspects M2's queue and decides ID 6 will own mutex next.
C: Mutex Manager sends add_thread(6) to Thread Manager.
D: Thread Manager gives ID 6 to Scheduler to add to Ready to Run queue.
E: Scheduler finds that ID 6 is a hetero. thread, does not add ID 6 to queue.
F: Scheduler instead sends *SIGNAL* command to ID 6's V-HWTI.
G: Hetero thread 6 resumes execution, now owner of mutex M2.

Fig. 10. Hthreads Mutex Unlock Sequence (Heterogeneous)



A: Thread 3 locks mutex M2 by calling hthread_mutex_unlock(M2), which sends signal to Mutex Manager.
B: Mutex Manager inspects M2's queue, sees that ID 6 owns the mutex.
C: Mutex Manager blocks TID 3 by placing it in M2's blocked queue, and returns blocked status to the caller.
D: Software interface receives blocked status, must now context switch.
E: Software interface asks for the next thread from Thread Manager.
F: Thread Manager requests a DEQ operation from the Scheduler.
G: Scheduler tells Thread Manager that TID 4 should run next.
H: Software Interface receives the next thread, context switches to TID 4.

Fig. 11. Hthreads Mutex Lock Sequence (Heterogeneous)

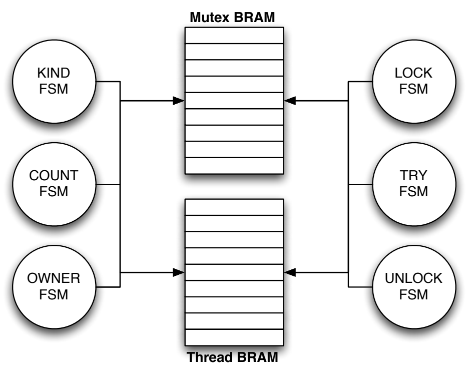


Fig. 12. Synchronization Manager

The lock command is used to request ownership of a mutex. It operates using a 3-state FSM pattern: idle, read, and modify. In the read state, the FSM will examine the lock count of the mutex; if it is zero, the FSM will grant the mutex to the requesting thread. If the lock count is not zero then the thread is placed on the waiting list for the mutex and the thread is blocked. The trylock command is a simple variation on the lock command which does not queue or block the requesting thread when the mutex cannot be granted.

The unlock command uses a slightly more complex 5-state FSM pattern: idle, read mutex, modify mutex, read next, and send next. This command examines the lock count in the read state and performance one of three possibilities:

1) If the value is zero then, then an error will be returned to the requesting thread.
2) If the value is one, then the mutex is granted to the next thread on the mutex's waiting queue.
3) If the value is greater than one, the lock count will be decremented and the command will finish successfully.

If the mutex must be granted to the next thread in the waiting queue, the unlock command must perform the read next and send next states. These states simply remove the next thread from the waiting queue and place it onto the send queue. The send queue is processed in FIFO order, and simply performs an add thread command to the scheduler. This will wake the next thread from its blocked state.

The remaining three state machines, get owner, get/set kind, and get count, are all simple state machines which either access or modify mutex status. These three state machines access the requested information in the read state and then return it to the requesting thread. If a set operation is occurring then the modify state will update the mutex status to the new value.

A. Performance Analysis

The performance for the SM is shown in Table V. This table shows that the SM completes its task in five clock cycles (50 ns) or less. This level of performance is achievable with standard synchronization primitives only when processors implement caches with complex memory coherency protocols, such as snoopy caches.

Like the scheduler, the SM has been implemented using FSMs with internal data structures accessed through BRAM. This gives the SM its high performance and deterministic behavior. Due to this and the determinism of the scheduler, the

TABLE V
HW TIMING OF SYNCHRONIZATION OPERATIONS

Operation	**Time (cycles)**
Lock Mutex	4
Unlock Mutex	5
Try Lock Mutex	3
Get Mutex Owner	3
Get Mutex Count	3
Get/Set Mutex Kind	3

TABLE VI
INTEGRATED TIMING OF SYNCHRONIZATION OPERATIONS

Operation	**Avg Time (ns)**	**Std. Dev. (ns)**
Lock Mutex	1524.54	52.19
Unlock Mutex	1097.63	27.33

hthreads operating system has very low jitter, and is suitable for use in real-time systems.

The only variable execution time inside the SM is caused by a bus transaction, which must occur when a thread is woken up when it is given ownership of a mutex. This occurs during an unlock operation if there are any threads waiting for the lock. The effects of this variable execution time are not visible in the execution time of the unlock command, because bus transactions are placed on a FIFO queue which is processed in parallel with the SM commands. The design of the unlock FSM guarantees that there will always be space in the queue for the next bus transaction. Thus, the unlock command needs only to queue the transaction in order to finish its operation and this can be completed in a deterministic single clock cycle.

Integrated performance testing of the synchronization manager measures how long it takes the software APIs in the operating system to complete a synchronization operation on behalf of a software thread. These numbers time how long the system call takes. Thus, they do not measure any additional overhead which would be required to make the user mode to kernel mode switch.

Table VI shows the integrated timing results for the synchronization operations. Only the lock and unlock operations are shown, as they are the most commonly executed synchronization operations. Other synchronization operations are substantially similar. The results show that the lock and unlock commands execute in approximately 1.5 μs and 1.1 μs respectively. These averages have a standard deviation of approximately 0.05 μs and 0.02 μs respectively. The hardware overhead for these commands is 0.04 μs and 0.05 μs respectively, which means that the majority of the software side execution time is due to overhead from the bus transactions and software processing.

