Proceedings of

OSPERT 2013

9th annual workshop on
Operating Systems Platforms for
Embedded Real-Time Applications

July 9th, 2013 in Paris, France

in conjunction with the
25th Euromicro Conference on Real-Time Systems
Paris, July 10-12, 2013

Editors:
Andrea Bastoni
Shinpei Kato

Contents

Message from the Chairs

In this 9th workshop on Operating Systems Platforms for Embedded Real-Time Applications we aimed at continuing the discussion-based focus and the interaction between Academia and Industry that traditionally characterize OSPERT. The workshop will be opened by Frederic Weisbecker's keynote who will discuss the challenges of virtual CPU-time accounting, dynamic ticks, and enabling the full tickless mode in the Linux Kernel. To increase the interaction possibilities, we have looked at providing ample discussion time between the paper presentations.

Following the positive trend of last year's OSPERT, the workshop will feature the presentation of traditional technical papers, forward-looking papers—with a strong focus on innovative ideas, open problems, and implementation issues—as well as a new category of papers that targets discussions on existing OS-related open source projects. OSPERT this year accepted 6 of 7 peer reviewed papers and will further include two invited papers on the comparison of scheduling latencies in Linux, PREEMPT_RT, and LITMUS^{RT} (a Linux-based framework to experiment scheduling and locking protocols), and on the challenges to uncover—through reverse engineering—and exploit the complex power-management and RTOS embedded features currently included in modern GPGPUs. We would like to thank all the authors for their hard work: given the presented topics and the quality of the submissions, we expect a lively workshop.

OSPERT 2013 would not have been possible without the support of many people. The first thanks goes to Gerhard Fohler and to the ECRTS chairs for making it possible to have this venue to discuss operating system and implementation-oriented real-time topics. We would also like to thank the program committee, for their effort in carefully selecting an interesting program and providing useful feedback to the authors.

Last, but not least, we would like to thank you, the audience, for actively contributing—through your stimulating questions and lively interest— to define and improve OSPERT. We hope you will enjoy this day.

The Workshop Chairs,
Andrea Bastoni
Shinpei Kato

Program Committee

Neil Audsley, *University of York*
Björn B. Brandenburg, *Max Planck Institute for Software Systems*
Robert Kaiser, *Hochschule RheinMain University of Applied Sciences*
Wolfgang Mauerer, *Siemens*
Paul McKenney, *IBM*
Thomas Nolte, *Malardaren University*
Gabriel Parmer, *George Washington University*
Rodolfo Pelizzoni, *University of Waterloo*
Steven Rostedt, *Red Hat*
Richard West, *Boston University*

Keynote Talk

Present and Future of Linux *dynticks*

Frederic Weisbecker
Red Hat

Linux has been able to stop the tick for a few years now. This feature is known as dynticks*. Although it was limited to idle CPUs, this was a great step towards enabling power-saving solutions. Now, the 3.10 Linux Kernel can extend this dynamic tick behavior not only to idle CPUs, but also to busy CPUs. This time, power-saving is not the only target, but rather latency and performance. For now the benefit is mostly to be expected in extreme real-time and High Performance Computing, but it may extend to more general purposes in the long run. This talk aims at diving into dynticks internals and speculates about its future improvements.*

Biography:
Frederic Weisbecker is a Linux Kernel developer working for Red Hat. His involvement and role in the Linux community has evolved over time: he has been working on tracing with *ftrace* and *perf events* subsystems, on timers and dynticks-mode, and he helped to remove the big kernel lock. In 2010, he took up the challenge of disabling the tick interrupt on non-idle processors. Eventually, after many changes and helps from other Linux Kernel developers, his work has been merged into the 3.10 kernel.

Program

	Tuesday, July 9th 2012
8:30-9:30	Registration
9:30-11:00	Keynote Talk: *Present and Future of Linux* dynticks *Frederic Weisbecker*
11:00-11:30	Coffee Break
11:30-13:00	Session 1: Implementation and Performance evaluation Investigation and Improvement on the Impact of TLB misses in Real-Time Systems *Takuya Ishikawa, Toshikazu Kato, Shinya Honda, Hiroaki Takada* Implementation of the Multi-Level Adaptive Hierarchical Scheduling Framework *Nima Moghaddami Khalilzad, Moris Behnam, Thomas Nolte* A Comparison of Scheduling Latency in Linux, PREEMPT RT, and LITMUS^{RT} *Felipe Cerqueria, Bjöern Brandenburg*
13:00-14:30	Lunch
14:30-16:00	Session 2: Power-Management and Open-Source Projects Towards power-efficient mixed-critical systems *Florian Broekaert, Sergey Tverdyshev, Laurent San, Agnes Fritsch* Reverse engineering power management on NVIDIA GPUs - Anatomy of an autonomic-ready system *Martin Peres* The state of Composite *Jiguo Song, Qi Wang, Gabriel Parmer*
16:00-16:30	Coffee Break
16:30-17:30	Session 3: Resource Sharing and Locking Priority Inheritance on Condition Variables *Tommaso Cucinotta* Deterministic Fast User Space Synchronization *Alexander Züpke*
17:30-18:00	Discussion and Closing Thoughts

	Wednesday, July 10th - Friday, July 12th 2013
	ECRTS main conference.

Investigation and Improvement on the Impact of TLB misses in Real-Time Systems

Takuya Ishikawa, Toshikazu Kato, Shinya Honda and Hiroaki Takada
Nagoya University, Nagoya, Japan

***Abstract*—Memory protection for real-time systems is needed to ensure safety of the systems, in recent years. The memory management unit used for memory protection uses the translation lookaside buffer (TLB) which is a caching mechanism. When using the TLB in real-time systems, the worst-case execution time (WCET) is estimated pessimistically, because it is difficult to predict the occurrence of TLB misses. In this paper, first, the impact of TLB misses on the WCET is evaluated. Secondly, methods to control the occurrence of TLB misses by using the TLB locking mechanism are proposed. The result of evaluation shows the effectiveness of the proposed methods.**

I. Introduction

Memory protection [1] for real-time systems is needed to ensure safety of the systems, in recent years [2]. A memory management unit (MMU) is hardware that manages virtual memory systems for memory protection, and it is used in most general-purpose systems and high-end embedded systems, such as avionic computer systems. An MMU performs address translation with a page table, which is a table for mapping a virtual address to a physical address. Furthermore, each entry of a page table contains a field for access permission. An MMU can detect run-time illegal memory access with an address translation. A translation lookaside buffer (TLB), where a page table entry is cached, is usually used to reduce the time of address translation by an MMU. In the case that a TLB does not contain a required entry, a TLB miss exception occurs, then a required entry is obtained from a page table in memory (page table walk) and replaced with another entry that is contained in a TLB. A page table walk requires a considerable amount of execution time to access memory several times.

Meanwhile, real-time analysis requires worst-case execution times (WCETs) of each task in a system [3] [4]. However, most real-time analysis uses WCETs that are pessimistically estimated, because it is difficult to estimate the exact WCET [5]. In the case that an MMU (a TLB) is used in real-time systems, WCETs have the potential to be estimated even more pessimistically. This is because it is difficult to predict the occurrence of TLB misses. Nevertheless the impact of TLB misses on WCETs and methods to improve predictability of WCETs are not studied.

In this paper, the impact of TLB misses on WCETs and methods to improve predictability of WCETs are studied. First, execution time distribution of a task in a micro-benchmark of real-time systems with an MMU is evaluated. The results of evaluation demonstrate that TLB misses have a considerable impact on WCETs. Furthermore, methods to reduce the impact of TLB misses are proposed. The proposed methods control the occurrence of TLB misses by using TLB locking, which makes a specified TLB entry not to be replaced (expired) and inhibit the occurrence of TLB misses related to that entry. A WCET of a task in the micro-benchmark become able to be estimated, when the proposed methods are applied to the micro-benchmark. In addition, response time of the micro-benchmark with the proposed methods is evaluated.

The rest of this paper is organized as follows. Section II describes evaluation environment and a micro-benchmark. Section III evaluates an impact of TLB misses on WCETs. Section IV presents methods to reduce the impact of TLB misses by using TLB locking. Section V evaluates the proposed methods. Section VI concludes this paper.

II. Evaluation Environment

In this paper, an SH7750R processor [6], whose core speed is 235MHz, with the MMU is used as the evaluation target processor. A TLB of an SH7750R processor has 64 entries for both data and instruction accesses, and each page size is 4096 bytes. A TLB miss exception of this processor is handled by a real-time operating system (RTOS), that is to say, handled by software. In this paper, TOPPERS/HRP2 kernel (High Reliable system Profile version 2: HRP2) [7], which is an RTOS with memory protection, is used. The specifications of HRP2 are described in the TOPPERS new generation kernel specification [8], which is based on the protection extension of μITRON4.0 specification [9]. In addition, each TLB entry is associated with an address space identifier (ASID) which identifies a currently executing task. HRP2 changes an ASID of the processor with the context switch instead of flushing the TLB. Besides, cache is disabled to make it easier to evaluate an impact of TLB misses on WCETs in this evaluation.

A micro-benchmark used in this evaluation includes automotive application software and multimedia application software. This benchmark is with consideration of automotive software integration [10], for example, that both automotive control systems and automotive navigation systems are in the same electronic control unit. The automotive application software and the multimedia application software are part of AutoBench and DENBench in EEMBC (The Embedded Microprocessor Benchmark Consortium) benchmark software [11]. Each benchmark software runs as one task of HRP2. Table I shows benchmark software names that are used in the micro-benchmark, a task ID that each benchmark is assigned to, the priority of each task, and memory consumption size of each benchmark. Here, A task with smaller value of the priority has higher priority. Each RT_TASK executes benchmark software of AutoBench, and NR_TASK executes benchmark software of DENBench. Every RT_TASK is a real-time task, and NR_TASK is a non real-time task. In addition, Figure 1 shows execution flow of the micro-benchmark.

TABLE I. BENCHMARKS USED FOR THE EVALUATION

task ID	Benchmark Software	Priority	Size (Byte)
RT_TASK1	ttsprk01	1	58,164
RT_TASK2	a2time01	2	6,564
RT_TASK3	rspeed01	3	4,088
RT_TASK4	puwmod01	4	13,392
RT_TASK5	canrdr01	5	9,360
NR_TASK	djpegv2data4	6	535,332

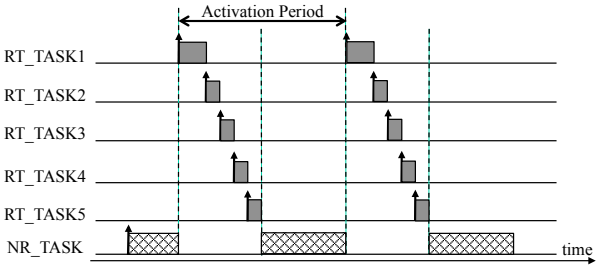


Fig. 1. Execution Flow of the Micro-Benchmark

Firstly, NR_TASK is activated, and RT_TASK1 is activated periodically at a constant period. After RT_TASK1 executes the own benchmark, RT_TASK1 activates RT_TASK2 and ends. Subsequently, RT_TASKn (n = 2,3,4) executes the own benchmark, activates RT_TASKn+1, and ends in the same manner as above. NR_TASK is not running while RT_TASK is running, because every RT_TASK is assigned higher priority than NR_TASK. Now, NR_TASK does not end until every RT_TASK is activated 10,000 times. The average execution time and the maximum execution time of each RT_TASK are 3 milliseconds and 5 milliseconds, respectively. The activation period of RT_TASK1 is 50 milliseconds.

III. IMPACT OF TLB MISSES ON WCETS

In this paper, an impact of TLB misses on WCETs is evaluated with the metric that is calculated by the following expression:

$$\frac{E_H}{E_T - E_H} \tag{1}$$

Here, E_H and E_T indicate the execution time of TLB miss exception handlers and an evaluated task. This evaluation metric is a ratio of the exception time of TLB miss exception handlers to the best case execution time of a evaluated task, which is execution time of that task in the case that TLB misses never occurs.

The evaluated task is RT_TASK1. The execution time of RT_TASK1 and the number of TLB misses that occur while RT_TASK1 is running are measured with each iteration, and the average and maximum values of the evaluation metric. Now, input data of RT_TASK1, that is ttsprk01, is constant, and this input data maximizes the execution time of RT_TASK1. This prevents fluctuations in the execution time of RT_TASK1 that are due to differences in input data, or paths in the program code. Furthermore, the execution time of RT_TASK1 does not include cache effects. In addition, the average execution time of each TLB miss exception handler is 16.9 microseconds.

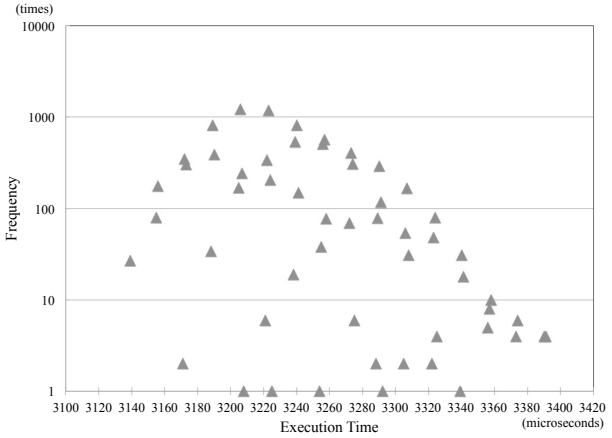


Fig. 2. Execution Time Distribution of the RT_TASK1

Figure 2 shows the execution time distribution of RT_TASK1. The horizontal axis indicates execution time of RT_TASK1 in microseconds, and the vertical axis indicates frequency of occurrence in logarithmic scale. The minimum execution time is 3,139 microseconds, and a TLB miss does not occur in that case. The WCET is 3,391 microseconds, and TLB misses occur 15 times in that case. The average and maximum values of the evaluation metric are 2.91 percent and 8.13 percent, respectively. Therefore, the WCET of RT_TASK1 includes the execution time of TLB miss exception handlers that is more than 8.13 percent of the essential execution time of RT_TASK1. This result shows that the impact of TLB misses on the WCET can be considerable.

IV. IMPROVEMENT ON WCET WITH TLB LOCKING

This section presents methods to reduce TLB misses and improve WCETs by taking advantage of TLB locking. TLB locking makes a specified TLB entry not to be replaced (expired) and inhibit the occurrence of TLB misses related to that entry. The proposed methods apply TLB locking to entries that are accessed by a real-time task in order to reduce TLB misses that occur while a real-time task is running and improve the WCET of a real-time task.

Generally, TLB locking assigns a normal entry of TLB as a locked entry. Furthermore, if a locked entry is released, this entry can be used as a normal entry. In the case that a TLB miss exception and TLB management is handled by an RTOS, TLB locking is implemented in software, and an RTOS should support TLB locking. On the other hand, some of ARM processors implement TLB locking in hardware [12]. In this paper, TLB locking is implemented in an RTOS, because evaluation target is an SH7750R processor as mentioned in Section II.

This paper proposes the two types of TLB locking methods: the method that statically locks target TLB entries (static locking method), and the method that dynamically locks those (dynamic locking method). Now, this paper assumes that all of the TLB entries which are accessed by a real-time task can be locked. However, if the number of TLB entries that are accessed by a real-time task is more than the number of TLB

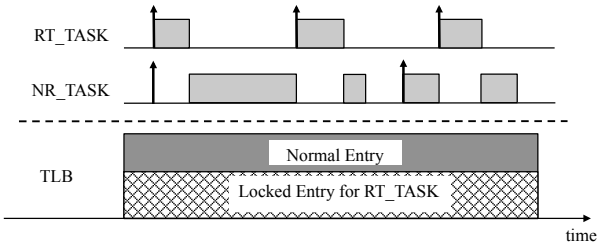


Fig. 3. Static Locking Method

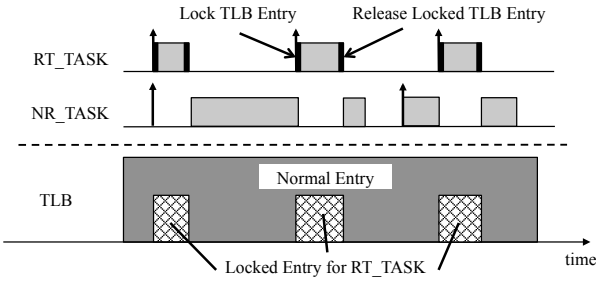


Fig. 4. Dynamic Locking Method

entries that can be locked, all of the entries that are accessed by a real-time task cannot be locked. In that case, an algorithm to select locked TLB entries so that WCETs are improved as effectively as possible should be considered.

A. Static Locking Method

The static locking method locks the target TLB entries during system initialization, and these entries remain locked while the system is running. Figure 3 shows a schema for the static locking method. This method locks the TLB entries that are accessed by a real-time task (RT_TASK) during system initialization. After that, those entries remain locked and only the rest of entries are used while non real-time task (NR_TASK) is running.

An advantage of the static locking method is that it is easy to analyze the WCET of RT_TASK. Furthermore, if all of the entries that are accessed by RT_TASK can be locked, the WCET of RT_TASK is deterministic because a TLB miss does not occur while RT_TASK is running. Another advantage is that TLB locking increase only the execution time of system initialization and other execution time is not increased. By contrast, a disadvantage is that the response time of an entire system can be increased. This is because the static locking method reduces TLB entries that NR_TASK can use, and TLB misses can be increased while NR_TASK is running.

B. Dynamic Locking Method

The static TLB locking can increase the response time of an entire system as described in previous section. If TLB entries that are accessed by a real-time task are locked and there are few TLB entries that are used by a non real-time task, TLB misses can be increased while a non real-time task is running. Thus, the response time of an entire system can be increased by TLB locking. In order to consider a solution for this challenge, the dynamic locking method is proposed in addition to the static locking method, and this paper evaluates both methods.

The dynamic locking method locks the target TLB entries when a job of a real-time task is activated, and releases the locked TLB entries when a job of a real-time task ends. After the locked TLB entry is released, this entry can be used as a normal entry. In order to simplify the problem, this paper assumes that a real-time task is not preempted. Figure 4 shows a schema for the dynamic locking method. This method locks the TLB entries that are accessed by RT_TASK when RT_TASK is activated. After that, those locked entries are released when RT_TASK ends.

The execution time of TLB locking should be short as possible. It is inadequate to obtain a locked entry by a page table walk, because the execution time of TLB locking is the same as a TLB miss exception handler in this case. Hence, the proposed methods statically prepare a table for locked TLB entries, and obtain the entry from this table in order to reduce time to search locked entries.

An advantage of the dynamic locking method is that it is easy to analyze the WCET of RT_TASK, that is the same as the static locking method. Another advantage is that the response time of an entire system has less potential to be increased than the static locking method. This is because the locked TLB entries are released while NR_TASK is running, TLB misses can be less than that in the case of the static locking method. By contrast, a disadvantage is that TLB locking increases the execution time of RT_TASK to lock and release TLB entries when every RT_TASK is activated and ends, respectively.

Moreover, before TLB locking, an entry which indicates the same virtual page of memory as a locked entry shall be invalidated. If different TLB entries indicate the same virtual page, an exception occurs. In the case of the static locking method, it is adequate to invalidate all TLB entries. This is because the static locking method locks an entry only during system initialization and invalidation of all entry at this time has little impact on subsequent TLB misses. Furthermore, invalidation of all entry can be done by only one memory access with an SH7750R processor, which is an access to the MMU control register. However, in the case of the dynamic locking method, invalidation of TLB entries is required when every RT_TASK is activated. Therefore, invalidation of entries can have much impact on TLB misses so that the response time of an entire system can be increased.

In order to consider a better method to invalidate a TLB entry, two methods are proposed and evaluated for the dynamic locking method: the dynamic locking method 1 and 2. The dynamic locking method 1 invalidates only TLB entries related to a locked entry. This method prevents invalidation of an entry which does not related to a locked entry. However, the execution time can be increased because invalidation is done with each entry. The dynamic locking method 2 invalidates all TLB entries. In addition, this method saves all TLB entries to memory before invalidation, and restores all TLB entries from memory when locked entries are released. This method can be expected not to increase the execution time of NR_TASK because TLB entries for NR_TASK are not expired by RT_TASK. However, saving and restoring all TLB entries

can increase the response time of an entire system.

V. Evaluation for TLB Locking Method

In this section, the effectiveness of the proposed TLB locking methods is evaluated with the micro-benchmark, which is described in Section II. We measured the execution time distribution and the evaluation metric, which is described in Section III, of RT_TASK1. In addition, we measured the execution time of each dynamic locking method, and the execution efficiency of the entire micro-benchmark.

Now, the static locking method locks TLB entries before NR_TASK is activated. The dynamic locking method locks TLB entries when RT_TASK1 is activated, and releases the locked entries when RT_TASKm (m = 1 or 5) ends. Each TLB locking method is evaluated in two cases: that each method is applied to only RT_TASK1, or that each method is applied to all of RT_TASKn (n = 1, 2, 3, 4, 5). In the case that each method is applied to only RT_TASK1 and all of RT_TASKn, 16 entries and 49 entries are locked, respectively. Now, in the case that each method is applied to only RT_TASK1, input the constant parameters to RT_TASK1, which are the same as the evaluation in Section III. On the other hand, in the case that each method is applied to all of RT_TASKn, input the random parameters to each RT_TASKn, which are provided in EEMBC benchmark software.

The execution time of each dynamic locking method is evaluated by varying the number of locked entries. Here, the execution time of a dynamic locking method includes a time to lock, release and invalidate TLB entries. Meanwhile, the execution efficiency of the entire micro-benchmark is evaluated by the response time, the execution time and TLB misses of NR_TASK Here, the response time is a time between after NR_TASK is activated and before NR_TASK ends, and the execution time is a time that NR_TASK is running.

A. Evaluation Result

Figure 5 shows the execution time distribution of RT_TASK1 in the case that each TLB locking method is applied to only RT_TASK1 with the result of Figure 2. The horizontal axis indicates execution time of RT_TASK1 in microseconds, and the vertical axis indicates frequency of occurrence in logarithmic scale. This result shows that all TLB locking methods can get rid of a TLB miss for RT_TASK1, and the evaluation metric is 0 percent in the all cases. In other words, no TLB miss occurs in the execution of RT_TASK1 with any TLB locking methods and the execution time is constant.