VII. CONCLUSION

Manycore architectures are bringing a welcome convergence of hardware platform capabilities with the requirements of real-time and embedded systems. The performance capabilities of these platforms can bring significant economic benefits by reducing the need to create application specific custom hardware. To realize these benefits new approaches to operating systems within both the general purpose and real-time systems domains will likely evolve. Manycores can only bri about higher performance if parallelism is increased throu the creation of more threads. Efficient implementations of (services will be *fundamental* in the success of these new systems. If services are inefficient, contention will result in severe latencies for threading operations challenging the very success of the manycore approach. Hardware microkernels, such as hthreads, show that hardware-resident system services have the potential to provide services with latencies that are minimal and constant, ideal for systems with scalable numbers of processors and threads. The minimal latency of the system services will allow programmers to create finer-grained threads to expose additional parallelism and still meet strict latency requirements. The distributed nature of the microkernel breaks up the bottleneck of shared data structures and enables parallelism within the kernel itself. The access and invocation mechanisms, including fundamental atomic operations, provide a system framework that resolves many of the uniformity and scalability issues that arise from processor heterogeneity.

The hthreads microkernel discussed in this paper is a stable set of system services that has been used for exploring programming models for MPSoC systems comprised of mixes of both general-purpose heterogeneous CPUs, and CPUs paired with custom accelerators. All timing metrics reported in this paper are actual on-chip, run-time measurements obtained using free running hardware counters within the system. The timing numbers are generated from tests running on Xilinx Virtex-II Pro and Virtex-5 FXT platforms. The Xilinx Virtex-II Pro XC2VP30 FPGA systems clock the PowerPC clocked at 300 MHz, while the FPGA components are clocked at 100 MHz (10 ns cycle time). Whereas the Virtex-5 FXT platform runs both the PowerPC and FPGA components at a uniform 125 MHz clock rate. Our comparisons do not adjust for the disparaties in clock frequencies between the slower running FPGA and faster desktop machines. Even at these slower clock rates the performance of the microkernel components were better than those reported on the faster desktop machines. The hthreads microkernel required only 18% of the logic gates and 6% of the embedded BRAM of the Xilinx Virtex-II Pro XC2VP30 FPGA. This represents a modest requirement for embedded systems implemented on modern platform FPGAs, or for inclusion in future manycore chips. Importanlty, the performance and jitter envelopes of the hthreads microkernel are tight enough for use in real-time, embedded systems and the structure of the microkernel supports large numbers of heterogeneous processors and threads with no additional performance penalties.

Acknowledgment

The work in this article is partially sponsored by National Science Foundation EHS contract CCR-0311599.

REFERENCES

[1] B. Senouci, A. Kouadri M, F. Rousseau, and F. Petrot, "Multi-CPU/FPGA Platform Based Heterogeneous Multiprocessor Prototyping: New Challenges for Embedded Software Designers," *The 19th IEEE/IFIP International Symposium on Rapid System Prototyping, 2008. RSP '08*, pp. 41–47, June 2008.

[2] C. H. Crawford, P. Henning, M. Kistler, and C. Wright, “Accelerating Computing with the Cell Broadband Engine Processor,” in *CF ’08: Proceedings of the 2008 Conference on Computing Frontiers*. New York, NY, USA: ACM, 2008, pp. 3–12.

[3] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang, “EXOCHI: Architecture and Programming Environment for a Heterogeneous Multi-Core Multithreaded System,” *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 156–166, 2007.

[4] G. J. Popek and R. P. Goldberg, “Formal Requirements for Virtualizable Third Generation Architectures,” *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[5] K. Adams and O. Agesen, “A Comparison of Software and Hardware Techniques for x86 Virtualization,” in *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 2006, pp. 2–13.

[6] D. R. Butenhof, *Programming with POSIX threads*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997, last accessed June 8, 2009.

[7] J. Agron, W. Peck, E. Anderson, D. Andrews, E. Komp, R. Sass, F. Baijot, and J. Stevens, “Run-Time Services for Hybrid CPU/FPGA Systems On Chip,” in *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS)*, December 2006.

[8] D. Andrews, R. Sass, E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, and E. Komp, “Achieving Programming Model Abstractions For Reconfigurable Computing,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 1, pp. 34–44, January 2008.

[9] C. Williams, “Linux Scheduler Latency,” Red Hat Inc. Technical Paper. [Online]. Available: http://www.linuxdevices.com/files/article027/rh-rtpaper.pdf

[10] T. Samuelsson, M. Akerholm, P. Nygren, J. Starner, and L. Lindh, “A Comparison of Multiprocessor RTOS Implemented in Hardware and Software,” in *Proceedings of the 15th Euromicro Workshop on Real-Time Systems*, 2003. [Online]. Available: http://www.mrtc.mdh.se/publications/0662.pdf

[11] V. Yodaiken, “The RTLinux Manifesto,” in *Proceedings of The 5th Linux Expo, Raleigh, NC*, 1999. [Online]. Available: citeseer.ist.psu.edu/yodaiken99rtlinux.html

[12] V. Yodaiken, C. Dougan, and M. Barabanov, “RTLinux/RTCore Dual-Kernel Real-Time Operating System,” FSMLabs Inc. Technical Paper. [Online]. Available: http://www.yodaiken.com/papers/rtlpro.pdf

[13] T. Suh, D. Blough, and H.-H. Lee, “Supporting Cache Coherence in Heterogeneous Multiprocessor Systems,” *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, vol. 2, pp. 1150–1155 Vol.2, February 2004.