Figure 6 shows the execution time of each dynamic TLB locking method with that of a TLB miss exception handler. The horizontal axis indicates the number of locked TLB entries, and the vertical axis indicates the execution time in microseconds. This result shows that the execution time of the dynamic locking method 1 is shorter than TLB miss exception handler, if more than 2 entries are locked. The execution time of the dynamic locking method 1 increases 9.6 microseconds for each additional one locked entry, whereas that of one TLB miss exception handler is 17 microseconds. This difference is caused by preparing a table for locked entries statically, as described in Section IV-B. On the other hand, the dynamic

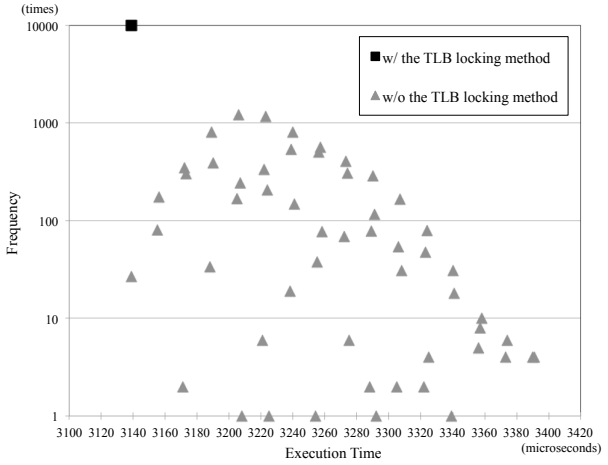


Fig. 5. Execution Time Distribution of the RT_TASK1 with TLB Locking

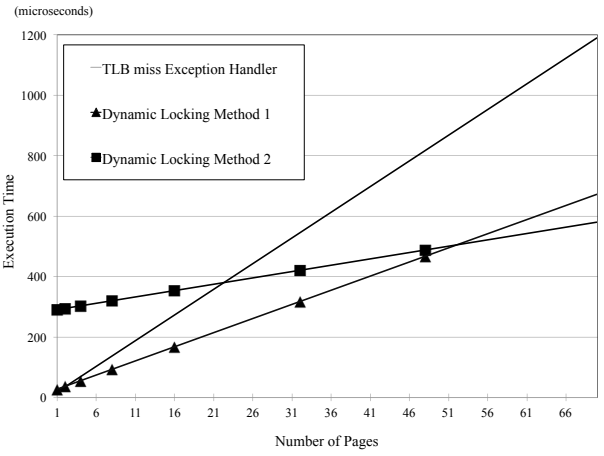


Fig. 6. Execution Time for TLB Locking Methods

locking method 2 takes 290 microseconds to lock only one entry. This is caused by saving and restoring all TLB entries. However, the execution time of the dynamic locking method 2 increases 4.2 microseconds for each additional one locked entry, and this increase is less than that of the dynamic locking method 1. Thus, the execution time of the dynamic locking method 2 is the shortest of the all TLB locking methods, if the number of locked entries is more than 52.

Figure 7 and Figure 8 show the result of the execution efficiency of the entire micro-benchmark in the case that each TLB locking method is applied to only RT_TASK1 and all RT_TASKn, respectively. These figures indicate the response time, the execution time and TLB misses of NR_TASK. The horizontal axis indicates the applied TLB locking method. The left vertical axis indicates the response time and the execution time, and the right vertical axis indicates the number of TLB misses. In these figure, the response time, the execution time and the number of TLB misses are indicated as a ratio of each value with a TLB locking to each value without a TLB locking.

In the case that the static locking method is applied, if few TLB entries are locked (Figure 7), every evaluated value

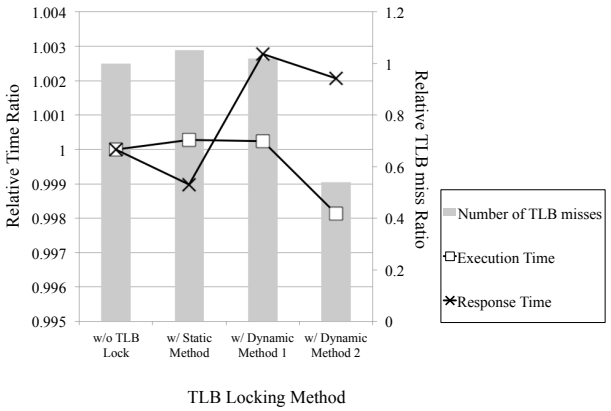


Fig. 7. Results for the NR_TASK with Appling TLB Locking to RT_TASK1

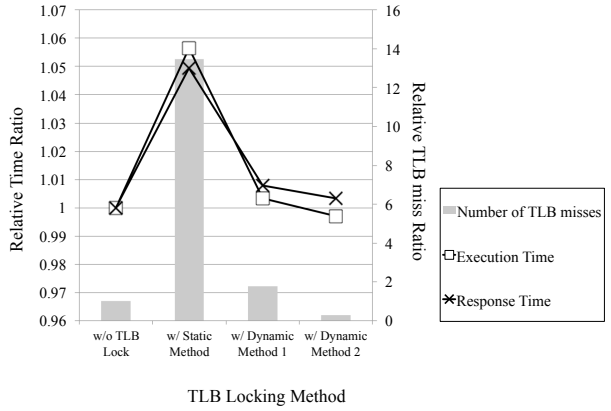


Fig. 8. Results for the NR_TASK with Appling TLB Locking to All RT_TASK

is ranged in 0.1 percent. The response time with the static locking method is less than that without a locking method. This is because the TLB locking minimizes the execution time of RT_TASK1. Otherwise if many TLB entries are locked (Figure 8), TLB misses occur 13 times as many as the case without a TLB locking, and the response time increases by 5 percent. This is because TLB entries which NR_TASK can be used are reduced, and TLB misses occur frequently.

In the case that each dynamic locking method is applied, the dynamic locking method 2 has less impact on NR_TASK than the dynamic locking method 1. This result shows that saving and restoring all TLB entries for NR_TASK are much more effective.

The result of this evaluation shows that all TLB locking methods are effective to improve WCETs. On the other hand, the execution time of a TLB locking method and the execution efficiency of an entire system is different depending on the locking method. As a result, if few TLB entries are locked, the static locking method is the most effective. Otherwise if many TLB entries are locked, the dynamic locking method 2 is the most effective. However, this conclusion can be depending on this evaluation environment and a different environment can make a different conclusion.

VI. Conclusion

This paper has presented the impact of TLB misses on WCETs of a real-time system and method to improve TLB misses and WCETs by using TLB locking. The impact of TLB misses is evaluated with the micro-benchmark, and the result shows that the impact of TLB misses on the WCET can be considerable. In order to reduce the impact of the TLB misses, the TLB locking methods in RTOS are proposed, which are the static locking method and dynamic locking method. The proposed methods apply TLB locking to entries which are accessed by a real-time task, and improve a WCET of a real-time task. Meanwhile, when a TLB entry is locked, other TLB entries related to the same virtual page as a locked entry shall be invalidated. Hence, the methods to invalidate TLB entry are proposed. The proposed methods are evaluated with the micro-benchmark. The result of evaluation shows that all TLB locking methods are effective to improve WCETs. Moreover, this evaluation shows that the TLB lock methods have different features each other. Thus, it is depending on a system which method should be applied.

As a future work, the impact of TLB misses on WCETs and the TLB locking methods should be evaluated with other environments, for example, different task sets, ARM processors which implement TLB locking in hardware, and cache enabled. In addition, algorithm to select a locked TLB entry should be considered for the case that all TLB entries which are accessed by a real-time task cannot be locked because of lack of a TLB entry which can be locked. Furthermore, if the proposed methods are implemented in hardware, more improved WCET and execution efficiency can be expected.

References

[1] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska, "Sharing and protection in a single-address-space operating system," *ACM Transactions on Computer Systems*, vol. 12, no. 4, pp. 271–307, Nov. 1994.

[2] F. Bruns, D. Kuschnerus, A. Showk, and A. Bilgic, "An extensible partitioning framework for safety-critical systems," in *Embedded Real-Time Software and Systems 2012*, Feb. 2012.

[3] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[4] J. P. Lehoczky, "Fixed priority scheduling of periodic task sets with arbitrary deadlines," in *Proceedings of the 11th IEEE Real-Time Systems Symposium*, 1990, pp. 201–209.

[5] J. M. López, J. L. Díaz, J. Entrialgo, and D. García, "Stochastic analysis of real-time systems under preemptive priority-driven scheduling," *Real-Time Systems*, vol. 40, no. 2, pp. 180–207, 2008.

[6] *SH7750, SH7750S, SH7750R Group Hardware Manual*, Renesas Electronics.

[7] TOPPERS/HRP2 kernel, http://www.toppers.jp/en/hrp2-kernel.html.

[8] TOPPERS Project, Inc., http://toppers.jp/en/index.html.

[9] TRON Association, "Protection Extention of μITRON4.0 Specification," 2002.

[10] R. Racu, A. Hamann, R. Ernst, and K. Richter, "Automotive software integration," in *Proceedings of the 44th annual Design Automation Conference*, 2007, pp. 545–550.

[11] EEMBC – The Embedded Microprocessor Benchmark Consortium, http://www.eembc.org/.

[12] ARM, "ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition."

Implementation of the Multi-Level Adaptive Hierarchical Scheduling Framework

Nima Moghaddami Khalilzad, Moris Behnam, Thomas Nolte
MRTC/Mälardalen University
P.O. Box 883, SE-721 23 Västerås, Sweden
nima.m.khalilzad@mdh.se

***Abstract*—We have presented a multi-level adaptive hierarchical scheduling framework in our previous work. The framework targets compositional real-time systems which are composed of both hard and soft real-time systems. While static CPU portions are reserved for hard real-time components, the CPU portions of soft real-time components are adjusted during run-time. In this paper, we present the implementation details of our framework which is implemented as a Linux kernel loadable module. In addition, we present a case-study to evaluate the performance and the overhead of our framework.**

I. Introduction

Hierarchical scheduling and resource reservation techniques have been widely used for composing independent hard real-time systems on a shared underlying hardware [1]. Using such techniques, the timing behavior of the individual systems (components) are studied in isolation, while the correctness of the entire systems is inferred from the correctness of the individual components before the composition. This compositional timing study is especially useful in open systems in which components are added or removed during the system's life time [2].

Hierarchical scheduling is often performed through CPU reservations. When dealing with hard real-time systems, based on the worst case CPU demand of the individual components, a CPU portion is reserved for each component such that the component's inner tasks are guaranteed to receive enough CPU resource time to complete their executions in time.

While there exists a variety of techniques to handle the composition when composing hard real-time systems (e.g., [2], [3], [4], [5]), the problem of composing soft and hard real-time systems together has not been deeply studied. A considerable group of soft real-time systems have the following attributes. First of all, they demonstrate a wide difference between their worst case and average case CPU demands. Secondly, occasional timing violations can be tolerated in these type of systems. Last but not least, resource demand analysis are rarely done for these systems. As a result, designers do not have enough information about the resource demand requirements of such systems. Note that the timing requirements are often known a priori, whereas, the resource requirements are unknown. In dealing with real-time systems that have the aforementioned attributes, reserving the CPU portions based on the worst case CPU demand of the tasks is not an efficient design approach. Because even if the worst case CPU demand is available, it will result in an unnecessary CPU overallocation. Consequently, the CPU resource will be wasted.

To address this problem, we presented an adaptive framework in [6] where for hard real-time systems we reserve the CPU portions based on their worst case resource demand. On the other hand, soft real-time systems receive dynamic CPU portions based on their actual need at each point in time. For acquiring the resource demand of the soft real-time components, we monitor their behavior during run-time and use the gathered information to adjust the component CPU reservations. While the design and evaluation of our framework is presented in [6], in this paper we present the implementation details of our **Ad**aptive **Hier**archical **Sched**uling (`AdHierSched`) framework[1]. In particular we present the following contributions in this paper.

- The data structures and the mechanisms that are used in the implementation of our framework as a Linux kernel loadable module.
- The performance evaluation of `AdHierSched` with respect to timing requirements of the real-time components.
- The overhead evaluation of the scheduler and the CPU reservation adapter component.

The rest of the paper is organized as follows. The related work is reviewed in Section II. In Section III we present our system model. Section IV presents the data structures and implementation techniques used in `AdHierSched`. We evaluate the performance of `AdHierSched` in Section V. The overhead of `AdHierSched` is presented in Section VI. Finally, the paper is concluded in Section VII.

II. Related work

There exists an enormous number of papers that address the implementation of real-time schedulers (e.g, RTLinux [7] and RTAI [8]). However, in this paper we only review a part of

[1]The source code is available at: http://www.idt.mdh.se/~adhiersched.

them that focus either on hierarchical scheduling or on adaptive scheduling.

In [9] hierarchical scheduling is done on top of the VxWorks operating system. Hierarchical scheduling on top of the FreeRTOS operating system is presented in [10]. ExSched [11] is a platform independent real-time scheduler which has a hierarchical scheduling plug-in. Hierarchical scheduling is also implemented in μC/OS-II [12]. All of these works are two-level hierarchical schedulers and are designed for hard real-time applications, i.e, they are not adaptive frameworks.

HLS [13] is a multi-level hierarchal scheduling implemented in Windows 2000 which targets composing soft real-time systems. In [14], Parmer and West presented a hierarchical scheme for managing CPU, memory and I/O. These frameworks are not adaptive in the sense that the resource demands are not monitored and hence the resource reservations (if used) are fixed during run-time.

Hierarchical scheduling is also used for virtualization purposes. Recursive virtual machines are proposed in [15] where each virtual machine can directly access the microkernel. A two-level hierarchical scheduler using L4/Fiasco as the hypervisor is presented in [16]. Lee *et al.* developed a virtualization platform using the Xen hypervisor [17]. A Virtual CPU scheduling framework in the Quest operating system is developed by Danish *et al.* [18]. In [19], the CPU reservations are used for scheduling virtual machines. The VirtualBox and the KVM hypervisores are scheduled using CPU reservation techniques in [20].

The AQuoSA framework [21] is an adaptive framework implemented in Linux which uses the CPU reservation techniques together with feedback loops to adjust the reservations during run-time. Our work is different than AQuoSA in the following two aspects. First of all, we target multi-level hierarchical systems while AQuoSA only targets flat systems, i.e the systems with one task per CPU reservation and without local schedulers. Secondly, we have implemented `AdHierSched` as a kernel loadable module, whereas, AQuoSA requires kernel patching.

ACTORS [22] is an adaptive framework which targets multicore systems. In this framework, the CPU reservations are used for providing isolation among real-time tasks, while the reservation sizes are being adjusted during run-time. ACTORS uses `SCHED_DEADLINE` [23] for implementing the CPU reservations. Similar to the AQuoSA framework, ACTORS addresses flat systems and not hierarchical systems.

AIRS [24] is a framework designed to provide high quality of service to interactive real-time applications. AIRS uses a new CPU reservation scheme as well as a new multiprocessor scheduling policy. Alike AQuoSA and ACTORS, AIRS targets flat systems.

III. Model

In this section, we explain how we model servers, tasks and systems.

A. Server model

We use the periodic resource model [25] in our framework, and we implement the periodic model using the periodic servers which work as follows. The servers are released periodically, providing their children with a predefined amount of the CPU time in each period. The periodic servers idle their CPU allocation if there is no active task/server inside them. Any server implementation compliant with the periodic resource model can be used in our framework. A periodic server is represented with the following 4-tuple $S_i^j =< P_i^j, B_i^j, Pr_i^j, \zeta_i^j >$, where P_i^j, B_i^j, Pr_i^j and ζ_i^j represent period, budget, scheduling priority and importance of server S_i^j. The importance value represents the relative importance of the servers with respect to their other sibling servers. This parameter is only used in an overload situation where the total CPU demand is more than the available CPU. In the overload situations, `AdHierSched` prioritizes the servers in the order of their importance. Note that the overload situation is only considered to happen in soft real-time servers. The superscript in the server notation, represents the parent server index. Based on the scheduling policy of S_j, a subset of the parameters in the server 4-tuple are used. For instance, when the scheduling is done according to EDF, Pr_i^j is ignored. In the case of soft real-time servers, B_i^j is adapted during run-time. Thus, the budget is a function of time $B_i^j(t)$. Consequently, server's children may receive a different share of the CPU in different server periods. The adaptation is done through the budget controller component based on online monitoring of the server's workload. The budget controller adapts the budgets such that each server receives just enough CPU time at each server period. The details of the adaptation mechanism is presented in our previous work [6].

Furthermore, server S_i^j is composed of n_i child servers and m_i child tasks. Server S_i^j schedules its children according to its local scheduling policy. Servers and tasks inherit the type of their parents, e.g., if a server is a soft-real-time server its children will also be treated as soft real-time servers/tasks. At each point in time there is at most one server assigned to the CPU which is called the "active server".

Since our adaptation mechanism is designed for the periodic servers, we focus on the explanation of the periodic servers in this paper. Nevertheless, the Constant Bandwidth Server (CBS) [26] is implemented in `AdHierSched`, and it can be used inside the periodic servers for providing timing isolation among tasks and servers that reside inside the same periodic server parent.

B. Task model

We assume the periodic task model in which a periodic task τ_i^j, which is a child of server S_j, is represented using the following parameters: task period T_i^j, task deadline D_i^j, worst case execution time C_i^j and task priority π_i^j. Similar to the server model, depending on the parent scheduling policy some task parameters may be ignored. At each point in time, at most one task is assigned to the CPU which is called the "running task". In the case of soft real-time tasks, we assume that the

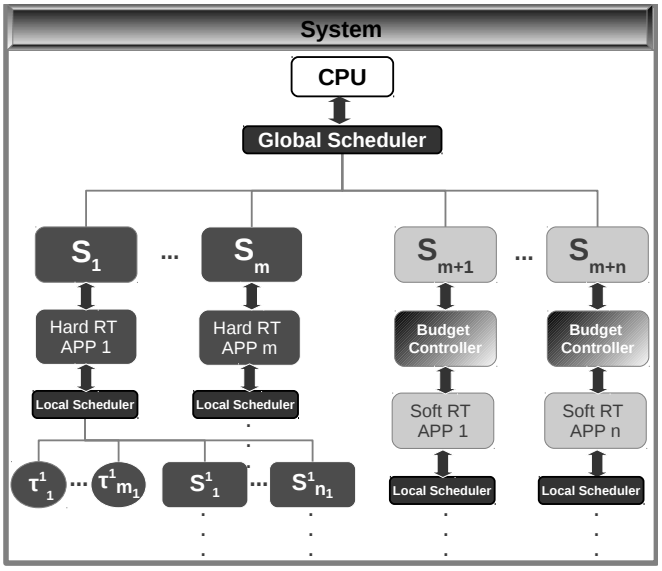


Fig. 1: Visualization of the system model.

tasks are dynamic, i.e, their execution times are changing in a wide range during run-time and their execution time is not known a priori. One instance of the task execution is called a job.

C. System model

We assume a single processor system which consists of n soft real-time servers and m hard real-time servers at the root level. The servers contain applications which consist of tasks and/or sub-servers. Therefore, our system model is a multi-level hierarchical model. A system has a global scheduler which schedules the servers and tasks at the root level of the hierarchy. In addition, there is a local scheduler in each server which is responsible for scheduling the server's inner children (both tasks and servers). Figure 1 illustrates our hierarchical system model. The hard real-time applications are shown using dark gray background in the figure. There is a budget controller component attached to the soft real-time servers. The budget controller component monitors the CPU demand of the applications and assigns a sufficient CPU portion to the servers.

Considering the system hierarchy, we would like to present two definitions that are used in the later sections of the paper for explaining the implementation of the scheduler.

Definition 1: S_i^j is an **ancestor** of S_κ^l if either $i = l$ or by upward traversing the parent of S_l we reach S_i^j. For instance, S_1 is an ancestor of S_5^3 in Figure 2.

Definition 2: S_i^j **outranks** S_κ^l if and only if one of an ancestor of S_l is S_j. For instance, S_2 outranks S_3^1 in Figure 2.

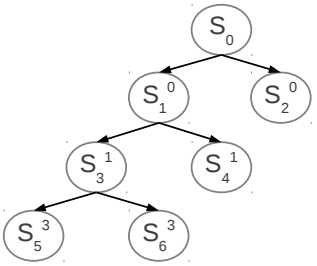


Fig. 2: Example tree structure (S_0 represents the root scheduler).

IV. Framework

In this section, we explain how our assumed model is implemented as a Linux kernel loadable module.

A. Real-time scheduling through a kernel loadable module

We use a similar idea to the work presented in [11]. The idea is to implement a real-time scheduler in Linux without modifying the kernel. To this end, we developed a kernel loadable module that plays a middleware role between real-time tasks and the Linux kernel. The module is responsible to release, run and stop the real-time tasks. When a task has to run, the module inserts it into the Linux run queue and changes its state to running. On the other hand, when the module has to stop a real-time task, it removes the task from the Linux run queue and the task goes to the sleep state. Thus, at each point in time, there is at most one real-time task (priority 0 to 99) in the Linux run queue. Consequently, no matter which Linux real-time scheduling class is used, the `schedule()` system call will always pick the single real-time task that is in the Linux run queue. Figure 3 illustrates the relation between the `AdHierSched` module and the Linux run queue.

B. Managing time

In order to manage the scheduling events, we use the classic Linux timers (low-resolution timers) available in `kernel/timer.c`. As will be explained in the rest of this section, we use one timer per task and two timers per server for managing their corresponding scheduling events. Therefore, `AdHierSched` does not have a release queue and instead it delegates the job of the release queue to the Linux timer list. Since the Linux timer list is implemented using the red-black trees, when the number of timers increases, retrieving and inserting them are still efficient $(O(logn))$. Nevertheless, we assume that systems will not excess a handful of levels, hence n will not be a large number. We insert the timers using the `setup_timer_on_stack` and `mod_timer` system calls, and remove them using the `del_timer` system call.

In order to convert the relative scheduling parameters to absolute parameters, we use the `jiffies` variable available in the kernel which return the current time.

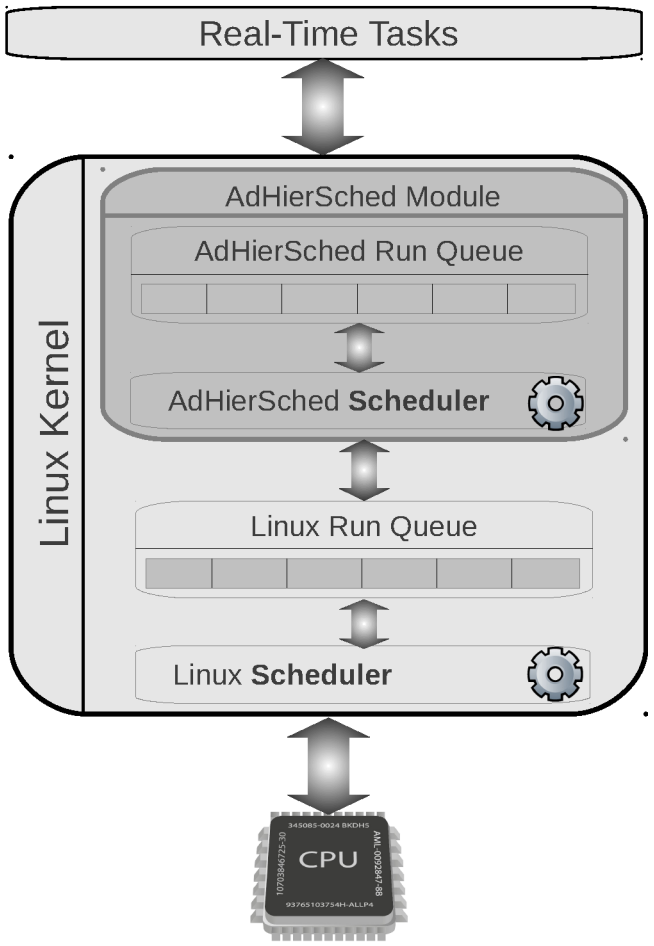


Fig. 3: `AdHierSched` module.

C. Task and server descriptors

`AdHierSched` uses its own task and server descriptors to store their corresponding information. The task descriptor, which is presented in Code Snippet 1, has a `timer_list` member called `period_timer` (line 17) which is used for running tasks periodically. When a task finishes its job, the `period_timer` is set to the next release of the task. Each `AdHierSched` task points to a Linux task (line 16). Tasks may be attached to a periodic server (line 18) and/or a CBS (line 19). The CBSs may be used for providing timing isolation among tasks that reside inside the same parent, i.e periodic server. The `dl_miss` member in line 7 and the `dl_miss_amount` member in line 15 are used to monitor the load situation of the tasks. These fields are used by the budget controller component to adapt the budget of servers. The `timestamp` member is used for monitoring the duration of the scheduling events such as the duration that tasks are assigned to the CPU.

The server descriptor is presented in Code Snippet 2. Our framework supports both constant bandwidth servers

Code Snippet 1: Task descriptor.

```
1:  struct Task {
2:  struct list_head head;
3:  int id;
4:  int priority;
5:  int state;
6:  int cnt; /* job number */
7:  int dl_miss; /* number of deadline misses */
8:  int missing_dl_flag;
9:  unsigned long period;
10: unsigned long release_time;
11: unsigned long exec_time;
12: unsigned long relative_deadline;
13: unsigned long abs_deadline;
14: unsigned long timestamp;
15: unsigned long dl_miss_amount;
16: struct task_struct *linux_task;
17: struct timer_list period_timer;
18: struct Server *parent;
19: struct Server *cbs; };
```

and periodic servers. The `type` member in line 4 indicates whether the server is a constant bandwidth server or a periodic server. The `children` member in line 3 is a pointer to the scheduling entities that are inside the server. The `control_period` field in line 8 stores the budget adaptation frequency. The `current_budget` field stores the remaining budget at each time point. The fields from line 15 to line 17 are used for the budget adaptation purpose. Each server has its own ready queue (line 19) which contains the child tasks and servers that are ready to run. The server structure has two `timer_list` members: `period_timer` and `budget_timer`. The `period_timer` is used for periodically releasing the servers, whereas, the `budget_timer` is used to stop the servers when their budget is depleted.

D. Timer handlers

There are two types of handlers: (i) release handlers (ii) budget depletion handlers. We have implemented each handler in a separate function. The list of timer handlers is as follows. (i) Task release, (ii) Server release, (iii) Periodic server budget depletion, (iv) CBS budget depletion.

E. Queue structure

We define the "scheduling entity" type as a generic type which covers both tasks and servers. Therefore, an entity can be either a task or a server. The ready queues store the scheduling entities in the order of their priority. The ready queue is implemented as a linked list through the `list_head` structure available in the Linux kernel. We have implemented two functions for inserting/deleting an entity to/from queue:

- `insert_queue(queue, entity)`
- `delete_queue(entity)`

Code Snippet 2: Server descriptor.

```
1: struct Server {
2: struct list_head head;
3: Children children;
4: int type;
5: int id;
6: int priority;
7: int cnt; /* number of jobs */
8: int control_period;
9: int importance; /* ζ*/
10: unsigned long budget;
11: unsigned long period;
12: unsigned long relative_deadline;
13: unsigned long abs_deadline;
14: unsigned long current_budget;
15: unsigned long consumed_budget;
16: unsigned long extra_req_budget;
17: unsigned long total_budget;
18: unsigned long timestamp;
19: struct Queue *ready_queue;
20: struct timer_list period_timer;
21: struct timer_list budget_timer;
22: struct Server *parent; };
```

1) Fixed priority scheduling: When the scheduling policy is fixed priority, the `insert_queue` function inserts the new entities based on their priorities.

2) EDF scheduling: The insertion to the ready queue, when the scheduling policy is EDF is based on the `abs_deadline` of the scheduling entities.

Note that since we use multiple ready queues (one queue per server):

$$q_i^j \leq n_i^j + m_i^j,$$

where q_i^j is the number of elements in the ready queue of S_i^j. Hence, the complexity of the insertion to the queue is $O(q_i^j)$.

F. Communication between tasks and `AdHierSched`

The communication between tasks and `AdHierSched` is done through a device file. The `AdHierSched` library provides a number of API functions. The API functions use the `ioctl()` system call for the communication purpose. When the message is delivered to the `AdHierSched` module, it relays the message to the message's corresponding function. The list of provided API functions is presented in Table I. The names of the functions are self explanatory, however, we explain a few of them here. As soon as `AdHierSched` receives a `run()` message, it releases all of the servers and tasks immediately. So, the release time of all scheduling entities will be equal if this function is used. The `stop()` function first stops inserting new timers to the timer list, i.e, it stops the release events. Secondly, it calls the `wake_up_process()` system call for all of the tasks that are still running. In other words, when the `stop()` function is called, the `AdHierSched` module no longer operates and Linux takes the complete responsibility of scheduling the real-time tasks. The `task_finish_job(task_id)` function should be called at the end of the task jobs. This call indeed changes the task status to sleep until the next release of the task. Note that it is possible to add/remove tasks and servers through the API functions while the module is running.

run()
stop()
create_task()
detach_task(task_id)
release_task(task_id)
task_finish_job(task_id)
detach_server(server_id)
release_server(server_id)
attach_task_to_mod(task_id)
create_server(queue_type, server_type)
attach_server_to_server(server_id, server_id2)
attach_task_to_server(server_id, task_id, server_type)
set_task_param(task_id, period, deadline, exec_time, priority)
set_server_param(server_id, period, deadline, budget, priority, server_type)

TABLE I: List of provided API functions by `AdHierSched` library.

G. Configuration and run

The API functions allow the users to configure their target system, i.e, to create their desirable hierarchy and to set the scheduling parameters. Once the system is set up, the Linux tasks need to be attached to the `AdHierSched` tasks using the `attach_task_to_mod(task_id)` API function. A sample task structure is presented in Code Snippet 3. Finally, the `run()` API function needs to be called to release all servers and tasks. When `AdHierSched` receives a `run()` call, it releases all servers and tasks and then tries to run them. Depending on the global level scheduling policy, among all released scheduling entities at the root level of the hierarchy, the one that has the highest priority or shortest deadline will be assigned to the CPU.

Code Snippet 3: Sample task structure.

```
1: int main(int argc, char* argv[]){
2: task_id = atoi(argv[1]);
3: attach_task_to_mod(task_id);
4: while i < job_no do
5:    /* periodic job */
6:    task_finish_job(task_id);
7: end while
8: detach_task(task_id);
9: return 0; }
```

If a server is assigned to the CPU, it will try to run its local ready queue. If from the server's ready queue a sub-server receives the CPU, the local ready queue running operation continues until the scheduler decides to run a task. As soon as server S_i^j becomes active, we insert its corresponding budget depletion timer (`budget_timer`) to be invoked at time t_{dep},

where:

$$t_{dep} = \texttt{jiffies} + B_i^j(t).$$

When the `jiffies` is equal to t_{dep}, the budget depletion timer handler is invoked. The handler deactivates its corresponding server (S_i^j) and all of its child servers. If S_i^j is an ancestor (see Definition 1) of the active server, the active server is stopped. If the running task is a child of the server that is getting deactivated, the running task is also stopped. Finally, the timer handler runs the first element that is in the ready queue of S_j (the parent of the server whose budget is depleted). When a server is stopped (either because of its parent budget depletion or because of a preemption), its remaining budget is updated.

Each scheduling entity belongs to a ready queue. The entities at the root level belong to the global ready queue, while the other entities belong to their parent server's ready queue. Therefore, when an entity causes a scheduling event, the event takes place at its corresponding ready queue.

The scheduling decisions are taken only at the scheduling events. We have the following scheduling events in the system.

- task and server release
- server (periodic and constant bandwidth) budget depletion
- task finishing its job
- task and servers leaving the system

When a task is released and the active ready queue is different than the task's ready queue, the task will wait until its ready queue, i.e., its server is activated. Even when the released task's parent is active, it will only be assigned to the CPU if it is able to preempt the running task or the active server. Note that the preemption rules depend on the parent's scheduling policy. When a server is released, it should wait unless one of the following conditions hold in which the released server is allowed to preempt the active server or the running task.

- The server's parent is active and the released server can preempt the running/active scheduling entity.
- The released server outranks (see Definition 2) the active server.

H. Budget adaptation

The budget adaptation is done periodically. The adaptation period is proportional to the server periods. The budget adaptation is done through a function which is called at certain server release events. When calling the budget adapter function, the pointer to the caller server structure is passed to the function. This function uses the `consumed_budget` and the `extra_req_budget` fields in the server data structure to derive the new `budget` field. The `extra_req_budget` field is updated by the server's child tasks and sub-servers that are violating their timing requirements. We also have a mechanism to guarantee that adapting the soft real-time server budgets does not influence the amount of provided budget to the hard real-time servers. For more details about the budget adaptation mechanism refer to [6].

V. Evaluation

In this section, we first design a case-study to study the performance of our framework. Thereafter, we present the results.

A. Tasks

As we mentioned earlier, `AdHierSched` mainly targets systems containing dynamic soft real-time applications. To this end, in our evaluations we use two types of dynamic real-time tasks. Moreover, we use tasks with fixed execution times. In general the following three types of tasks are used in the case-study.

1) Fixed execution time tasks (static tasks). These tasks are indeed a simple `C` program that contain a loop with a constant number of instructions.
2) Mplayer media player[2]. We have modified the source code of the Mplayer media player such that it registers itself to the `AdHierSched` module before starting the playback. Thus, the `AdHierSched` module schedules the player task. In addition, after decoding and playing frames, Mplayer uses the `task_finish_job(task_id)` API function to inform the `AdHierSched` module that a job execution is finished.
3) Image processing program. This program is developed using the OpenCV library and its objective is to filter a color range of its input frame. The input is a movie file to this application in our case-study.

B. Setup

We use an Intel Core i5-2540M processor clocked at 2.60 GHz in which only CPU 0 is active. Our hardware is equipped with 4 GB of memory. In addition, Ubuntu 12.04.2 with Linux kernel version 3.8.2 is used in the evaluations. The scheduler resolution is set to one millisecond.

C. Case-study

The case-study that we investigate in this paper is a system composed of five applications of which one is a hard real-time application and the rest are soft real-time applications. Figure 4 illustrates the structure of the case-study system that we are using in this section. Note that τ_1^1, τ_1^2 and S_1 are hard real-time tasks and a server respectively. The hard real-time server uses a fixed priority scheduler, while the rest of the servers use EDF schedulers for scheduling their children. The tasks are assumed to be ordered based on their priority, i.e, $\pi_1^1 > \pi_2^1$. We use different inputs for the same type tasks. The specifications of the tasks and servers used in the case-study are presented in Table II. All scheduling parameters presented in the table are in milliseconds. We assume that the servers are

[2] http://www.mplayerhq.hu

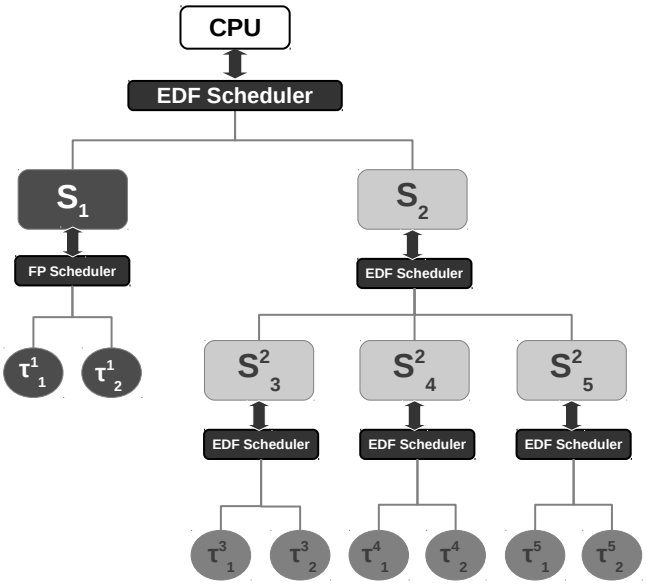


Fig. 4: The sample system investigated in the case-study.

	Hard-Soft	Task type	$P_j - T_i^j$
S_1	HRT server	-	100
τ_1^1	HRT task	1	200
τ_2^1	HRT task	1	400
S_2	SRT server	-	10
S_3^2	SRT server	-	20
τ_1^3	SRT task	2	40
τ_2^3	SRT task	3	200
S_4^2	SRT server	-	100
τ_1^4	SRT task	3	350
τ_2^4	SRT task	3	200
S_5^2	SRT server	-	75
τ_1^5	SRT task	3	350
τ_2^5	SRT task	3	150

TABLE II: Servers and tasks specification in the case-study.

ordered based on their importance meaning that $\zeta_3^2 > \zeta_4^2 > \zeta_5^2$.

D. Workload

In order to observe the workload of the applications, we ran each server separately while assigning 100 % of the CPU to them. The average and the maximum CPU demand of the tasks in the servers are reported in Table III. To illustrate the workload variations of the soft real-time serves we present the CPU demand percentage of S_3^2 in Figure 5.

Server	AVG	MAX
S_3^2	13.35	60.00
S_4^2	12.36	54.00
S_5^2	11.98	44.00
total	37.43	158

TABLE III: The CPU demand percentage of the servers.

Moreover, in our experiments we observe that $C_1^1 = C_2^1 =$ 31. Therefore, based on the suggestion presented in [25] we choose the following period for S_1: $P_1 = 100$. In addition, we derive the minimum budget that guarantees the schedulability of τ_1^1 and τ_1^2 using the analysis presented in [25] which is $B_1 = 39$. Therefore, 39 % of the total bandwidth will be assigned to S_1 and the rest of the bandwidth (61 %) may be utilized by S_2.

Server	fixed	adaptive
S_3^2	3.36	1.11
S_4^2	27.28	6.49
S_5^2	2.19	4.69
total	32.83	12.29

TABLE IV: The deadline miss ratio percentage of the servers.

E. Adaptive budgets

Recall that `AdHierSched` targets soft real-time applications for which their run-time behavior is not known a priori. Therefore, assuming that we have no information about the task CPU demands, we assign an initial budget to the servers and then we let the budget controller to adapt the budgets. We choose the deadline miss ratio as our performance metric that is the number of jobs that finish their execution after their deadline points, divided by the total number of finished jobs. The number of jobs for a server is equal to the sum of its tasks' jobs. As a result of assigning adaptive budgets, the soft real-time servers experience an average of 4.09 % deadline miss ratio. While the most important server S_3^2 experiences only 1.1 % deadline miss ratio. As we show in the rest of this section, the system is overloaded. Thus, missing deadlines is inevitable. However, adapting the bandwidth of the servers, we are serving the real-time tasks in such a way that the available CPU bandwidth is efficiently utilized.

F. Static budgets

Allocating the soft real-time server budgets based on the maximum demand of their tasks is impossible because the sum of the bandwidth (158 %) is more than the available bandwidth (61 %). Therefore, in another experiment we assign the server budgets based on their average CPU demand. Table IV summarizes the results of the case-study for both adaptive and static budget allocation experiments. The adaptive CPU allocation technique results in a total of 20 % less deadline misses than the static technique. In addition, since S_3^2 is the most important application in the system, the deadline misses avoided for this server is of more importance than the other deadline misses potentially avoided. Figure 6 illustrates the budget adaptations of the soft real-time servers used in the case-study.

VI. Overhead

In this section, we report the overhead imposed by the `AdHierSched` module in the case-study presented in Section V. Note that our measurements exclude the Linux scheduler overhead that is responsible to assign the `AdHierSched`

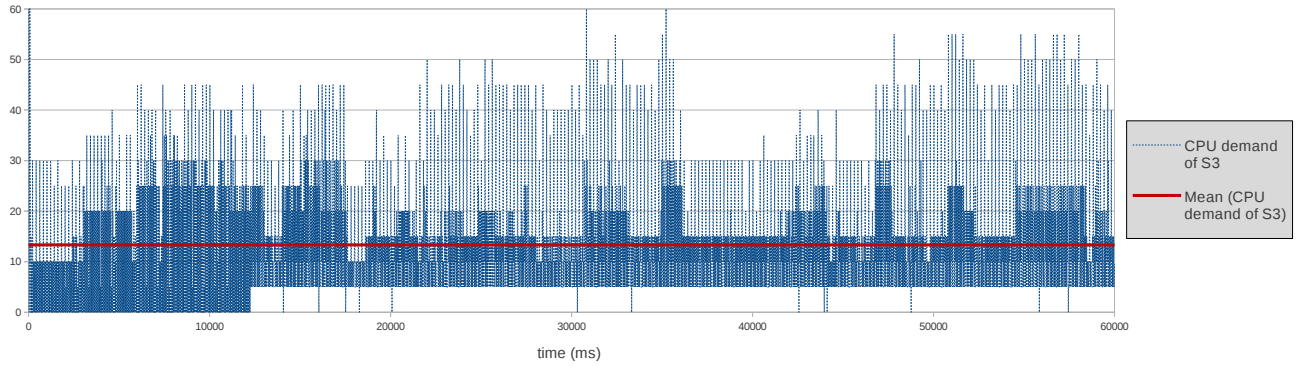


Fig. 5: The CPU demand variation of S_3^2.

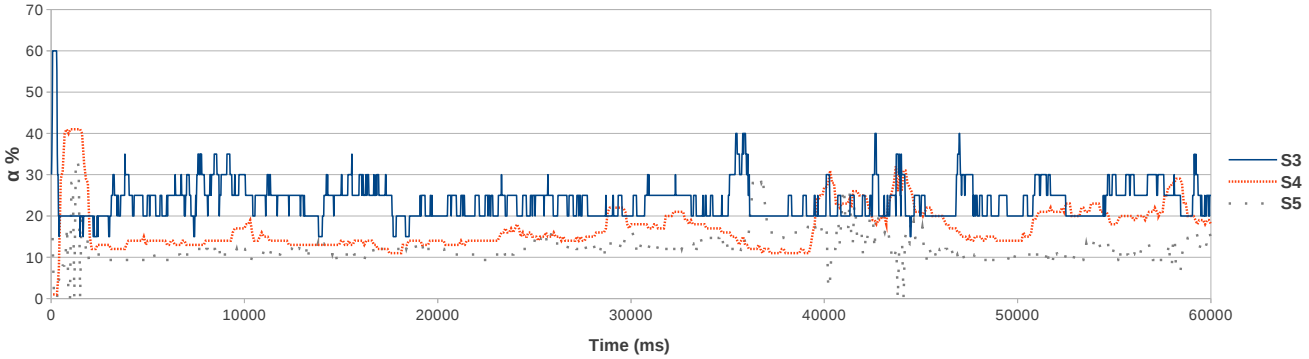


Fig. 6: The budget adaptations over the course of one minute experiment.

real-time tasks to the CPU. Therefore, we do not include the context switch overhead.

There are two sources of overhead: (i) the multi-level hierarchical scheduling overhead, i.e., the amount of extra calculation that is done just for scheduling the real-time time tasks in a hierarchical manner. (ii) the budget adaptation overhead, i.e., the amount of extra calculations that are done because of adapting the server budgets.

We measured the two types of overhead for the case-study. The total overhead is less than 0.2 % ($\simeq$ 122 milliseconds). Figure 7 shows that the budget adaptation overhead ($\simeq$ 8 milliseconds) has a small share of the total overhead. The figure represents the overhead present in the case-study explained in Section V. The overhead has been measured using time stamps that are monitoring the execution length of the timer handlers and the `task_finish_job(task_id)` API function. Then, the total value is divided by the total time that the experiment ran.

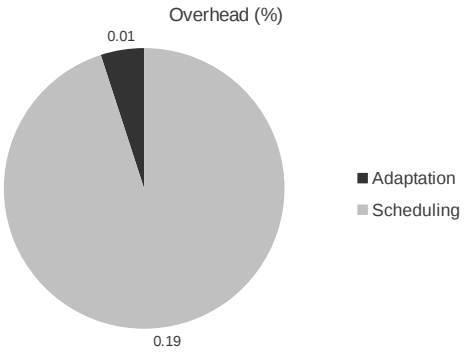


Fig. 7: The overhead of the `AdHierSched` module.

VII. Conclusion

In this paper, we presented the implementation details of our adaptive hierarchal scheduling framework which is called `AdHierSched`. We showed how the framework is implemented in the Linux kernel as a kernel loadable module. We demonstrated that our framework can efficiently deal with unknown workloads. Finally, we reported the overhead of our framework.

Our implementation can be improved in a number of ways. For instance, we can use a more efficient queue structure to reduce the overhead of the `AdHierSched` scheduler. As a

next step we are contemplating extending our framework to multiprocessors. Although we are not currently considering I/O operations, we would like to investigate the implications of modeling them in our adaptive framework. For instance, we can model the I/O requests as critical sections and we can use available semaphore based protocols such as SIRAP [27] and HSRP [28].

ACKNOWLEDGMENT

The authors wish to thank Shinpei Kato for his help with adopting the Mplayer code, and acknowledge the constructive comments of anonymous reviewers.

REFERENCES

[1] G. Lipari and S. Baruah, "A hierarchical extension to the constant bandwidth server framework," in *Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium (RTAS'01)*, May 2001, pp. 26–35.

[2] Z. Deng and J. W.-S. Liu, "Scheduling real-time applications in an open environment," in *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS'97)*, December 1997, pp. 308–319.

[3] A. Mok, X. Feng, and D. Chen, "Resource partition for real-time systems," in *Proceedings of the 7th Real-Time Technology and Applications Symposium (RTAS'01)*, May 2001, pp. 75–84.

[4] L. Almeida and P. Pedreiras, "Scheduling within temporal partitions: response-time analysis and server design," in *Proceedings of the 4th ACM International Conference on Embedded Software (EMSOFT'04)*, September 2004, pp. 95–103.

[5] F. Zhang and A. Burns, "Analysis of hierarchical EDF pre-emptive scheduling," in *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS'07)*, December 2007, pp. 423–434.

[6] N. M. Khalilzad, M. Behnam, and T. Nolte, "Multi-level adaptive hierarchical scheduling framework for composing real-time systems," in *Proceedings of the 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'13), to appear*, August 2013.

[7] M. Barabanov and V. Yodaiken, "Real-time linux," *Linux journal*, vol. 23, March 1996.

[8] P. Mantegazza, E. L. Dozio, and S. Papacharalambous, "RTAI: Real Time Application Interface," *Linux Journal*, vol. 2000, no. 72es, April 2000.

[9] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. J. Bril, "Towards hierarchical scheduling on top of VxWorks," in *Proceedings of the 4th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'08)*, July 2008, pp. 63–72.

[10] R. Inam, J. Maki-Turja, M. Sjodin, S. M. H. Ashjaei, and S. Afshar, "Support for hierarchical scheduling in FreeRTOS," in *Proceedings of the 16th IEEE International Conference on Emerging Technologies Factory Automation (ETFA'11)*, September 2011, pp. 1–10.

[11] M. Åsberg, T. Nolte, S. Kato, and R. Rajkumar, "ExSched: An external CPU scheduler framework for real-time systems," in *Proceedings of the 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'12)*, August 2012, pp. 240–249.

[12] M. van den Heuvel, R. J. Bril, J. J. Lukkien, and M. Behnam, "Extending a HSF-enabled open-source real-time operating system with resource sharing," in *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'10)*, July 2010, pp. 71–81.

[13] J. Regehr and J. Stankovic, "HLS: a framework for composing soft real-time schedulers," in *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS'01)*, December 2001, pp. 3–14.

[14] G. Parmer and R. West, "HIRES: A system for predictable hierarchical resource management," in *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'11)*, April 2011, pp. 180–190.

[15] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson, "Microkernels meet recursive virtual machines," in *Proceedings of the 2nd USENIX symposium on Operating systems design and implementation (OSDI'96)*, 1996, pp. 137–151.

[16] J. Yang, H. Kim, S. Park, C. Hong, and I. Shin, "Implementation of compositional scheduling framework on virtualization," *SIGBED Rev*, vol. 8, pp. 30–37, 2011.

[17] J. Lee, S. Xi, S. Chen, L. T. Phan, C. Gill, I. Lee, C. Lu, and O. Sokolsky, "Realizing compositional scheduling through virtualization," in *Proceedings of the 18th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'12)*, April 2012, pp. 13–22.

[18] M. Danish, Y. Li, and R. West, "Virtual-cpu scheduling in the quest operating system," in *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'11)*, April 2011, pp. 169–179.

[19] T. Cucinotta, G. Anastasi, and L. Abeni, "Respecting temporal constraints in virtualised services," in *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC'09)*, vol. 2, July 2009, pp. 73–78.

[20] M. Åsberg, N. Forsberg, T. Nolte, and S. Kato, "Towards real-time scheduling of virtual machines without kernel modifications," in *Proceedings of the 16th IEEE International Conference on Emerging Technology and Factory Automation (ETFA'11), Work-in-Progress (WiP) session*, September 2011, pp. 1–4.

[21] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari, "AQuoSA-adaptive quality of service architecture," *Softw. Pract. Exper.*, vol. 39, no. 1, pp. 1–31, January 2009.

[22] E. Bini, G. Buttazzo, J. Eker, S. Schorr, R. Guerra, G. Fohler, K.-E. Årzen, V. Romero, and C. Scordino, "Resource management on multicore systems: The ACTORS approach," *Micro, IEEE*, vol. 31, no. 3, pp. 72–81, May-June 2011.

[23] D. Faggioli, M. Trimarchi, F. Checconi, M. Bertogna, and A. Mancina, "An implementation of the earliest deadline first algorithm in Linux," in *Proceedings of the ACM symposium on Applied Computing (SAC'09)*, March 2009, pp. 1984–1989.

[24] S. Kato, R. Rajkumar, and Y. Ishikawa, "AIRS: Supporting interactive real-time applications on multicore platforms," in *Proceedings of the 22nd Euromicro Conference on Real-Time Systems (ECRTS'10)*, July 2010, pp. 47–56.

[25] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Proceedings of the 24th IEEE Real-Time Systems Symposium, (RTSS'03)*, December 2003, pp. 2–13.

[26] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS'98)*, December 1998, pp. 4–13.

[27] M. Behnam, I. Shin, T. Nolte, and M. Nolin, "Sirap: a synchronization protocol for hierarchical resource sharing in real-time open systems," in *Proceedings of the 7th ACM & IEEE international conference on Embedded software (EMSOFT'07)*, 2007, pp. 279–288.

[28] R. Davis and A. Burns, "Resource sharing in hierarchical fixed priority pre-emptive systems," in *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, December 2006, pp. 257–270.

A Comparison of Scheduling Latency in Linux, PREEMPT_RT, and LITMUS$^{\mathrm{RT}}$

Felipe Cerqueira Björn B. Brandenburg
Max Planck Institute for Software Systems (MPI-SWS)

Abstract

Scheduling latency under Linux and its principal real-time variant, the PREEMPT_RT patch, are typically measured using cyclictest*, a tracing tool that treats the kernel as a black box and directly reports scheduling latency. LITMUS$^{\mathrm{RT}}$, a real-time extension of Linux focused on algorithmic improvements, is typically evaluated using* Feather-Trace*, a fined-grained tracing mechanism that produces a comprehensive overhead profile suitable for overhead-aware schedulability analysis. This difference in tracing tools and output has to date prevented a direct comparison. This paper reports on a port of* cyclictest *to LITMUS$^{\mathrm{RT}}$ and a case study comparing scheduling latency on a 16-core Intel platform. The main conclusions are:* ***(i)*** *LITMUS$^{\mathrm{RT}}$ introduces only minor overhead itself, but* ***(ii)*** *it also inherits mainline Linux's severe limitations in the presence of I/O-bound background tasks.*

1 Introduction

Real-time tasks are usually activated in response to external events (*e.g.*, when a sensor triggers) or by periodic timer expirations (*e.g.*, once every millisecond). At the kernel level, both types of activations require the following sequence of steps to be carried out:

1. the processor transfers control to an interrupt handler to react to the device or timer interrupt;
2. the interrupt handler identifies the task waiting for the event and resumes it, which causes the task to be added to the ready queue;
3. the scheduler is invoked to check whether the resumed real-time task should be scheduled immediately (and, if so, on which processor); and finally,
4. if the resumed real-time task has higher priority than the currently executing task, then it is dispatched, which requires a context switch.

In theory, the highest-priority real-time task should be scheduled *immediately* when its activating event occurs, but in practice, Step 1 is delayed if interrupts are temporarily masked by critical sections in the kernel,[1] Steps 2–4 are delayed by cache misses, contention for memory bandwidth, and (in multiprocessor systems) lock contention, Step 3 is further delayed if preemptions are temporarily disabled by critical sections in the kernel,[2] and Step 4 generally causes a TLB flush (on platforms without a tagged TLB), which causes additional delays. Thus, even for the highest-priority task, there is always a delay between the activating event and the instant when the task starts executing. This delay, called *scheduling latency*, affects the response times of all tasks and imposes a lower bound on the deadlines that can be supported by the system. Therefore, it is essential to consider scheduling latency when determining whether a system can provide the desired temporal guarantees. The focus of this paper is an empirical evaluation of scheduling latency in Linux and two of its real-time variants, PREEMPT_RT and LITMUS$^{\mathrm{RT}}$, using *cyclictest*, a latency benchmark.

PREEMPT_RT and *cyclictest*. The importance of scheduling latency has made it a standard metric for the evaluation of real-time operating systems in general, and Linux and its real-time extensions in particular. Concerning the latter, the PREEMPT_RT patch—the *de facto* standard real-time variant of Linux—specifically aims to improve scheduling latency by reducing the number and the length of critical sections in the kernel that mask interrupts or disable preemptions [21, 27]. The efficacy of these changes is commonly quantified using *cyclictest*, a scheduling latency benchmark originally created by Thomas Gleixner and currently maintained by Clark Williams [2].

A key feature of *cyclictest* is that it is easy to use, and as a result it has been widely adopted as the universal benchmark of real-time performance under Linux. For instance, it has been applied to evaluate different hardware and software platforms (*e.g.*, [15, 25, 26, 29]) as well as various aspects of the kernel (*e.g.*, [16, 22, 24, 31]); the *cyclictest* approach has even been extended to continuously monitor the scheduling latency of real applications in a production environment [20]. In short, low scheduling latency as reported by *cyclictest* can be considered the gold standard of real-time performance in the Linux real-time community.

LITMUS$^{\mathrm{RT}}$. In contrast to the PREEMPT_RT project, which has been primarily driven by industry concerns, the **Li**nux **T**estbed for **Mu**ltiprocessor **S**cheduling in **R**eal-**T**ime Systems [1, 10, 13, 14] is a primarily algorithms-oriented real-time extension of Linux. While PREEMPT_RT aggressively optimizes scheduling latency, LITMUS$^{\mathrm{RT}}$ facilitates the implementation and evaluation of novel scheduling and locking policies, which it does by introducing a scheduler

[1] In Linux, via the *local_irq_disable()* interface.

[2] In Linux, via the *preempt_disable()* interface.

plugin interface. That is, the PREEMPT_RT patch reengineers core parts of the kernel to avoid delaying Steps 1 and 3, whereas LITMUS$^{\mathrm{RT}}$ primarily modularizes the scheduling logic that is invoked in Step 3, and leaves other aspects of the kernel unchanged.

LITMUS$^{\mathrm{RT}}$ has served its purpose well and has enabled a number of studies exploring the tradeoff between system overheads and analytical temporal guarantees across a diverse range of scheduling approaches (*e.g.*, see [5, 7, 14, 19, 23]; a full list is provided online [1]). The key component of LITMUS$^{\mathrm{RT}}$ that enables such studies is Feather-Trace [11], a light-weight event tracing toolkit for x86 platforms that is used to collect fine-grained measurements of kernel overheads. With Feather-Trace, it is possible to extract detailed overhead profiles, which can then be incorporated into overhead-aware schedulability analysis to formally validate timing requirements under consideration of system overheads (this process is discussed in detail in [10, Ch. 4]).

The overhead characteristics of LITMUS$^{\mathrm{RT}}$'s scheduling policies are well documented and have been evaluated in detail in prior work [5, 10]. However, because *cyclictest* cannot be used (without modifications) to evaluate LITMUS$^{\mathrm{RT}}$ (see Sec. 2), and because Feather-Trace and *cyclictest* produce fundamentally different outputs (see Sec. 4), to date it has unfortunately not been possible to directly compare LITMUS$^{\mathrm{RT}}$ with Linux and the PREEMPT_RT patch.

In this paper, we present a comprehensive experimental evaluation of scheduling latency under LITMUS$^{\mathrm{RT}}$ based on data obtained with a ported version of *cyclictest*. By comparing LITMUS$^{\mathrm{RT}}$ against stock Linux and PREEMPT_RT, we seek to address two questions concerning the real-time capabilities of the current LITMUS$^{\mathrm{RT}}$ implementation:

1. Does the LITMUS$^{\mathrm{RT}}$ scheduling framework introduce a significant overhead in terms of scheduling latency? And:
2. To what extent is LITMUS$^{\mathrm{RT}}$ affected by high scheduling latencies due to not (yet) incorporating the improvements of the PREEMPT_RT patch?

To answer the first question, we compared LITMUS$^{\mathrm{RT}}$ against the underlying Linux version; to answer the second question, we compared LITMUS$^{\mathrm{RT}}$ against the latest stable version of the PREEMPT_RT patch.

The rest of the paper is organized as follows. Sec. 2 reviews how *cyclictest* operates and describes a faithful port of *cyclictest* to LITMUS$^{\mathrm{RT}}$. In Sec. 3, we present our experimental setup and discuss our results. In Sec. 4, we compare *cyclictest* with Feather-Trace and remark on some of the advantages and limitations of the *cyclictest* benchmark. In Sec. 5, we conclude and mention future work directions.

2 Porting *cyclictest* to LITMUS$^{\mathrm{RT}}$

While *cyclictest* is a remarkably versatile utility, it cannot be applied to LITMUS$^{\mathrm{RT}}$ "out of the box", since LITMUS$^{\mathrm{RT}}$ provides its own, non-standard system call and userspace API, which must be used to configure a real-time task's scheduling parameters. In this section, we discuss the (few) changes that were required to port *cyclictest* to LITMUS$^{\mathrm{RT}}$.

Since the validity of the measurements depends on a correct and unbiased application of *cyclictest*, we begin with a review of *cyclictest* in Sec. 2.1 and explain how its scheduling parameters are configured under stock Linux and PREEMPT_RT in Sec. 2.2. In Sec. 2.3, we review LITMUS$^{\mathrm{RT}}$'s task model and show how *cyclictest* was mapped to it. Finally, Sec. 2.4 briefly discusses a simple, but unexpected problem with the resolution of Linux's one-shot timers that we encountered during the experiments.

2.1 An Overview of *cyclictest*

The execution of *cyclictest* can be divided into three phases. During the initialization phase, the program creates a configurable number of threads (according to the specified parameters), which are then admitted as real-time tasks. The processor affinity mask is also set, which enables migration to be restricted. After that, each thread starts a periodic (*i.e.*, cyclic) execution phase, during which *cyclictest* executes the main measurement loop. An iteration (*i.e.*, one test cycle) starts when the thread's associated one-shot timer expires.[3] Once the thread resumes, a sample of scheduling latency is recorded as the difference between the current time and the instant when the timer should have fired. The timer is then rearmed to start a new iteration and the thread suspends. After a configurable duration, the thread is demoted to best-effort status and exits, and the recorded scheduling latency samples are written to disk.

The cyclic phase, during which the measurements are collected, uses only standard userspace libraries (for example, POSIX APIs to set up timers, synchronize threads, *etc.*) and does not rely on scheduler-specific functionality. Since LITMUS$^{\mathrm{RT}}$ maintains compatibility with most userspace APIs, only the code pertaining to task admission and exit must be adapted, *i.e.*, it suffices to replace *sched_setscheduler()* system calls with LITMUS$^{\mathrm{RT}}$'s library functions for task admission. Importantly, *cyclictest*'s core measurement loop does not have to be changed, which helps to avoid the inadvertent introduction of any bias.

The required modifications, which amount to only 18 lines of new code, are discussed in Sec. 2.3 below. To illustrate how LITMUS$^{\mathrm{RT}}$'s interface differs from stock Linux's interface, we first review how *cyclictest* is configured as a real-time task in Linux (either with or without the PREEMPT_RT patch).

2.2 Setting Task Parameters under Linux

In accordance with the POSIX standard, Linux implements a fixed-priority real-time scheduler (with 99 distinct priority levels). Tasks are dispatched in order of decreasing priority,

[3] *cyclictest* supports a large number of options and can be configured to use different timer APIs. We focus herein on the tool's default behavior.

and ties in priority are broken according to one of two policies: under the SCHED_RR policy, tasks of equal priority alternate using a simple round-robin scheme, and, under the SCHED_FIFO policy, multiple tasks with the same priority are simply scheduled in FIFO order with respect to the time at which they were enqueued into the ready queue. By default, tasks are allowed to migrate among all processors in Linux, but it is possible to restrict migrations using processor affinity masks, and *cyclictest* optionally does so.

Fig. 1 summarizes the relevant code from *cyclictest* that is used to admit a thread as a SCHED_FIFO real-time task under Linux. First, the thread defines the CPU affinity mask, which is assigned with *pthread_setaffinity_np()* (lines 6–13 in Fig. 1). To attain real-time priority, the thread calls the *sched_setscheduler()* system call with the desired scheduling policy and priority as parameters (lines 17-20). Finally, after the measurement phase, the thread transitions back to non-real-time status by reverting to the SCHED_OTHER best-effort policy (lines 24-25).

In the experiments discussed in Sec. 3, *cyclictest* was configured to spawn one thread per core and to use the SCHED_FIFO policy with the maximum priority of 99. Further, processor affinity masks were assigned to fix each measurement thread to a dedicated core. Since there is only one task per core, the choice of tie-breaking policy is irrelevant.

Executing the initialization sequence depicted in Fig. 1 under LITMUS$^{\mathrm{RT}}$ would *not* result in an error (LITMUS$^{\mathrm{RT}}$ does not disable SCHED_FIFO); however, it would also not achieve the desired effect because, for historical reasons, real-time tasks must use a different API (which also allows specifying more explicit and detailed parameters) to attain real-time status in LITMUS$^{\mathrm{RT}}$. We thus adapted *cyclictest* to use LITMUS$^{\mathrm{RT}}$'s API to create an analogous setup.

2.3 Setting Task Parameters under LITMUS$^{\mathrm{RT}}$

LITMUS$^{\mathrm{RT}}$ implements the *sporadic task model* [28], in which real-time tasks are modeled as a sequence of recurrent jobs and defined by a tuple $T_i = (e_i, d_i, p_i)$, where e_i denotes the *worst-case execution time* (WCET) of a single job, d_i the *relative deadline*, and p_i the *minimum inter-arrival time* (or *period*). Under LITMUS$^{\mathrm{RT}}$'s event-driven scheduling policies, the parameter e_i is optional and used only for budget enforcement (if enabled). The parameter d_i, however, is required for scheduling plugins based on the *earliest-deadline first* (EDF) policy, and the parameter p_i is always required to correctly identify individual jobs. In LITMUS$^{\mathrm{RT}}$, all task parameters are expressed in nanosecond granularity since this is the granularity internally used by the kernel.

As mentioned in Sec. 2.1, each thread in *cyclictest* executes in a loop, alternating between resuming, measuring latency, and suspending. The wake-up timer is armed periodically, according to a configurable interval I (in microseconds) defined as a parameter.[4] *cyclictest*'s periodic pattern

```
1  struct thread_param *par;
2
3  /* par contains the cyclictest configuration
4   * and thread parameters */
5
6  if (par->cpu != -1) {
7      CPU_ZERO(&mask);
8      CPU_SET(par->cpu, &mask);
9      thread = pthread_self();
10     if(pthread_setaffinity_np(thread,
11             sizeof(mask), &mask) == -1)
12         warn("Could not set CPU affinity");
13 }
14
15 /* ... */
16
17 memset(&schedp, 0, sizeof(schedp));
18 schedp.sched_priority = par->prio;
19 if (setscheduler(0, par->policy, &schedp))
20     fatal("Failed to set priority\n");
21
22 /* measurement phase */
23
24 schedp.sched_priority = 0;
25 sched_setscheduler(0, SCHED_OTHER, &schedp);
```

Figure 1: Task admission in Linux (original *cyclictest*).

of execution exactly matches the assumptions underlying the sporadic task model and can thus be trivially expressed with parameters $d_i = p_i = I$. To avoid having to estimate the per-job (*i.e.*, per-iteration) execution cost of *cyclictest*, we set e_i to a dummy value of 1 ns and disable budget enforcement, so that each thread can always execute without being throttled by LITMUS$^{\mathrm{RT}}$.

The code necessary to realize admission of a *cyclictest* thread as a real-time task under LITMUS$^{\mathrm{RT}}$ is shown in Fig. 2. The first step is defining the task parameters (in particular, e_i, d_i, and p_i) in lines 3–8 and initializing the userspace interface with *init_rt_thread()*. Budget enforcement is disabled (line 7) and the maximum possible priority is assigned (line 8). LITMUS$^{\mathrm{RT}}$'s fixed-priority plugin currently supports 512 distinct priorities; the *priority* field is ignored by EDF-based plugins.

Next, if a *partitioned* scheduler is used, a scheduling approach where each task is statically assigned to a core, the task must specify its assigned processor in the *rt_task* structure (line 13) and perform this migration (line 14), which is accomplished by calling *be_migrate_to()*.[5] Otherwise, if a *global* scheduler is used, a scheduling approach where tasks may migrate freely, a processor assignment is not required (and ignored by the kernel if provided). The task parame-

[4] In fact, *cyclictest* allows two parameters: i, the timer interval of the first thread, and d, an increment which is added to the interval of each consecutive thread. For example, if $i = 1000$ and $d = 100$, *cyclictest* launches threads with intervals $I \in \{1000, 1100, 1200, \ldots\}$ μs. For simplicity, we assume a single interval I. By default, and as employed in our experiments, *cyclictest* uses $i = 1000\mu s$ and $d = 500\mu s$.

[5] The function *be_migrate_to()* is currently implemented as a wrapper around Linux's processor affinity mask API, but could be extended to incorporate LITMUS$^{\mathrm{RT}}$-specific functionality in the future. The "be_" prefix stems from the fact that it may be called only by best-effort tasks.

```
1  struct rt_task rtt; /* LITMUS^RT API */
2
3  init_rt_task_param(&rtt);
4  rtt.exec_cost = 1;
5  rtt.period = par->interval * 1000;
6  rtt.relative_deadline = par->interval * 1000;
7  rtt.budget_policy = NO_ENFORCEMENT;
8  rtt.priority = LITMUS_HIGHEST_PRIORITY;
9
10 init_rt_thread();
11
12 if (par->cpu != -1) {
13     rtt.cpu = par->cpu;
14     if (be_migrate_to(par->cpu) < 0)
15         fatal("Could not set CPU affinity");
16 }
17
18 if (set_rt_task_param(gettid(), &rtt) < 0)
19     fatal("Failed to set rt_param.");
20
21 if (task_mode(LITMUS_RT_TASK) != 0)
22     fatal("failed to change task mode.\n");
23
24 /* measurement phase */
25
26 task_mode(BACKGROUND_TASK);
```

Figure 2: Task admission in LITMUS$^{\mathrm{RT}}$ (modified *cyclictest*).

ters are stored in the process control block (and validated by the kernel) with the system call *set_rt_task_param()* in line 18. Finally, *task_mode(LITMUS_RT_TASK)* is called in line 21, which causes the thread to be admitted to the set of real-time tasks. After the measurement phase, *task_mode(BACKGROUND_TASK)* is used to give up real-time privileges and return to SCHED_OTHER status.

With these changes in place, *cyclictest* is provisioned under LITMUS$^{\mathrm{RT}}$ equivalently to the configuration executed under Linux (both with and without the PREEMPT_RT patch). This ensures a fair and unbiased comparison.

2.4 The Effect of Timer Resolution on *nanosleep()*

Despite our efforts to establish a level playing field, we initially observed unexpectedly large scheduling latencies under LITMUS$^{\mathrm{RT}}$ in comparison with SCHED_FIFO, even in an otherwise idle system. This was eventually tracked down to a systematic $50\mu s$ delay of timer interrupts, which was caused by the fact that Linux subjects *nanosleep()* system calls to timer coalescing to reduce the frequency of wake-ups. As this feature is undesirable for real-time tasks, it is circumvented for SCHED_FIFO and SCHED_RR tasks. A similar exception was introduced for LITMUS$^{\mathrm{RT}}$, which resolved the discrepancy in expected and observed latencies.

It should be noted that LITMUS$^{\mathrm{RT}}$ provides its own API for periodic job activations, and that this API has never been subject to timer coalescing, as it does not use the *nanosleep* functionality. The issue arose in our experiments only because we chose to not modify the way in which *cyclictest* triggers its periodic activations (since we did not want to change the actual measuring code in any way).

3 Experiments

We conducted experiments with *cyclictest* to evaluate the scheduling latency experienced by real-time tasks under LITMUS$^{\mathrm{RT}}$ in comparison with an unmodified Linux kernel. The results were further compared with latencies as observed under Linux with the PREEMPT_RT patch. Our testing environment consisted of a 16-core Intel Xeon X7550 2.0GHz platform with 1 TiB RAM. Features that lead to unpredictability such as hardware multithreading, frequency scaling, and deep sleep states were disabled for all kernels, along with every kernel configuration option associated with tracing or debugging. Background services such as *cron* were disabled to the extent possible, with the notable exception of the remote login server *sshd* for obvious reasons.

We used *cyclictest* to sample scheduling latency under six different kernel and scheduling policy combinations. Under LITMUS$^{\mathrm{RT}}$, which is currently still based on Linux 3.0, we focused our analysis on a subset of the event-driven scheduler plugins: the *partitioned EDF* plugin (PSN-EDF), the *global EDF* plugin (GSN-EDF), and the *partitioned fixed-priority* plugin (P-FP).[6] We did not evaluate LITMUS$^{\mathrm{RT}}$'s Pfair [4] plugin, which implements the PD^2 [3] scheduling policy, since PD^2 is a quantum-driven policy and hence not optimized to achieve low scheduling latency.[7]

We further evaluated SCHED_FIFO in three Linux kernels: in Linux 3.0 (the stock version, without any patches), Linux 3.8.13 (again, without patches), and Linux 3.8.13 with the PREEMPT_RT patch. Though we compare scheduling latencies of two different underlying versions of Linux, both considered versions exhibit a similar latency profile (for which we provide supporting data in Sec. 3.5), so our comparison of LITMUS$^{\mathrm{RT}}$ and PREEMPT_RT is valid despite the difference in base kernel versions.

For each scheduling policy and kernel, we varied the set of background tasks to assess scheduling latency in three scenarios: **(i)** a system with no background workload, **(ii)** a system with a cache-intensive, CPU-bound background workload, and **(iii)** a system with an interrupt-intensive, I/O-bound background workload. Of these three scenarios, scenario (i) is clearly the best-case scenario, whereas scenario (iii) puts severe stress onto the system. Scenario (ii) matches the background workload that has been used in prior LITMUS$^{\mathrm{RT}}$ studies (*e.g.*, see [5, 6, 10, 12]).

cyclictest was executed with standard SMP parameters (one thread per processor), with periods in the range of $I \in \{1000\mu s, 1500\mu s, 2000\mu s, \ldots\}$ and the *-m* flag enabled, which locks memory pages with *mlockall()* to prevent page faults. The result of each execution is a histogram of observed scheduling latencies, where the x-axis represents the

[6] The "S" and "N" in the plugin names PSN-EDF and GSN-EDF refer to support for predictable suspensions and non-preemptive sections; see [8, 10]. These algorithmic details are irrelevant in the context of this paper.

[7] Under a quantum-driven scheduler, worst-case scheduling latencies cannot be lower than the quantum length, which in the current version of LITMUS$^{\mathrm{RT}}$ is tied to Linux's scheduler tick and thus is at least $1ms$.

measured delay and the y-axis the absolute frequency of the corresponding value plotted on a log scale. Samples were grouped in buckets of size 1 μs. Each test ran for 20 minutes, generating around 5.85 million samples per configuration.

The outcome of the experiments is depicted in Figs. 3–7 and analyzed in the following sections. In Secs. 3.1–3.3, we first discuss the differences and similarities in scheduling latency incurred under LITMUS$^{\mathrm{RT}}$'s P-FP plugin, Linux 3.0, and Linux 3.8.13 (both with and without the PREEMPT_RT patch), and then in Sec. 3.4 we compare the results of the three considered LITMUS$^{\mathrm{RT}}$ scheduler plugins with each other. Finally, Sec. 3.5 compares Linux 3.0 and Linux 3.8.13.

3.1 Idle System

As a first step, we evaluated the latency of the system without background tasks under P-FP and PREEMPT_RT, both running on Linux 3.0, and Linux 3.8.13 with and without the PREEMPT_RT patch. An idle system represents a best-case scenario as non-timer-related interrupts are rare, because kernel code and data is likely to remain cached between scheduler activations, and since code segments within the kernel that disable interrupts have only few inputs to process.

As can be seen in Fig. 3, the maximum observed scheduling latency was below $20\mu s$ under each of the four schedulers (insets (a)-(d)), and even below $12\mu s$ under the PREEMPT_RT configuration. The maximum observed scheduling latency under stock Linux 3.8.13 is somewhat higher than under both LITMUS$^{\mathrm{RT}}$ and stock Linux 3.0. As high-latency samples occur only rarely, we ascribe this difference to random chance; with a longer sampling duration, latencies of this magnitude would likely be detected under LITMUS$^{\mathrm{RT}}$ and Linux 3.0, too. All three Linux variants exhibited comparable average and median latencies, close to $2.8\mu s$ and $2.6\mu s$, respectively. Scheduling latencies under LITMUS$^{\mathrm{RT}}$'s P-FP scheduler were slightly higher with a median and average of roughly $3.4\mu s$ and $3.1\mu s$, respectively.

Considering the overall shape of the histograms, all four schedulers exhibit similar trends. Slight differences are visible in PREEMPT_RT's histogram, which resembles the results for the other Linux versions in the initial 0-5μs interval, but lacks higher latency samples. This suggests that PREEMPT_RT avoids outliers even in best-case scenarios.

Overall, as there are not many sources of latency in the absence of a background workload (such as the disabling of interrupts and contention at the hardware level), the observed scheduling latencies are suitably low under each tested kernel. While the LITMUS$^{\mathrm{RT}}$ patch does appear to increase latencies slightly on average, it does not substantially alter the underlying latency profile of Linux as other factors dominate. From this, we conclude that the LITMUS$^{\mathrm{RT}}$ framework does not inherently introduce undue complexity and overheads.

Next, we discuss the effects of increased processor and cache contention.

3.2 CPU-bound Background Workload

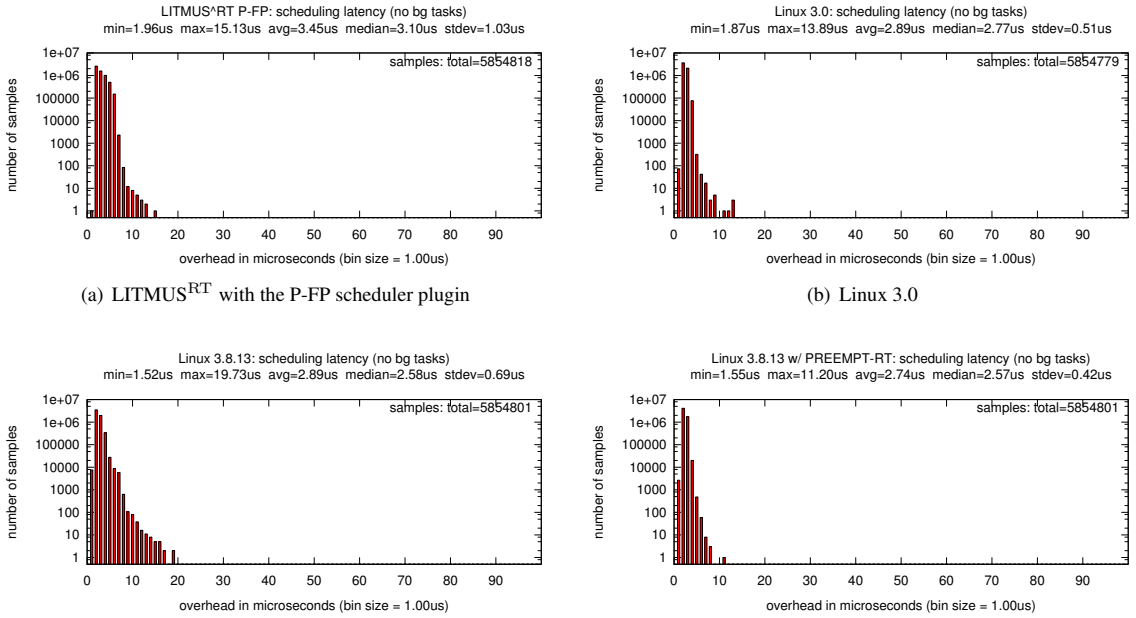
Kernel overheads in general, and thus also the scheduling latency experienced by real-time tasks, vary depending on the contention for limited hardware resources such as memory bandwidth and shared caches. A cache-intensive, CPU-bound background workload can thus be expected to result in worsened latencies, as cache misses and contention in the memory hierarchy are more likely to occur. To evaluate such a scenario, we executed *cyclictest* along with CPU-bound background tasks. For each processor, we instantiated a task consisting of a tight loop accessing random memory locations to generate cache misses and contention. Each such task was configured to have a working set of 20 MiB, to exceed the size of each processor's exclusive L2 cache, which has a capacity of 18 MiB. This causes significant cache pressure. Fig. 4 shows the recorded latencies for PSN-EDF, Linux 3.0, and Linux 3.8.13 (with and without PREEMPT_RT).

A pairwise comparison between the same policies in Fig. 3 and Fig. 4 illustrates how scheduling latencies are impacted by cache and memory issues. As expected, the average and maximum latencies under P-FP, Linux 3.0 and stock Linux 3.8.13, depicted in insets (a), (b), and (c), respectively, increased noticeably. While average and median scheduling latencies increased by only one to two microseconds in absolute terms, the increase in relative terms is significantly higher, exceeding 45 percent in the case of average latency under both LITMUS$^{\mathrm{RT}}$ and Linux 3.0. Most significant is the increase in observed maximum latency, which reached roughly $48\mu s$, $73\mu s$, and $65\mu s$ under LITMUS$^{\mathrm{RT}}$, Linux 3.0, and Linux 3.8.13, respectively. This shows that even a modest, compute-only background workload can significantly impact observable latencies in mainline Linux.

Interestingly, the advantages of PREEMPT_RT (inset (d)) become more apparent in this scenario: with the PREEMPT_RT patch, Linux was able to maintain low latencies despite the increased load, both in terms of average as well as maximum latency ($3.4\mu s$ and $17.42\mu s$, respectively). The corresponding stock kernel incurred significantly worse latencies (see the longer tail in Fig. 4(c)).

Comparing the distribution of samples, it can be seen that the observed scheduling latency under LITMUS$^{\mathrm{RT}}$'s P-FP plugin follows a slightly different pattern than either Linux 3.0 or Linux 3.8.13. In particular, LITMUS$^{\mathrm{RT}}$'s distribution appears to be "wider" and "heavier," with a less rapid decrease in the frequency of samples in the range of $1\mu s$–$40\mu s$. This explains LITMUS$^{\mathrm{RT}}$'s slightly higher average and median scheduling latencies, which are about $1\mu s$ higher than under either Linux 3.0 or Linux 3.8.13. However, note that LITMUS$^{\mathrm{RT}}$, Linux 3.0, and Linux 3.8.13 are all similarly subject to long tails, which indicates that the observed maximum latencies are caused by factors unrelated to LITMUS$^{\mathrm{RT}}$ (*i.e.*, they are caused by issues in the underlying Linux kernel, which the PREEMPT_RT patch addresses).

Nonetheless, the histograms reveal that, in the average case, LITMUS$^{\mathrm{RT}}$ adds measurable (but not excessive) ad-



(a) LITMUS$^{\mathrm{RT}}$ with the P-FP scheduler plugin

(b) Linux 3.0

(c) Linux 3.8.13 without the PREEMPT_RT patch

(d) Linux 3.8.13 with the PREEMPT_RT patch

Figure 3: Histograms of observed scheduling latency in an otherwise idle system.

ditional overhead. We suspect two primary sources for this additional overhead. First, LITMUS$^{\mathrm{RT}}$'s scheduling path needs to acquire (at least) one additional spin lock compared to stock Linux, which is especially costly in the presence of high cache and memory-bandwidth contention. This additional spin lock acquisition stems from the fact that LITMUS$^{\mathrm{RT}}$'s scheduling state is not protected by Linux's runqueue locks; however, Linux's runqueue locks must still be acquired prior to invoking LITMUS$^{\mathrm{RT}}$'s scheduling framework. And second, the increased average-case overheads might be due to a lack of low-level optimizations in LITMUS$^{\mathrm{RT}}$ (in comparison with the mature codebase of Linux). Given that LITMUS$^{\mathrm{RT}}$ is primarily a research-oriented project focused on algorithmic real-time scheduling issues, a certain lack of low-level tuning is not surprising.

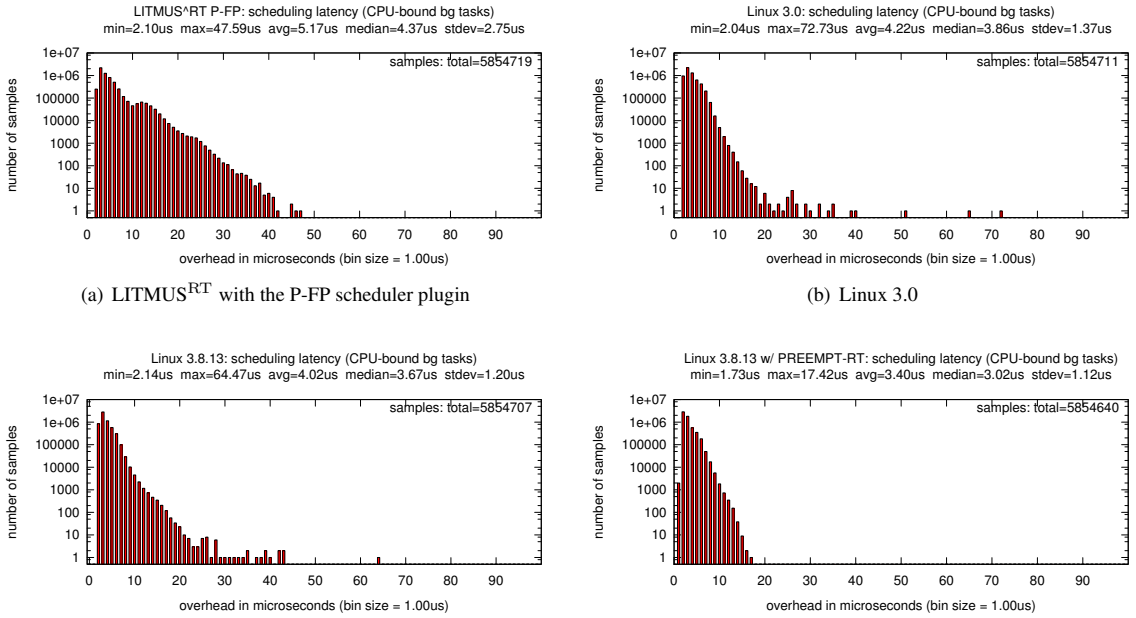
As was already briefly mentioned, the CPU-bound background workload matches the setup that has been used in prior LITMUS$^{\mathrm{RT}}$-based studies (*e.g.*, [5, 6, 10, 12]). As is apparent when comparing Fig. 3(a) with Fig. 4(a), our data confirms that the CPU-bound workload generates sufficient memory and cache pressure to magnify kernel overheads. Conversely, conducting overhead experiments *without* a cache-intensive background workload does not yield an accurate picture of kernel overheads. Next, we discuss the impact of interrupt-intensive background workloads.

3.3 I/O-bound Background Workload

Interrupts are challenging from a latency point of view since interrupt handlers typically disable interrupts temporarily and may carry out significant processing, which both directly affects scheduling latency. It should be noted that Linux has long supported *split interrupt handling* (*e.g.*, see [9]), wherein interrupt handlers are split into a (short) *top half* and a (typically longer) *bottom half*, and only the top half is executed in the (hard) interrupt context, and the bottom half is queued for later processing. However, in stock Linux, bottom halves still effectively have "higher priority" than regular real-time tasks, in the sense that the execution of bottom halves is not under control of the regular SCHED_FIFO process scheduler[8] and thus may negatively affect scheduling latencies. Further, bottom halves may still disable interrupts and preemptions for prolonged times.

Considerable effort has been invested by the developers of the PREEMPT_RT patch to address these very issues. This is accomplished by forcing bottom half processing to take place in kernel threads (which can be scheduled such that they do not delay high-priority real-time tasks), and by identifying and breaking up code segments that disable interrupts and preemptions for prolonged durations. In contrast, since LITMUS$^{\mathrm{RT}}$ is currently based on stock Linux, and since the focus of LITMUS$^{\mathrm{RT}}$ is the exploration and evaluation of new scheduling policies (and *not* the reengineering of the underlying Linux kernel), no such improvements are present in LITMUS$^{\mathrm{RT}}$. A key motivation for our experiments was to determine to which extent LITMUS$^{\mathrm{RT}}$ is penalized by the

[8] Bottom halves are processed by so-called *softirqs*, which in stock Linux are invoked from interrupt and exception return paths.



(a) LITMUS$^{\mathrm{RT}}$ with the P-FP scheduler plugin

(b) Linux 3.0

(c) Linux 3.8.13 without the PREEMPT_RT patch

(d) Linux 3.8.13 with the PREEMPT_RT patch

Figure 4: Histograms of observed scheduling latency in the presence of a CPU-bound background workload.

absence of such improvements.

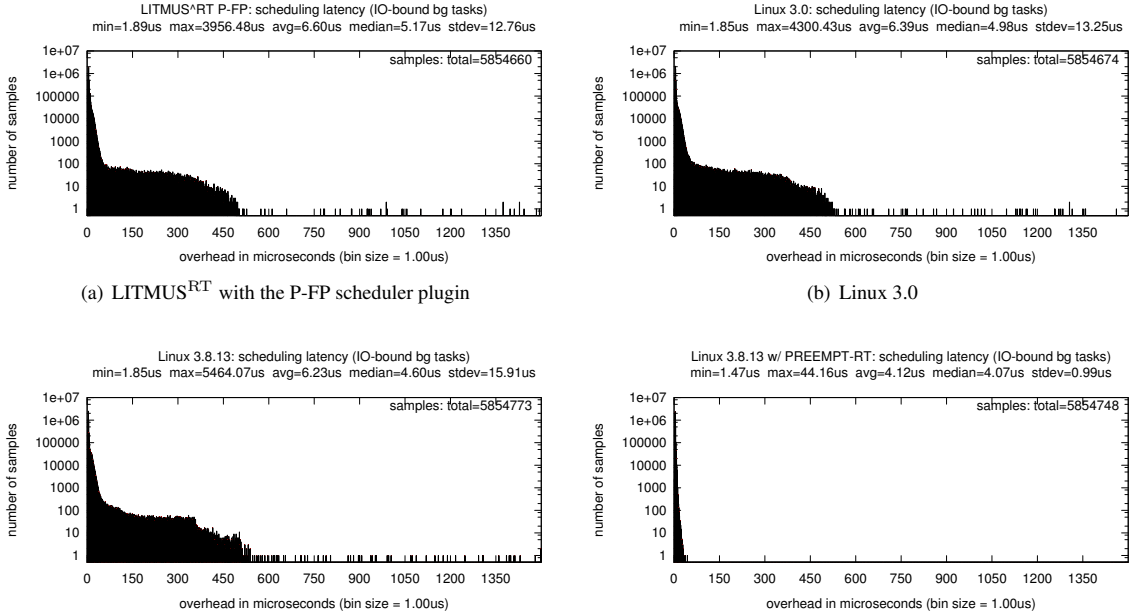
We explored the impact of interrupt-intensive workloads on scheduling latency with I/O-bound background tasks that generate a large number of interrupts, system calls, and scheduler invocations. To simulate such workloads, we used a combination of the following three tools.

1. *hackbench*, a standard stress test for the Linux scheduler [30]. Under the employed default settings, it creates 400 processes that exchange tokens via (local) sockets, thus causing frequent system calls and scheduler invocations (due to blocking reads).
2. *Bonnie++*, a standard file system and hard disk stress test [17]. *Bonnie++* tests file creation, random and sequential file access, and file system metadata access. We configured *Bonnie++* to use *direct I/O*, which circumvents Linux's page cache (and thus results in increased disk activity). This workload results in a large number of system calls, disk interrupts, and scheduler invocations (due to blocking reads and writes).
3. *wget*, a common utility to download files via HTTP. We used *wget* to download a 600 MiB file from a web server on the local network in a loop. The downloaded file was immediately discarded by writing it to */dev/null* to avoid stalling on disk I/O. One such download-and-discard loop was launched for each of the 16 cores. This workload generates a large number of network interrupts as Ethernet packets are received at the maximum rate sustained by the network (with 1 Gib links in our lab).

In combination, these three workloads cause considerable stress for the entire kernel, and can be expected to frequently trigger code paths that inherently have to disable interrupts and preemptions. While the three tools may not reflect any particular real-world application, the chosen combination of stress sources is useful to consider because it approaches a worst-case scenario with regard to background activity. The resulting distributions of observed scheduling latency under P-FP, Linux 3.0, and Linux 3.8.13 with and without the PREEMPT_RT patch are depicted in Fig. 5.

Scheduling latencies are severely affected by the I/O-bound background workload under LITMUS$^{\mathrm{RT}}$, Linux 3.0, and stock Linux 3.8.13 alike. The corresponding histograms, shown in insets (a)–(c) of Fig. 5, respectively, exhibit a long, dense tail. Note that the x-axis in Fig. 5 uses a different scale than Fig. 3 and 4: scheduling latencies in excess of $5ms$ were observed in this scenario, two orders of magnitude worse than in the previous ones. Scheduling latencies in this range clearly limit these kernels to hosting applications that are not particularly latency-sensitive.

In contrast, Linux 3.8.13 with the PREEMPT_RT patch maintained much lower scheduling latencies, in the order of tens of microseconds, despite the stress placed upon the system, which can be seen in Fig. 5(d). Nonetheless, the maximum observed scheduling latency did increase to $44\mu s$, which shows that, even with the PREEMPT_RT patch, non-negligible latencies arise given harsh workloads. However, this maximum was still significantly lower than the maximum latency previously observed under Linux 3.8.13



(a) LITMUS$^{\mathrm{RT}}$ with the P-FP scheduler plugin

(b) Linux 3.0

(c) Linux 3.8.13 without the PREEMPT_RT patch

(d) Linux 3.8.13 with the PREEMPT_RT patch

Figure 5: Histograms of observed scheduling latency in the presence of an I/O-bound background workload.

without the PREEMPT_RT patch in the presence of only CPU-bound workloads, which is apparent when comparing Fig. 4(c) with Fig. 5(d). Remarkably, the average and median scheduling latency under PREEMPT_RT worsened by less than $0.7\mu s$ with the introduction of the I/O-bound workload.

Finally, we also ran two variations of the I/O-bound workload with varying degrees of disk activity. First, we disabled *bonnie++* altogether, which brought down the maximum observed latencies under Linux 3.0, Linux 3.8.13 (without the PREEMPT_RT patch), and LITMUS$^{\mathrm{RT}}$ to around $550\mu s$, which is still too high for practical purposes, but shows that the extreme outliers are caused by disk-related code. And second, we tried launching an instance of *bonnie++* on each core, which brought the disk I/O subsystem to its knees and caused latency spikes in the range of 80–200 *milliseconds* (!) under the three non-PREEMPT_RT kernels. Remarkably, the maximum observed scheduling latency under PREEMPT_RT remained below $50\mu s$ even in this case.

Overall, our experiment asserts the importance of PREEMPT_RT in turning Linux into a viable real-time platform. Given the huge differences in maximum observed latency, LITMUS$^{\mathrm{RT}}$ would be substantially improved if it incorporated PREEMPT_RT. Though this will require considerable engineering effort (both patches modify in part the same code regions), there are no fundamental obstacles to rebasing LITMUS$^{\mathrm{RT}}$ on top of the PREEMPT_RT patch.

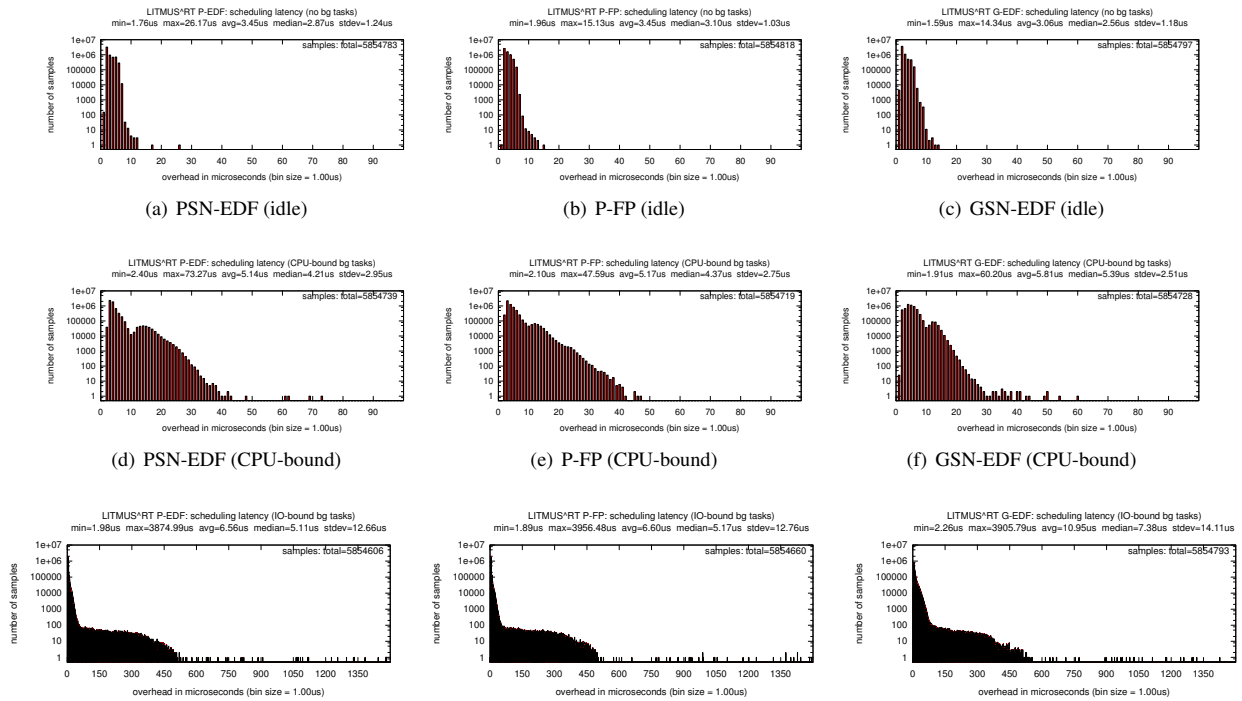
3.4 Scheduling Latency of LITMUS$^{\mathrm{RT}}$ Plugins

In the preceding sections, we have focused on LITMUS$^{\mathrm{RT}}$'s P-FP plugin, since it implements the same scheduling policy as SCHED_FIFO (albeit with a larger number of priorities and support for additional real-time locking protocols) and thus allows for the most direct comparison. We also investigated how scheduling latency varies among the three evaluated LITMUS$^{\mathrm{RT}}$ scheduler plugins. Fig. 6 compares the P-FP, PSN-EDF and GSN-EDF plugins in LITMUS$^{\mathrm{RT}}$, under each of the three considered background workloads.

Comparing insets (g), (h), and (i), it is apparent that the three plugins are equally subject to high scheduling latencies (approaching $4ms$) in the case of the I/O-bound background workload. This is not surprising, since the long tail of high scheduling latencies is caused by the design of the underlying Linux kernel, and thus independent of the choice of plugin.

Further, comparing Fig. 6(a) with Fig. 6(b), and Fig. 6(d) with Fig. 6(e), it is apparent that the PSN-EDF and P-FP plugins yield near-identical scheduling latency distributions, despite the difference in implemented scheduling policy. This, however, is expected since the tests run only one real-time task per processor; the real-time scheduler is hence not stressed and the cost of the scheduling operation is so small compared to other sources of latency that any differences between fixed-priority and EDF scheduling disappear in the noise. Differences emerge only for higher task counts [10].

However, looking at Fig. 6(f) and Fig. 6(i), it is apparent that the scheduling latency is noticeably higher under GSN-



(a) PSN-EDF (idle) (b) P-FP (idle) (c) GSN-EDF (idle)

(d) PSN-EDF (CPU-bound) (e) P-FP (CPU-bound) (f) GSN-EDF (CPU-bound)

(g) PSN-EDF (I/O-bound) (h) P-FP (I/O-bound) (i) GSN-EDF (I/O-bound)

Figure 6: Histograms of observed scheduling latency under LITMUS$^{\mathrm{RT}}$ with the PSN-EDF, P-FP, and GSN-EDF plugins, under each of the three considered background workloads.

EDF in the average case, which is due to its more complex implementation. Issues such as contention caused by coarse-grained locking, extra bookkeeping, and cache-coherence delays when accessing shared structures increase both the median and average observed scheduling latencies.

While this shows that LITMUS$^{\mathrm{RT}}$'s implementation of global scheduling incurs higher overheads, there is little reason to employ global scheduling when the number of tasks does not exceed the number of available cores (which is the case in the considered *cyclictest* setup). If the number of tasks actually exceeds the number of available cores—that is, if the scheduling problem is not entirely trivial—then other factors such as the impact of interference from higher-priority tasks or a need for bounded tardiness [18] can make minor differences in scheduling latency a secondary concern, with only little impact on overall temporal correctness.

3.5 Linux 3.0 vs. Linux 3.8

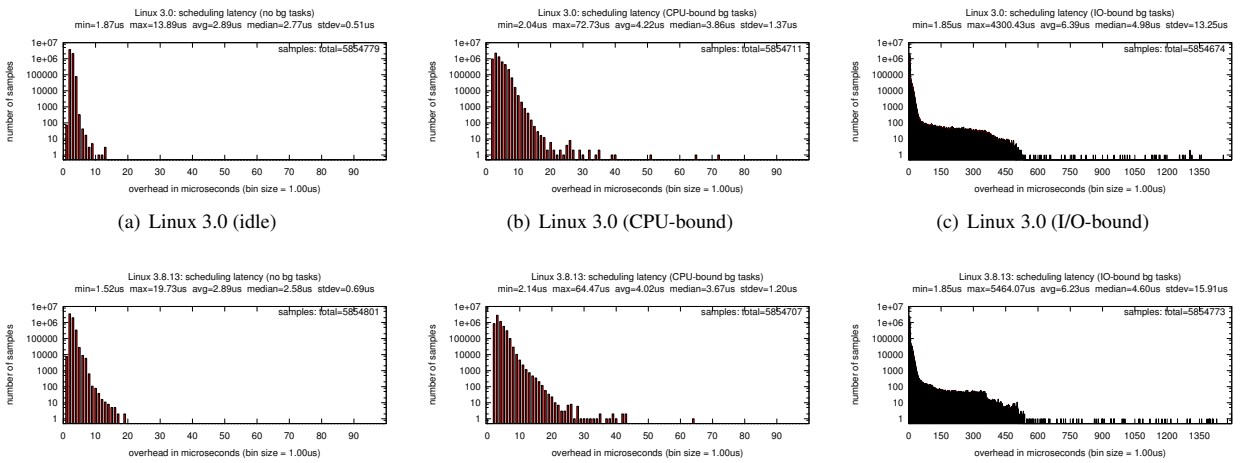
In this paper, we compared the latency of LITMUS$^{\mathrm{RT}}$ and Linux with the PREEMPT_RT patch using the latest versions of each patch, which are based on Linux 3.0 and Linux 3.8.13, respectively. As already discussed in the preceding sections, to verify that comparing the two patches is valid despite the difference in the underlying kernel version, we also measured the scheduling latencies exhibited by the two underlying (unpatched) Linux versions. For ease of comparison, the results are repeated in Fig. 7.

A comparison of inset (a)-(c) with insets (d)-(f) shows that, though the observed maxima vary (for example, from $13.89\mu s$ to $19.73\mu s$ in the scenario without background tasks), the shapes of the distributions are largely similar. Further, there are no substantial differences in the average and median latencies of the two kernel versions. This indicates that no significant improvements concerning latency and preemptivity have been incorporated since Linux 3.0. Therefore, a direct comparison between the LITMUS$^{\mathrm{RT}}$ patch and the PREEMPT_RT patch is valid.

This concludes the discussion of our experimental results. Next, we briefly discuss how the presented *cyclictest* experiments differ from the overhead and latency tracing typically used to evaluate LITMUS$^{\mathrm{RT}}$.

4 Limitations of *cyclictest*

As discussed in Sec. 1, LITMUS$^{\mathrm{RT}}$ is normally evaluated using Feather-Trace, not *cyclictest*. While *cyclictest* is a very useful tool to assess and compare different kernel versions (*e.g.*, it can be used to test whether a proposed patch has a negative impact on scheduling latency), it also has some limitations if used as the sole metric for estimating a system's



(a) Linux 3.0 (idle) (b) Linux 3.0 (CPU-bound) (c) Linux 3.0 (I/O-bound)

(d) Linux 3.8.13 (idle) (e) Linux 3.8.13 (CPU-bound) (f) Linux 3.8.13 (I/O-bound)

Figure 7: Histograms of observed scheduling latency under Linux 3.0 and 3.8.13, under each of the three considered background workloads.

capability to provide temporal guarantees.

The primary advantage of *cyclictest* is that it provides an easy-to-interpret metric that reflects various sources of unpredictability as a single, opaque measure. That is, it treats the kernel and the underlying hardware as a black box and reports the actual cumulative impact of system overheads and hardware capabilities on real-time tasks. For application developers, this is convenient as it requires neither post-tracing analysis nor a detailed understanding of the kernel.

In contrast, Feather-Trace yields a large number of (non-human-readable) event timestamps that require matching, filtering, post-processing, and a statistical evaluation. The resulting overhead profile is primarily intended for integration into schedulability analysis and is less suitable to direct interpretation. However, while *cyclictest* is arguably more convenient, LITMUS$^{\mathrm{RT}}$'s Feather-Trace approach provides a more complete picture since it yields the data required to assess the impact of kernel overheads on tasks other than the highest-priority task, as we explain next.

The main feature of Feather-Trace is that it integrates many tracepoints in the kernel, which can be used to collect fine-grained overheads. By measuring and considering the various sources of delay *individually*, a detailed *analysis* of the worst-case cumulative delay can be carried out.

For example, for a task other than the highest-priority task, the cumulative delay incurred depends on the worst-case scheduling latency *and* the delays due to preemptions by higher-priority tasks, which in turn depends on context-switching overheads, scheduling overheads in the presence of potentially many ready tasks, and so on. With Feather-Trace in LITMUS$^{\mathrm{RT}}$, it is possible to measure all these aspects individually, and then account for them during schedulability analysis (see [10, Ch. 3] for a comprehensive introduction to overhead accounting), such that the observed worst-case overheads are fully reflected in the derived temporal guarantees for *all* tasks (and not just the highest-priority task).

As another example, consider how tasks are resumed under partitioned schedulers such as the P-FP plugin (or SCHED_FIFO with appropriate processor affinity masks). If a real-time task resumes on a remote processor (*i.e.*, any processor other than its assigned partition), an *inter-processor interrupt* (IPI) must be sent to its assigned processor to trigger the scheduler. IPIs are of course not delivered and processed instantaneously in a real system and thus affect scheduling latency *if they arise*. When scheduling *cyclictest* on hardware platforms with processor-local timers (such as local APIC timers in modern x86 systems), however, such IPIs are not required because the interrupt signaling the expiry of *cyclictest*'s one-shot timer is handled locally. If we simply execute *cyclictest* under PSN-EDF, P-FP, or SCHED_FIFO with appropriate processor affinity masks to determine "the worst-case latency," it will never trace the impact of such IPIs, even though an actual real-time application that is triggered by interrupts from devices other than timers (*e.g.*, such as a sensor) would actually be subject to IPI delays. In contrast, in the methodology used to evaluate LITMUS$^{\mathrm{RT}}$ (see [10, Ch. 4]), Feather-Trace is used to measure IPI latencies, which are then correctly accounted for in the schedulability analysis to reflect the worst-case task-activation delay.

In summary, it is impossible to derive how real-time tasks other than the highest-priority task are affected by overheads from *cyclictest*-based experiments, because overhead-aware schedulability analysis is fundamentally required to make temporal guarantees for all tasks. Such an analysis is made possible by Feather-Trace's ability to extract specific overheads. While obtaining measurements in a fine-grained manner is more involved than simply running *cyclictest*, Feather-Trace's fine-grained measurement approach provides a flexi-

bility that is not achievable with coarse-grained approaches such as *cyclictest*. This, of course, does not diminish *cyclictest*'s value as a quick assessment and debugging aid, but it should not be mistaken to provide a general measure of a system's "real-time capability"; it can only show the lack of such capability under certain circumstances—for instance, by exposing scheduling latencies in excess of $5ms$ in the presence of I/O-bound background tasks.

5 Conclusion and Future Work

We presented an empirical evaluation of scheduling latency under LITMUS$^{\mathrm{RT}}$ using *cyclictest*. We ported *cyclictest* to LITMUS$^{\mathrm{RT}}$'s native API and collected samples of scheduling latency under several of its event-driven scheduler plugins, in three system configurations (an idle system, a system with CPU-bound background tasks, and a system with I/O-bound background tasks). For the purpose of comparison, we repeated the same measurements under Linux 3.0, Linux 3.8.13, and Linux 3.8.13 with the PREEMPT_RT patch using the original, unmodified *cyclictest* version.

The results obtained from an idle system and in the presence of CPU-bound background tasks showed that while LITMUS$^{\mathrm{RT}}$ introduces some additional overheads, the difference is minor in absolute terms and manifests only in the average and median scheduling latencies. Importantly, LITMUS$^{\mathrm{RT}}$ was not observed to affect the maximum scheduling latencies negatively, which is due to the fact that other factors in mainline Linux have a much larger impact on worst-case delays. We conclude from these observations that LITMUS$^{\mathrm{RT}}$ does not impose inherently impractical overheads. Further, we believe that the observed minor increase in average and median scheduling latency is not fundamental, but caused by a lack of low-level optimizations that could be rectified with additional engineering effort.

However, our data also documents that LITMUS$^{\mathrm{RT}}$ inherits its mainline Linux's weaknesses in the presence of I/O-bound background tasks. Again, LITMUS$^{\mathrm{RT}}$ did not increase the observed maximum scheduling latency, but the latency profile of the underlying Linux 3.0 kernel renders it unfit for serious (hard) real-time applications. Further, our experiments confirmed that this is still the case with the more recent mainline Linux version 3.8.13. It would thus be highly desirable to combine LITMUS$^{\mathrm{RT}}$'s algorithmic improvements with the increased responsiveness under load achieved by the PREEMPT_RT patch, which remains as future work.

References

[1] The LITMUS$^{\mathrm{RT}}$ project. `http://www.litmus-rt.org`.

[2] Real-time linux wiki. cyclictest - RTwiki. `https://rt.wiki.kernel.org/index.php/Cyclictest`.

[3] J. Anderson and A. Srinivasan. Mixed Pfair/ERfair scheduling of synchronous periodic tasks. In *Proc. of the 13th Euromicro Conference on Real-Time Systems*, pages 76–85. IEEE, 2001.

[4] S. Baruah, N. Cohen, C. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

[5] A. Bastoni, B. Brandenburg, and J. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers. In *Proc. of the 31st Real-Time Systems Symposium*, pages 14–24, 2010.

[6] A. Bastoni, B. Brandenburg, and J. Anderson. Is semi-partitioned scheduling practical? In *Proc. of the 23rd Euromicro Conference on Real-Time Systems*, pages 125–135, 2011.

[7] A. Block. *Adaptive multiprocessor real-time systems*. PhD thesis, University of North Carolina at Chapel Hill, 2008.

[8] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson. A flexible real-time locking protocol for multiprocessors. In *Proc. of the 13th IEEE Conference on Embedded and Real-Time Computing Systems and Applications*, pages 47–57, 2007.

[9] D. Bovet and M. Cesati. *Understanding The Linux Kernel*. O'Reilly & Associates Inc, third edition, 2005.

[10] B. Brandenburg. *Scheduling and locking in multiprocessor real-time operating systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2011.

[11] B. Brandenburg and J. Anderson. Feather-trace: A light-weight event tracing toolkit. In *Proc. of the Workshop on Operating Systems Platforms for Embedded Real-Time applications*, pages 61–70, 2007.

[12] B. Brandenburg and J. Anderson. A comparison of the M-PCP, D-PCP, and FMLP on LITMUS$^{\mathrm{RT}}$. In *Proc. of the 12th Intl. Conference on Principles of Distributed Systems*, pages 105–124, 2008.

[13] B. Brandenburg, A. Block, J. Calandrino, U. Devi, H. Leontyev, and J. Anderson. LITMUS$^{\mathrm{RT}}$: a status report. *9th Real-Time Linux Workshop*, 2007.

[14] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proc. of the 27th IEEE Real-Time Systems Symposium*, pages 111–123, 2006.

[15] G. Chanteperdrix and R. Cochran. The ARM fast context switch extension for Linux. *Real Time Linux Workshop*, 2009.

[16] R. Cochran, C. Marinescu, and C. Riesch. Synchronizing the Linux system time to a PTP hardware clock. In *Proc. of the 2011 Intl. IEEE Symposium on Precision Clock Synchronization for Measurement Control and Communication*, pages 87–92, 2011.

[17] R. Coker. *bonnie++ — program to test hard drive performance*. Linux manual page.

[18] U. Devi. *Soft real-time scheduling on multiprocessors*. PhD thesis, Chapel Hill, NC, USA, 2006.

[19] G. Elliott and J. Anderson. Globally scheduled real-time multiprocessor systems with GPUs. *Real-Time Systems*, 48(1):34–74, 2012.

[20] C. Emde. Long-term monitoring of apparent latency in PREEMPT_RT Linux real-time systems. *12th Real-Time Linux Workshop*, 2010.

[21] L. Fu and R. Schwebel. Real-time linux wiki. RT PREEMPT HOWTO. `https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO`.

[22] L. Henriques. Threaded IRQs on Linux PREEMPT-RT. In *Proc. of the 5th Intl. Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 23–32, 2009.

[23] C. Kenna, J. Herman, B. Brandenburg, A. Mills, and J. Anderson. Soft real-time on multiprocessors: are analysis-based schedulers really worth it? In *Proc. of the 32nd Real-Time Systems Symposium*, pages 93–103, 2011.

[24] J. Kiszka. Towards Linux as a real-time hypervisor. In *Proc. of the 11th Real-Time Linux Workshop*, pages 205–214, 2009.

[25] K. Koolwal. Investigating latency effects of the linux real-time preemption patches (PREEMPT_RT) on AMD's GEODE LX platform. In *Proc. of the 11th Real-Time Linux Workshop*, pages 131–146, 2009.

[26] A. Lackorzynski, J. Danisevskis, J. Nordholz, and M. Peter. Real-time performance of L4Linux. In *Proc. of the 13th Real-Time Linux Workshop*, pages 117–124, 2011.

[27] P. McKenney. A realtime preemption overview. 2005. LWN. http://lwn.net/Articles/146861/.

[28] A. Mok. *Fundamental design problems of distributed systems for the hard-real-time environment*. PhD thesis, 1983.

[29] M. Traut. Real-time CORBA performance on Linux-RT PREEMPT. *9th Real-Time Linux Workshop*, 2007.

[30] C. Williams and D. Sommerseth. *hackbench — scheduler benchmark/stress test*. Linux manual page.

[31] B. Zuo, K. Chen, A. Liang, H. Guan, J. Zhang, R. Ma, and H. Yang. Performance tuning towards a KVM-based low latency virtualization system. In *Proc. of the 2nd Internation Conference on Information Engineering and Computer Science*, pages 1–4. IEEE, 2010.

Towards power-efficient mixed-critical systems

Florian Broekaert, Agnes Fritsch, Laurent San
AAL Advanced Architecture Lab – EDS service
Thales Communications & Security SA (TCS)
Gennevilliers, France
{florian.broekaert, agnes.fritsch,
laurent.san}@thalesgroup.com

Sergey Tverdyshev
R&D Department
SYSGO AG
Klei-Winternheim, Germany
{sergey.tverdyshev}@sysgo.com

***Abstract*—Critical systems performance compliancy and predictability is an utmost priority. However, these systems also face power issues and could benefit of smart power management policies. Standard OS power management policies do not fulfill the requirements of critical systems. In this paper we explore architectures integrating power-management techniques into a RTOS designed for safety and security critical systems. These architectures treat the power efficiency as one of the fine-tuning knobs in the designs of mixed-critical systems.**

Keywords—low-power; power-aware real-time scheduling; safety; security; mixed-critical;

I. INTRODUCTION

Over the last years, power consumption has become a major concern for the worldwide electronics' systems industry. Interest in energy management has grown for reasons such as to cut down power consumption, to maximize autonomy of battery powered devices, and to decrease thermal dissipation. Power management policies are now wide spread on standard operating systems but are dedicated to general purpose applications, and therefore, usually do not fit into critical or real-time systems. To overcome this limitation and to fulfill needs of applications with real-time constrains, TCS has developed a solution consisting in a framework with a portable power management runtime environment based on a low-power scheduler. Evaluations have already been conducted on soft real-time applications and have shown promising results. However, implementation has never been experimented on RTOS with critical applications.

The aim of this paper is to explore integration of reliable power-management policies into a RTOS. To ensure a quick feedback on our ideas in mixed-critical domain we have selected SYSGO PikeOS [5] [6]) real-time operating system, which is widely used in mixed-critical systems.

The paper is organized as follows. Section II exposes general topics in power management. Section III presents a solution developed by TCS based on a low-power scheduler that performs power management at task level. Section IV details safety and security scheduling and its implementation in PikeOS. We conclude in section V by presenting some possible integration perspectives of low-power scheduler concepts through PikeOS extensions.

II. POWER MANAGEMENT FOR CRITICAL SYSTEMS

In this section, general topics related to power management are introduced.

A. HW knobs to reduce energy consumption

In digital systems, power consumption mainly comes from:

- Static power that is related to the leakage current and depends on transistor technology and impacted by temperature. As CMOS technology is scaling down, this problem of static leakage power is becoming worse.
- Dynamic power represented by $P_{dynamuc} \propto \alpha C V^2 f$, [1] with α the activity factor, C the switching capacitance, V the supply voltage and f the frequency.

Modern HW designs provide several means to reduce system overall power consumption, e.g. power/clock gating, multiple power domains. The two most popular techniques taking benefit of these knobs are:

- Dynamic Voltage and Frequency Scaling (DVFS).
  It consists in changing the processor speed by scaling down/up the Voltage/Frequency couple depending on the workload. Reducing frequency implies longer processing time, but the overall impact on dynamic power is drastic.
- Dynamic Power Management (DPM).
  It consists in shutting down power supplies associated to specific areas of the chip or peripherals during period of inactivity. Usually, several levels sleep modes are available. The deeper we go in these low-power modes, the less energy is consumed but the more time is required to go in/out the modes.

From a SW point of view, both techniques abstract the HW resources as a set of modes with characteristics such as performance, power consumption, and switching time between modes. The Advanced Configuration and Power Interface (ACPI [2]) specifies platform independent interfaces for hardware power management to implement such modes.

B. DVFS or DPM?

The common goal of DVFS and DPM is to decrease processor power consumption by reducing time spent in idle mode. Figure 1 details the benefits of DVFS and DPM techniques on energy consumption savings when a task finishes before its deadline and generates slack time.

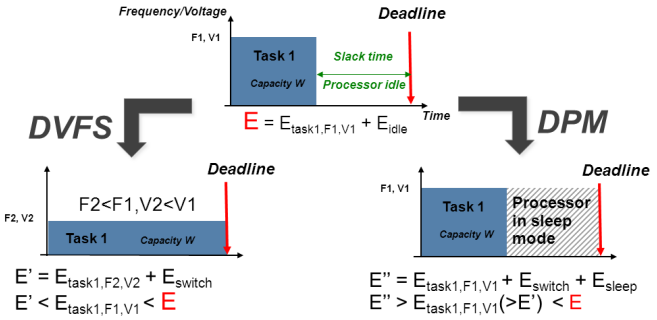


Fig. 1. DVFS and DPM strategies

An advantage of DVFS compared to DPM is the fast switching time between modes (hundreds of microseconds range versus millisecond range due to context saving). DVFS allows keeping processor active (idle mode) even if there is no instruction to be executed whereas for DPM the processor is put into a sleep mode, and thus, requires wake-up mechanism (timer, event) with extra timing overhead. DVFS also enables decreasing the peak power by using a lower voltage/frequency operating point. DPM allows reaching lower power consumption for the processor than the lowest DVFS mode when processor is idle.

The choice between DPM and DVFS depends on the application idle/activity time ratio. The bigger this ratio is, the better the power savings gains with a DPM strategy will be compared to DVFS. However, when DPM is used, the developer needs mechanisms to return from sleep (e.g timer, interruption) and switching time must be taken into account.

C. OS Power Management policies

Power management policies taking benefit of DVFS and DPM can be implemented at:

- Application level:
  Directives are directly inserted in the application and rely on an explicit and proper exploitation by the developer. Problem may occur in a multi-applications context because then decisions should require to be coordinated to avoid possible functional disorder.
- Operating System level:
  Integration is done as service of the OS to scale down power only when the system performance can be relaxed by monitoring HW and SW events. This layer identifies or predicts time intervals where resources are not being used and then adapts power the power modes depending on workload.

However, existing solutions are not suitable for mixed critical systems [9]. On one side, commercial RTOS power management solutions are not commonly available. On the other side, traditional OS power management do trade-offs between power saving gains and the system performance/reactivity. As example, Linux power management relies on the ACPI CPUfreq and CPUidle framework and implements decision heuristics that rely on the past activities. The consequence of this a-posteriori scheme is that future events might be missed if performance level has been lowered too much. Thus, it is wise to disable standard OS power management to guarantee system determinism. These limitations come from the fact that these policies do not take into account some of the key requirements from critical applications, e.g. timing information, deadlines. The key challenge to overcome these limitations is to design a reliable power management layer for real-time systems.

III. Implementation of a Low-Power Scheduler

In this chapter, we describe the solution investigated by TCS to fulfill needs of soft real-time applications and an example of its implementation in Linux.

A. The Low-Power scheduler

Low-power scheduling consists of power-aware task allocation algorithms. This topic has been widely studied in the past. Some strategies dynamically adjust processor modes at task scheduling level to achieve expected performance level required for completing a job while still ensuring compliancy with the application real-time constraints [10]. Figure 2 depicts TCS low-power scheduler solution named "*SCHED_EDF_SMP_DVFS*" (similar to [11]) which follows an intra-task approach and combines a classical Earliest Deadline First (EDF) scheduling with DVFS and slack time recuperation and compares it to a regular EDF scheduling

This scheduler relies on the concept of task's Actual Execution Time (AET). The task's AET is monitored and compared to the Worst Case Execution Times (WCET). The observed slack time is used as an extra time for the next scheduled task. In order to reduce power consumption, the scheduler minimizes application idle time by spreading tasks over as many active cores as possible within the SMP and selects the lowest voltage/frequency mode still complying with the WCET.

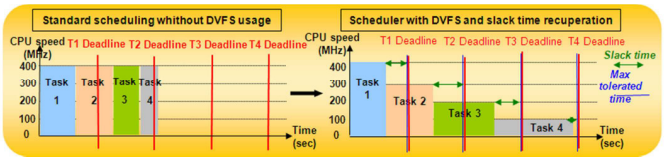


Fig. 2. Principle of DVFS scheduling on a monocore system

Energy savings results depend on how often and how long idle periods are. These periods may be large if WCET differs widely from the AET and are, thus, very application-specific. To reach an efficient usage of the platform resources, the most recent task scheduling techniques propose a mixed design-time exploration and run-time approach, where the power-performance trade-off is explored at the system level [3] [4].

To integrate such kind of DVFS/DPM algorithms into common OS such as Linux, TCS developed a framework described in the next section.

B. The Low-Power framework

Few works describe the integration of low-power algorithms for real-time systems into OS. To tackle drawbacks of standard OS power management policies, TCS started to develop a workflow (patent filed), named "*lp_framework*", to

enable application with timings constraints to benefit from a reliable strategy. A solution easily portable on various OS and HW platforms has been proposed. Objective of such infrastructure is to be able to host low-power scheduling algorithms (e.g. *SCHED_EDF_SMP_DVFS*) and to have minimal impacts on the user application integration. The solution is constituted by a runtime and a set of user APIs. Three main components are needed to host this runtime on top of an OS:

- FIFO fixed priority scheduler and POSIX thread support
- High resolution timer to count the elapsed time
- DVFS driver

During task execution, modifications of the core state may occur at specific points, termed "segment" boundaries. A segment is a section of code under timing constraint, which exports timing information at runtime. This decomposition enables the scheduler to identify and monitor the different execution paths taken by a task. It maintains an accurate vision of the work already accomplished and the work that remains to be done in a given task. This reporting is achieved through a specific API, in conjunction with an extension of POSIX threads, allowing timing annotations (WCET, deadlines) on thread-based tasks. The efforts made in standardizing dynamic scheduling interfaces [8][8] have been used as background.

Splitting a task into segments constitutes a trade-off. More segments help the scheduler track progression better, but incurs more timing overhead. In addition to these API hooks, the low-power scheduler requires a user-supplied segment table, which maps the name of all the segments in the application to their associated WCET, at the different core modes (filled at design time), as well as their deadlines.

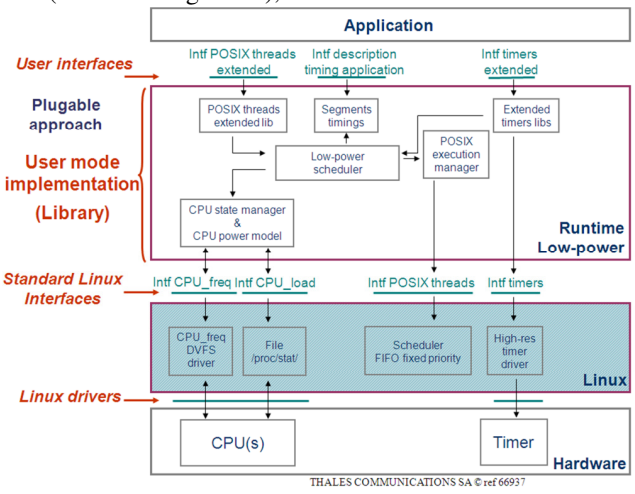


Fig. 3. Low-Power runtime architecture

As depicted in Fig. 3, the infrastructure is constituted by:

- The low-power scheduler: It holds scheduling strategies and is responsible for thread creation and synchronization.
- The CPU state manager & power model. This component is used to store an abstraction of processor in terms of power states, frequency and switch time duration.
- The POSIX thread extended lib. It is used to interpret the extended attributes related to the POSIX threads.
- The POSIX execution manager. This component is used to make the translation between the tasks formalism handled by the low-power scheduler and the Linux POSIX threads kernel scheduler.
- The segments timings table.

In the framework of the EU FP7-288307 funded project PHARAON, TCS has developed and implemented this runtime as a user space library on Linux for a quad core cortex A9 platform. This choice has been made for portability reasons but introduces overheads and could be thus limited for hard real-time systems. To ensure more determinism, implementation must be done in kernel space or on RTOS.

IV. SCHEDULING FOR SAFETY AND SECURITY CRITICAL SYSTEMS

A. *Pike OS*

PikeOS is a real-time operating system for safety and security critical applications [5] [6]. PikeOS is certified according standards DO-178B for avionics, IEC 61508 for railway and EN 50128 for safety in general. The PikeOS origin lies in the avionic area (e.g. Airbus A350, A400M), which is well-known for requiring highly robust components. Currently, PikeOS is used in different critical domains, it has highly modular and runs on a variety of hardware platforms.

Architecturally PikeOS consists of two major components: a micro-kernel and a virtualisation layer (see Figure 4). The micro-kernel is very compact and provides the very basic functionality inspired by the ideas of Liedtke [7]. The virtualisation layer is implemented on the top of the micro-kernel and provides separated execution partitions, also known as virtual machines, for user applications. User applications run in the isolated partitions which can be "personalised" with APIs, e.g. POSIX, OSEK, Linux etc.

The scheduler of PikeOS is a multi-core real-time time-triggered scheduling with support of integration events-triggered threads. In the rest of this section we focus on the scheduling algorithm implemented in PikeOS.

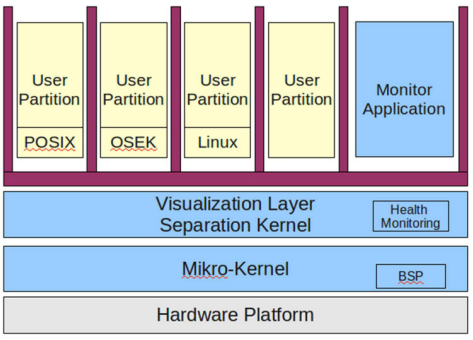


Fig. 4. PikeOS architecture

B. *Pike OS scheduling with time partionning*

Time triggered scheduling means that all relevant events are processed at predefined points. Usually, time-triggered

scheduling is defined before the system is deployed, i.e. the scheduler configuration is made offline. Such a scheduler is executed periodically, which guarantees predictable and repetitive scheduling of system processes.

The main part of such a scheduler is the configuration. In a configuration one splits some amount of time (e.g. 1 second) into *slots* (also known as time windows, time frames, etc.) and assigns processes/threads to be active at specific slots. The result of the splitting and assignment is one period of the time-triggered scheduling. In this document we call this period *major time frame (MTF)* and a set of consequent slots logically combined together a *time partition* (we denote a time partition as *tau*). The major time frame is repeatedly executed all over again during the whole lifetime of a system. During runtime the scheduler makes lookups in the configuration to decide if the next time partition has to be activated. The main goal of such time partitioning [5] is to ensure that activities in one time partition do not affect the timing of the activities in other time partitions. Time Partitioning allows the system integrator:

- Allocate a certain amount of CPU time to each virtual machine.
- One time partition can consist of more than one time slot.
- Multiple virtual machines can belong to the same time partition.
- Privileged processes are allowed to create and move threads to time partitions different from the time partition configured for their virtual machine.
- Independent scheduling inside every time partition.
- Support for different priorities inside every time partition.

Time partitioning is very important in the context of mixing safety, security, and real-time requirements. The best example is a safety critical application, which has to be able to work without interference (in this case the corresponding safety requirement "do the job") and at the same time it should not be possible to exploit scheduling algorithms as a timing covert channel. Time partitioning can also help system integrators to setup a safe access (i.e. access for safety-critical applications) or safe scheduling of accesses.

C. Background Time Partition

When all threads in a given VM and/or a given time slot have finished all their jobs, in a purely time-triggered partitioning the rest of the time has to be "burned". If a hypervisor forces switch of time slots, it can destroy the planned scheduling, e.g. next VMs will be scheduled earlier and could finish earlier than the expected arrival of some events.

We suggest the usage of a *background time partition* (we also called it *tau0*) to extend the time-triggered nature of time partitioning with event-triggered and priority-based features.

In a scheduler with a background partition there are always two active time partitions: the current one (according to the static time partitioning) and the background one. The next task is the one with the highest priority from these two time partitions. If priority of tasks is equal, then the background partition has precedence. Figure 5 depicts the scheduling decision in PikeOS.

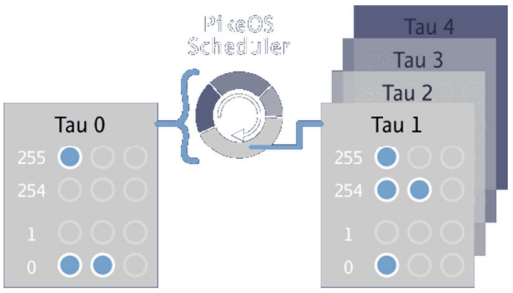


Fig. 5. Scheduling with Background Partition

Background partition allows system integrator to implement the following scenarios:

Utilisation of free time

System integrator assigns to the background partition a VM (or threads from a VM) with low priorities. In such configuration if there are not any active threads in current time partition, the threads from background partition will be automatically scheduled. Thus, the available resources will be utilised.

Shortening latency for processing real-time/critical events

System integrators assign to the background partition a VM with a thread, which should process some real-time event (or event with the shortest possible reaction time). In a default configuration this thread has the lowest priority, and thus, excluded from scheduling. Once a high-priority events (which is implemented as interrupt) arrives, the interrupt processing routine should rise the priority of the corresponding thread. For example, if it sets the priority of the thread in background partition to the maximum, it will be the next one to schedule.

V. Concept for Low Power Scheduling for Mixed Critical Systems

In this section we present three candidate algorithms for the implementation phase of our project. The goal is to propose extensions to systems using multiple software partitions with different criticalities. The first one introduces "*power budget*" for VM, second exploits background time-partition, and the third benefits of multi-core system with homogeneous or heterogeneous processing cores.

A. Power Budgeting VM

We introduce to the VM configuration a new parameter "*power budget*". This parameter defines how much power a VM is allowed to consume during its time partition.

- If a low critical VM has overpassed a predefined limit, PikeOS scheduler cuts off this VM from the assigned CPUs and applies a power management strategy to either reduce CPU frequency or even to put CPU in sleep mode depending on the remaining time before next VM slot.
- If a critical VM has overpassed its power budget during its time partition, the extra power consumed will be removed from the initial power budget of the next low critical VM scheduled.
- For battery powered devices, when power depletion becomes critical (e.g. low-level battery signal event), power budget of less critical VMs could be scaled down to extend autonomy in favor of critical VM.

These strategies will allow system integrator to configure a system where less critical VMs do not interfere with high-critical VM on the power resource. A strategy similar to the SCHED_EDF_SMP_DVFS policy (described in section III.A) at task level could have been considered as well. However, for safety and security reasons, we do not plan to schedule the next VM after current VM reached its power limit to guarantee predefined behavior of the major time frame and complete isolation of VM in space (e.g. memory) and time.

B. Power management of the background time-partition

We propose to allocate low-priority VMs to the background time-partition and exploit its ability to consume available CPU time after a critical VM in the foreground time-partition finishes its job (see IV.B for details). Thus, this designs allows to fine-tune CPU frequency when the low priority VMs are scheduled and the system integrator can selects the power management service to be applied to the VM, either:

- The standard OS power management service
- The low-power scheduler and its infrastructure such as described in III.B

This design requires a new parameter "CPU frequency" that must be controlled by the PikeOS scheduler. This is because when a critical VM is scheduled after a low priority VM, the CPU frequency status must be checked (and modified if needed) to ensure that VM execution will comply with the task performance as requested by the system integrator. The additional switching time overhead must also be taken into account for the switch between VMs.

C. Heterogeneous HW architectures

Multicore hardware platforms with heterogeneous cores such as ARM big.LITTLE architectures provide a great environment for mixed-critical systems power management. At task level, power management is transparently handled by the CPUfreq framework (compatible with the low power scheduler). When a high power mode is applied, tasks are mapped onto the "big" cores whereas when low power mode is applied, tasks are mapped onto the "LITTLE" cores.

In our approach, for heterogeneous architectures, we propose to reserve some CPUs for critical VMs to guarantee deterministic behavior. The system integrator must select at the design time such cores (or the frequency in case of big.LITTLE architectures). From a power consumption point of view, low performance CPUs can be allocated to the low priority VM to reduce power consumption and be managed via power-aware extension of the PikeOS scheduler.

VI. CONCLUSION

Non-functional aspects such as power or thermal management are becoming important features even for critical systems. DVFS and DPM are typical mechanisms used to limit power consumption. However, for critical systems, power management strategies available on standard OS are not suited because these knobs must obey to determinism, safety, and security requirements.

We presented TCS *lp_framework* solution with the *SCHED_EDF_SMP_DVFS* strategy implemented at the level of task scheduler. In mixed-critical use-cases a similar strategy can only be applied on low priority tasks, to satisfy requirements of high priority tasks and keep required security aspects. Integration of such power management into RTOS (in our case SYSGO PikeOS) can overcome this limitation and provide a base to treat power management on mixed-critical system designs.

Three algorithms have been presented and we believe that they provide great flexibility to choose the depth of integration of the power management functionality into RTOS PikeOS. This will also allows us to keep the size of the trusted base (or the source code which has to be certified) under full control. We see the following integration possibilities:

- Extension of PikeOS scheduler (e.g. V.A)
- Integration of power management policies at the level of user VM (V.B)
- Integration of power management policies at the level of PikeOS extensions (V.C)

Reaching optimal power savings is a challenge. Thus, in addition to the runtime algorithms proposed in this paper, a design methodology shall be also proposed to the system integrator to retrieve parameters that will help the runtime configuration taking better decisions during execution. For instance, schedulability analysis must be conducted to define the lowest VM's frequency affinity sill matching the system integrator performances in worst-case scenario. We also plan to enrich our algorithms with inputs from such methodologies.

REFERENCES

[1] T. Burd and R. Brodersen. Energy efficient CMOS microprocessor design. In Proceedings of the the 28th Hawaii International Conference on System Sciences, pages 288–297. IEEE Comput. Soc. Press, 1995,

[2] ACPICA website, http://www.acpica.org/community/

[3] Ch. Ykman-Couvreur, V. Nollet, Th. Marescaux, E. Brockmeyer, Fr. Catthoor, and H. Corporaal. Design-Time Application Mapping and Platform Exploration for MP-SoC Customized Run-Time Management. IET Comput. Digit. Tech. vol. 1(2), pp. 120-128 (2007)

[4] S. Gheorghita, M. Palkovic, J. Hamers, A.Vandecappelle, S. Mamagkakis, T. Basten and L. Eeckhout. System-Scenario-Based Design of Dynamic Embedded Systems. ACM Trans. on Design Automation of Electronic Systems, vol. 14(1), (2009)

[5] Kaiser, R., Wagner, S.: Evolution of the PikeOS microkernel. In: Kuz, I., Petters, S.M. (eds.) MIKES: 1st International Workshop on Microkernels for Embedded Systems (2007), http://ertos.nicta.com.au/publications/papers/Kuz_Petters_07.pdf

[6] SYSGO AG: PikeOS RTOS technology embedded system software for safety critical real-time systems. http://www.sysgo.com (2013)

[7] Liedtke, J.: On micro-kernel construction. In: Proceedings 15th ACM Symp Operating systems principles. pp. 237–250. ACM Press (1995)

[8] OMG Real-time CORBA scheduling, available at http://www.omg.org/technology/documents/formal/realtime_CORBA

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

[10] D. Zhu, R. Melhem, and B. Childers, "Scheduling with dynamic Voltage/Speed Adjustment Using Slack Reclamation in Multi-Processor Real-Time Systems", IEEE Trans. on Parallel & Distributed Systems, vol. 14, no. 7, pp. 686 - 700, 2003.

[11] D. Shin, J. Kim, and S. Lee. Intra-task voltage scheduling for low-energy hard real-time applications. IEEE Design & Test of Computers, 18(2):20–30, 2001

Reverse engineering power management on NVIDIA GPUs - Anatomy of an autonomic-ready system

Martin Peres
Ph.D. student at LaBRI
University of Bordeaux
Hobbyist Linux/Nouveau Developer
Email: martin.peres@labri.fr

***Abstract*—Research in power management is currently limited by the fact that companies do not release enough documentation or interfaces to fully exploit the potential found in modern processors. This problem is even more present in GPUs despite having the highest performance-per-Watt ratio found in today's processors. This paper presents an overview of the power management features of modern NVIDIA GPUs that have been found through reverse engineering. The paper finally discusses about the possibility of achieving self-management on NVIDIA GPUs and also discusses the future challenges that will be faced by researchers to study and improve on autonomic systems.**

I. Introduction

Historically, CPU and GPU manufacturers were aiming to increase performance as it was the usual metrics used by consumers to select what to buy. However with the rise of laptops and smartphones, consumers started preferring higher battery-life, slimmer, more silent and cooler devices. Noise was also a concern among some desktop users who started using water cooling in place of fans. This led CPU/GPU manufacturer to not only take performance into account, but also performance-per-Watt. A higher performance-per-Watt means a lower heat dissipation for the same amount of computing power which in turn allows shrinking the radiator/fan to keep the processor cool. This results in slimmer and/or more silent devices. Decreasing power usage is also a major concern of datacenters and supercomputers as they already consumed 1.5% of the electricity production in the USA in 2007 [1].

As power management (PM) is a non-functional/auxiliary feature, it is usually non-standard, poorly documented and/or kept secret. As some PM features require the intervention of the host system, a driver is often needed to obtain the best performance-per-Watt. This is not a problem for closed-source OS such as Microsoft Windows™or Mac OS™as the manufacturer usually supplies a binary driver for these platforms. It is however a major problem for Open Source operating systems like Linux as those features are not documented and cannot be re-implemented easily in an Open Source manner.

The absence of documentation and the lack of open hardware also sets back research as researchers are limited to using the documented PM features. This lack of documentation is most present in the GPU world, especially with the company called NVIDIA which has been publicly shamed for that reason by Linus Torvalds [2], creator of Linux. With the exception of the ARM-based Tegra, NVIDIA has never released enough documentation to provide 3D acceleration to a single GPU after the Geforce 4 (2002). Power management was never supported nor documented by NVIDIA. As GPUs are leading the market in terms of performance-per-Watt [3], they are a good candidate for a reverse engineering effort of their power management features. The choice of reverse engineering NVIDIA's power management features makes sense as they are usually preferred in supercomputers such as Titan [4], the current fastest supercomputer. As a bonus, it would allow many laptops to increase their battery life when using the Linux community driver for NVIDIA GPUs called Nouveau [5].

Nouveau is a fork of NVIDIA's limited Open Source driver, xf86-video-nv [6] by Stephane Marchesin aimed at delivering Open Source 3D acceleration along with a port to the new graphics Open Source architecture DRI [7]. As almost no documentation from NVIDIA is available, this driver mostly relies on reverse engineering to produce enough knowledge and documentation to implement the driver. The Nouveau driver was merged into Linux 2.6.33, released in 2009, even though 3D acceleration for most cards have been considered experimental until 2012. I personally joined the team in 2010 after publishing [8] a very experimental version of power management features such as temperature reading and reclocking, which are essential to perform Dynamic Voltage/Frequency Scaling (DVFS). Since then, power management is being investigated and implemented in the driver whenever a feature is well understood.

This paper is an attempt at documenting some of the features that have been reverse engineered by the Nouveau team and compare them to the state of the art. Some of the documented features have also been prototyped and implemented in Nouveau. A brief evaluation of the GPU power consumption has also been carried out.

Section II presents an overview of the power-related functionalities needed to make the GPU work. Performance counters and their usage is detailed in section III. In section IV, we document the hardware fail-safe feature to lower temperature and power consumption. Section V introduces NVIDIA's RTOS embedded in the GPU. Finally, section VI presents the vision of autonomic computing and how NVIDIA GPUs can fit to this model.

II. General overview of a modern GPU

This section is meant to provide the reader with information that is important to understand the following sections.

A. General-Purpose Input/Output (GPIO)

A General-Purpose Input/Output (GPIO) is a pin of a chip that can be software-selected as a binary input or output at run time.

Input GPIO pins are used by NVIDIA to sense if the external power supply or the SLI brige is connected, if the GPU is overheating (in the case of an external temperature sensor) or to read the fan rotational speed. Output GPIO pins are used by NVIDIA to drive a laptop's backlight, select the memory and core voltage or adjust the fan speed.

Some GPIO pins can be connected to special features like hardware pulse-width modulation (PWM) controllers which allows for instance to vary the amount of power sent to the fan or the backlight of a laptop's screen. Some chips also have a hardware tachometer which calculates the revolution time of the fan. All these operations could be done in software but they would require the CPU to wake up at least dozens of times per second, issue a read/write request through the PCIe [9] port and then go back to sleep again. Waking up the CPU for such trivial operations is inefficient and should thus be avoided.

B. Energy sources

A modern desktop GPU draws its power from the PCIe port or PCIe connectors (6 or 8 pins). The PCIe port and 6-pin PCIe connectors can each source up to 75W while the 8-pin PCIe connector can source up to 150W.

These power sources all provide different voltages that are way higher than the operating voltage of the GPU. The DC-DC voltage conversion is done by the voltage controller which sources from the PCIe port and connectors and outputs power to the GPU at its operating voltage. The output voltage is software-controllable using a GPIO pin. A GPIO pin is also usually used by cards with PCIe connectors to sense if they are connected to a power source. This allows NVIDIA's driver to disable hardware acceleration when the connectors are not connected, avoiding the GPU from draining too much current from the PCIe port. In some cases, a GPIO pin can also control the emergency shutdown of the voltage controller. It is used by the GPU to shutdown its power when the card overheats.

There are currently only up to two power domains on NVIDIA cards, one for the GPU and one for memory. Memory's power domain is limited to 2 possible voltages while the GPU usually has at least 3 and can have up to 32 voltage possibilities for its power domain.

On some high-end cards, the voltage controller can also be queried and configured through an I^2C interface. Those expensive controllers can sense the output power along with the power draw of each of the energy sources.

Figure 1 illustrates a simple view of the power subsystem.

C. Temperature management

Just like modern CPUs, GPUs are prone to overheating. In order to regulate their temperature, a temperature sensor needs to be installed. This sensor could be internal to the chip or be external using an I^2C [10] sensor. Usually, external sensors can also drive a fan's speed according to the card's temperature. When using the internal sensor, the driver is usually responsible for polling the temperature and updating the fan speed.

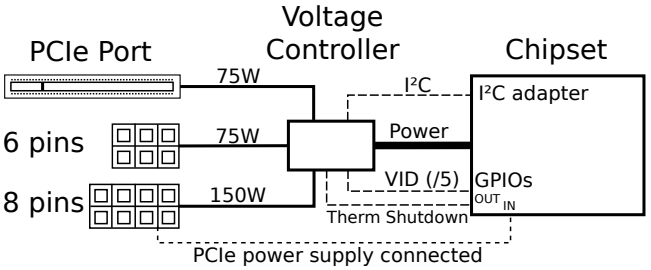


Figure 1. Overview of the energy sources of the GPU

D. Clock tree

Having the possibility to change frequencies on the fly is another feature of modern GPUs aimed at lowering power consumption. On NVIDIA GPUs, there are two clock sources, the PCIe clock (100 MHz) and an on-board crystal (usually 27 MHz). The frequency of the clocks is then increased by using a Phase-locked loop (PLL). A PLL takes the input frequency F_{in} and outputs the frequency F_{out}. The relation between F_{in} and F_{out} in the simplest PLL is detailed in equ. 1. The parameters N and M are integers and have a limited range. This range depends on the PLL and is usually documented in the Video BIOS (VBIOS). This means that not all output frequencies are achievable.

$$F_{out} = F_{in} * \frac{N}{M} \tag{1}$$

It would be simplistic to think that the GPU uses a single clock for the whole card. Indeed, a modern GPU has multiple engines running asynchronously which are clocked by different clock domains. There are up to 13 clock domains on Fermi [11] among which we can find the host, video RAM (VRAM), shader, copy (asynchronous copy engine) and ROP domains. The number of clock domains varies depending on the chipset and the chipset family.

The clock tree has been reverse engineered by using the performance counters (detailed in section III) which can count every clock cycle of most clock domains. The clocks can be read through nvatiming, a tool from the envytools repository [12]. This tool should be run before and after modifying the content of a register that is suspected to configure the clock tree. As a start, it is possible to check all the registers NVIDIA's proprietary driver reads from or writes to when reclocking the GPU. These reads/writes can be logged by tracing the Memory-Mapped IO (MMIO) accesses of the proprietary driver using the MMIO-trace feature of the Linux kernel [13].

E. The role of the video BIOS

On an IBM-PC-compatible computer, the BIOS is responsible for the Power-On Self Test (POST). During the POST phase, the BIOS performs some sanity checks and then initialises VBIOS peripherals by giving them addresses in the

linear address space and running their internal BIOS when applicable.

Similarly, in the case of the video card, the video BIOS (VBIOS) is responsible for booting up the card enough to provide the VGA/VESA interface to the host. This interface is then used by the BIOS and the bootloader to display boot information on the screen.

In order to bring up this interface, the VBIOS configures the GPU's clock tree, configures the memory controllers, uploads default microcodes to some engines, allocates a framebuffer to store the content of the screen and finally performs graphic mode setting. The VBIOS is then called by the BIOS when an application on the host requests anything such as changing the resolution. The VBIOS is written in a 16 bit x86 code and should be considered as being a low-level simple driver.

As NVIDIA does not select the VRAM chips, the voltage controller, where the GPIO pins are connected or what graphics connectors (HDMI, DVI, VGA, DP, etc...) are available, the card manufacturers need a way to tell the VBIOS how to set up the card at boot time. This is why NVIDIA stores the manufacturer-dependent information in tables in the VBIOS. These tables can be decoded by nvbios, found in the envytools collection [12]. This collection also contains nvagetbios and nvafakebios which respectively allow to download the VBIOS or to upload a new VBIOS non-permanently.

As manufacturers are using the same BIOS for several cards, some of the tables are indexed by a strap register. This register's value is set by the manufacturer by tying some pins of the GPU to specific voltages.

1) The GPIO & external device tables: They store which devices or functions are accessible through the chip's pins and how to access them. The EXTDEV table references I^2C-accessible devices while the GPIO table only references GPIO-accessible functions. Both also represent what is accessible by a number, called a tag or type. In the case of the GPIO table, each entry contains at least a GPIO pin, the associated tag (for instance, PANEL_PWR), the direction (input or output), the default state (HIGH or LOW) and if the GPIO is inverted or not. In the case of the EXTDEV table, each entry contains the device type/tag, which I^2C bus it is accessible on and at which address. Possible devices could be the ADT7473 which is an external temperature management unit or the voltage controller PX3540.

2) The thermal table: Its role is to store temperature-related parameters, as defined by the OEM. The temperature sensor's parameters (offset and slope) can be adjusted. It also defines hysteresis and temperature thresholds such as fan_boost, downclock and shutdown which are respectively defining the temperature at which the fan should be set to 100%, the temperature at which the card should be downclocked and the temperature at which the computer should be shut down. Finally, the fan response to the temperature can be linear or trip-point based. The thermal table then stores the parameters for either method.

3) The performance level table: It specifies up to 4 performance levels. A performance level is defined by a set of clock speeds, a core voltage, a memory voltage and memory timings and a PCIe link width speed. The voltage is stored as an ID that needs to be looked up in the voltage-mapping table. It is by altering this table that [3] managed to force NVIDIA's proprietary driver to set the clocks the way they wanted.

4) The voltage and voltage-mapping tables: The voltage table contains {*Voltage ID*, *voltage*} tuples describing the various supported voltages and how to configure the voltage controller. Those voltages are referenced by the voltage-mapping table which defines {*ID*, *voltage_min*, *voltage_max*} tuples. The voltage_min and voltage_max parameters of this table define an acceptable voltage range for the performance level referencing the ID.

5) The RAM type, timings and timings-mapping tables: Their role is to tell the driver which memory type and timings should be set when reclocking memory. The RAM type table is indexed by the strap. This information is necessary in order to program the memory controller properly as DDR3, GDDR3 and GDDR5 do not have the same configuration registers. The timings-mapping table contains several entries, each covering a memory frequency range. The values of each entry tell how to configure the memory controllers whenever the driver wants to use a frequency from within this range. Each entry contains sub-entries which are indexed by the strap register. Each sub-entry contains the timing ID of the timings table that should be used along with some memory-specific parameters. The timings table contains several entries which are referenced by the timings-mapping table. Each entry is either the values that should be written to the memory timings registers or the aggregation of several parameters that allow the driver to calculate these values. Unfortunately, the equations to calculate the values from the parameters greatly varied in the Geforce 8 era and are not completely understood on some GPUs.

F. Reclocking process

Reclocking is the act of changing the frequencies of the clocks of the GPU. This process drastically affects the performance and the power consumption of the GPU. The relation between clocks, power consumption and performance is very hardware- and task-dependent. There is however a known relation between the voltage, the frequency of the clock and the final power consumption for CMOS circuits [14] as shown by equ. 2 and 3.

$$P = P_{dynamic} + P_{static} \tag{2}$$

$$P_{dynamic} = CfV^2 \tag{3}$$

P is the final power consumption of the circuit and results from both the dynamic ($P_{dynamic}$) and the static (P_{static}) power consumption of the circuit.

The static power consumption comes from the leakage power of the transistors when they are not being switched (clock gated). It can only be lowered by shutting down the power (power gating) of the units of the GPU that are not in use. On NVIDIA hardware, power gating an engine can be done by clearing its associated bit in one register. Patents [15] and [16] on engine-level power gating from NVIDIA are

informative about the way power gating may be implemented on their hardware.

The dynamic power consumption is influenced by the capacitance of the transistor gates C (which decreases with the transistor size), the frequency at which the circuit is clocked f and the voltage at which the circuit operates V. As C is only dependent on the way transistors where etched, it cannot be adjusted at run time to lower power consumption. Only f and V can be adjusted at run time by respectively reprogramming the clock tree's PLLs and reprogramming the voltage controller. While f has an immediate impact on performance, V has none even though it needs to be increased with f in order for the chip to be able to meet the needed switching time. Another way to decrease the dynamic power consumption is to cut the clock of the parts of the chip that are not used at the moment to compute something meaningful (clock gating). The actual implementation by NVIDIA's has not been reverse engineered yet but hints of how it works may be found in patents [17] and [18].

Practical tests showed that reclocking a Geforce GTX 480 can achieve a 28% lower power consumption while only decreasing performance by 1% for a given task [3].

G. Reclocking a GPU - Rough overview & constraints

Due to the fact that GPUs are almost not documented and that the driver's interfaces are mostly closed, DVFS has poorly been studied in practice on GPUs, contrarily to CPUs. The only stable open-source implementation of discrete-GPU DVFS that I know of is available in the Radeon driver [19] and has been available since 2010. Some insightful comments from one of the engineer who made this happen are publicly available [20]. Reclocking on NVIDIA GPUs with Open Source software has been an on-going task since 2010 [8] and I presented my first experimental DVFS implementation at the X.org developer conference in 2011 [21]. Since then, Ben Skeggs has also managed to implement an experimental DVFS support for some selected cards of the Kepler family which may be published this year.

The differences found in GPUs compared to CPUs are the multiple clock domains and the fact that not all clocks can be adjusted at any time mainly due to the real time constraints imposed by streaming pixels to the screen. This constraint is imposed by the CRT Controller (CRTC) which reads the pixels to be displayed to the screen (framebuffer) from the video memory and streams it through the VGA/DVI/HDMI/DP connection. As reclocking memory requires putting the VRAM in a self-refresh mode which is incompatible with answering memory requests and as reclocking the PLLs takes a few microseconds, reclocking cannot be done without disturbing the output of the screen unless done during the screen's vertical blank period. This period was originally introduced to let the CRT screen move the electron beam from the bottom-right to the top-left corner. This period lasts about 400 to 500 μs and is more than enough to reclock memory. However, on a screen refreshed 60 times per second, vblank only happens every 16.6ms. In the case where the user connects several monitors to the GPU, the vblank periods are not usually synchronised which prevents reclocking memory tearlessly on all monitors. In this case, the NVIDIA binary driver selects the highest performance level and deactivates DVFS.

Contrarily to memory reclocking, engines reclocking can be done at any time as long as the GPU is idle. The GPU can generally be forced to idle by disabling command-fetching and waiting for the GPU to finish processing. The engine reclocking process is not fully understood on NVIDIA cards although I found an empirical solution that managed to reclock the GPU millions of times without crashing on the Tesla chipset family. Further investigation is still needed.

III. PCOUNTER : Performance counters

The (hardware) performance counters are a block in modern microprocessors that count low-level events such as the number of branch taken or the number of cache hit/miss that happened while running a 3D or a GPGPU application. On NVIDIA's Kepler family, there are 108 different GPGPU-related monitorable events documented by NVIDIA.

A. Usage

Performance counters provide some insight about how the hardware is executing its workload. They are a powerful tool to analyse the bottlenecks of a 3D or a GPGPU application. They can be accessed through NVIDIA PerfKit [22] for 3D applications or through Cupti [23] for GPGPU applications.

The performance counters can also be used by the driver in order to dynamically adjust the performance level based on the load usage of the GPU and provide DVFS.

Some researchers also proposed to use performance counters as an indication of the power consumption with an average accuracy of 4.7% [24].

B. How does it work

The performance counter engine (PCOUNTER) is fairly well understood thanks to the work of Marcin Kościelnicki. Here is a short description on how this block works in the Tesla family (Geforce 8).

PCOUNTER receives hardware events through internal connections encoded as a 1-bit value which we call signal. This signal is sampled by PCOUNTER at the rate of clock of the engine that generated the event. An event counter is incremented every time its corresponding signal is sampled at 1 while a cycles counter is incremented at every clock cycle. This simplistic counter is represented by figure 2.

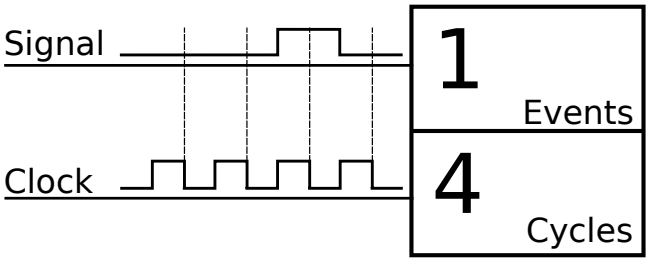


Figure 2. Example of a simple performance counter

However, it is expensive to have a counter for all possible signals. The signals are thus multiplexed. Signals are grouped into domains which are each clocked by one clock domain.

There are up to 8 domains which hold 4 separate counters and up to 256 signals. Counters do not sample one signal, they sample a macro signal. A macro signal is the aggregation of 4 signals which have been combined using a function. An overview of this logic is represented by figure 3.

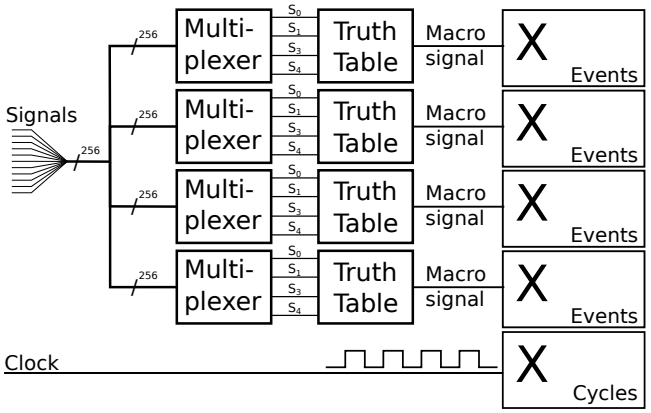


Figure 3. Schematic view of a domain from PCOUNTER

The aggregation function allows to specify which combination of the 4 signals will generate a 1 in the macro signal. The function is stored as a 16 bit number with each bit representing a combination of the signals. With $s_x(t)$ being the state of selected signal x (out of 4) at time t, the macro signal will be set to 1 if the bit $s_3(t) * 2^3 + s_2(t) * 2^2 + s_1(t) * 2^1 + s_0(t)$ of the function number is set.

As an example, to monitor signal s_0 only, the aggregation function should be programmed so as no matter the state of signals 1, 2 and 3, whenever signal 0 is high, the output of aggregation function should be high too. As can be seen in table I, the function number should have all the odd bits set and all the even bits cleared. The function number is thus $1010101010101010_{(2)}$.

Table I. TRUTH TABLE TO MONITOR s_0 ONLY

s_3	s_2	s_1	s_0	Decimal	Selection
0	0	0	0	0	
0	0	0	1	1	X
0	0	1	0	2	
0	0	1	1	3	X
0	1	0	0	4	
0	1	0	1	5	X
0	1	1	0	6	
0	1	1	1	7	X
1	0	0	0	8	
1	0	0	1	9	X
1	0	1	0	10	
1	0	1	1	11	X
1	1	0	0	12	
1	1	0	1	13	X
1	1	1	0	14	
1	1	1	1	15	X

The 4 counters of the domain can be used independently (quad-event mode) or used together (single-event mode). The single-event mode allows counting more complex events but it is not discussed here. PCOUNTER also has a record mode which allows saving the content of the counters in a buffer in VRAM periodically so as the driver does not have to poll the counters as often. The full documentation can be found of at hwdocs/pcounter.txt in envytools [12]. Most signals are still unknown although some compute-related signals are already available on nve4 (Kepler) thanks to the work of Christoph Bumiller and Ben Skeggs. More signals are currently being reverse engineered by Samuel Pitoiset [25] as part of his Google Summer of Code 2013 project.

IV. PTHERM : THERMAL & POWER BUDGET

PTHERM is a piece of hardware that monitors the GPU in order to make sure it does not overheat or exceed its power budget.

A. Thermal management

The primary function of PTHERM is to make sure the GPU does not exceed its temperature budget. PTHERM can react to some thermal events by automatically set the fan speed to 100%, by lowering some frequencies of the GPU, or by shutting down the power to the card.

Reading the temperature from the internal sensor can be done simply by reading a register which stores the temperature in degrees Celsius. The sensor's calibration was performed in factory and is stored in fuses which allows the GPU to monitor its temperature at boot time.

PTHERM generates thermal events on reaching several temperature thresholds. Whenever the temperature reaches a threshold, an interruption request (IRQ) can be sent to the host for it to take actions to lower the temperature. The IRQ can be sent conditionally, depending on the direction of the temperature (rising, falling or both). The hardware can also take actions to lower the temperature by forcing the fan to 100% or by automatically lowering the clock frequency. The latter feature will be explained in the following subsections. However, only 3 thresholds can generate automatic hardware response. The others are meant to be used by the driver.

In the case where the GPU is using an external temperature sensor, hardware events are gathered through the chip's GPIO pins which are connected to the external sensor. The external chip is then responsible for monitoring the temperature and compare it to certain thresholds. These threshold are programmable via I^2C and are set at boot time during the POST procedure and again when the driver is loaded by the host computer.

B. Power reading

Estimating the power consumption can be done in real time using two different methods.

The first one is to read the power consumption by measuring the voltage drop across a shunt resistor mounted in series with the chip's power line. This voltage drop is linear with the current flowing through this power line with a factor of R_{shunt}. The instantaneous power consumption of the chip is then equal to the voltage delivered by the voltage controller times the measured current. This method is explained in figure 4.

However, this technique requires an Analog-to-Digital-Converter and some dedicated circuitry The cost of this solution is quite high as it requires dedicated hardware on the PCB of the GPU and a fast communication channel between the ADC and the chip. Also, fast ADCs are expensive. Therefore, it

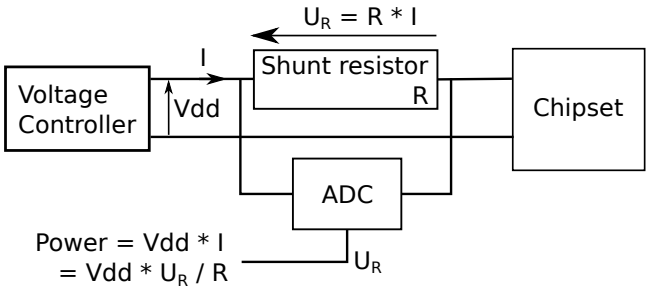


Figure 4. Measuring the power consumption of a chip

explains why only the voltage controllers from high-end cards can output this information.

Another solution is to monitor power consumption by monitoring the block-level activity inside the chip. As explained by one of NVIDIA's patents [26], power consumption can be estimated by monitoring the activity of the different blocks of the chip, give them a weight according to the number of gates they contain, sum all the values, low-pass filter them then integrate over the refresh period of the power estimator. This is very much alike the approach explained in [24], where performance counters were used to compute the power consumption, except that it is done by the hardware itself. This solution was introduced by NVIDIA in 2006, is explained in patent [26] and is told to produce a power estimation every 512 clock cycles of an unspecified clock. In our case, it seems to be the host clock, sourced by the PCIE port and usually running at 277 MHz. Polling the power estimation register seems to validate this theory as the refresh rate seems to be around 540 kHz.

NVIDIA's method is thus very fast and cheap as it only needs a small increase of gate count on the GPU. Moreover, the output of this method can be used internally to dynamically adjust the clock of some engines to stay inside the power budget. This technique is explained in the following subsection.

Unfortunately, on GPU families Fermi and newer, NVIDIA stopped specifying the activity block weights which disables power readings. It is still possible to specify them manually to get the power reading functionality back. However, those weights would have to be calculated experimentally.

C. FSRM: Dynamic clock frequency modulation

PTHERM's role is to keep the GPU in its power and thermal budget. When the GPU exceeds any of its budget, it needs to react by lowering its power consumption. Lowering the power consumption is usually done by reclocking the GPU but full reclocking cannot be done automatically by the hardware because it cannot calculate all the parameters and follow the reclocking process.

Letting the driver reclock the GPU when getting close to overheating is acceptable and PTHERM can assist by sending an IRQ to the driver when the temperature reaches some thresholds. However, in the case where the driver is not doing its job, because it is locked up or because the driver is not loaded, the chip should be able to regulate its temperature without being forced to cut the power of the GPU.

In the case of meeting the power budget, reacting fast to an over current is paramount to guarantee the safety of the power supply and the stability of the system in general. It is thus very important to be able to reclock often.

It is not possible to reprogram the clock tree and adjust the voltage fast-enough to meet the 540 kHz update rate needed for the power capping. However, the clock of some engines can be slowed down. This will linearly affect the dynamic part of the power consumption albeit not as power efficient as a full reclocking of the GPU because the voltage is not changed.

A simple way to lower the frequency of a clock is to divide it by a power of two although the granularity is too coarse to be used directly for the power capping capability. It is however possible to lower the average clock frequency by sometimes selecting the divided clock and then selecting the original clock the rest of the time. For the lack of a better name, I decided to call this technique Frequency-Selection Ratio Modulation (FSRM). FSRM can be implemented by using the output of a Pulse-Width Modulator (PWM) to a one bit multiplexer. When the output of the PWM is high, the original clock is being used while the divided clock is used when the output is low. Any average clock frequency between the divided clock and the original clock is thus achievable by varying the duty cycle of the PWM.

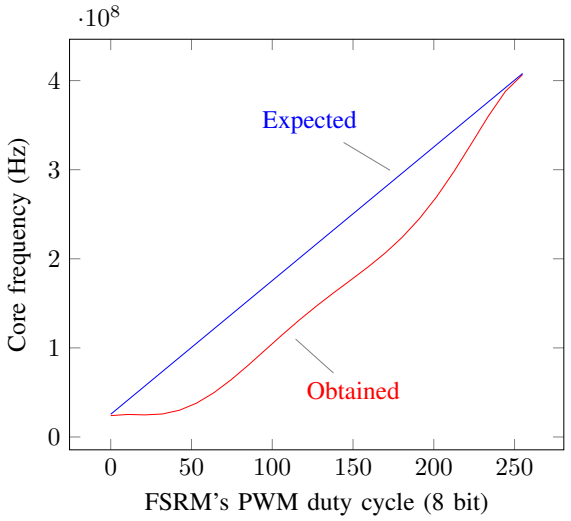


Figure 5. Frequency of the core clock (@408MHz, 16-divider) when varying the FSRM

Figure 5 presents the expected frequency response of the above system along with what has actually been measured through the performance counters when tested on NVIDIA's hardware implementation. Judging by the differences, it seems like NVIDIA also added a system to smooth the change between the slowed and the normal clock. The difference is also likely explained by the fact that the clock selection may only happen only when both the original and the divided clock are rising. This also raises the problem of synchronising the original and the divided clock as the divided clock has to go through more gates than the original one. In order to synchronise them, the original clock would have to be shifted in time by adding redundant gates. This issue is known as clock skew [27].

D. FSRM usage & configuration

The FSRM is used to lower PGRAPH clock's frequency. PGRAPH is the main engine behind 2D/3D acceleration and GPGPU, and probably responsible for most of the power consumption.

There are several events that can trigger use of the FSRM:

- PGRAPH idling;
- Temperature reaching one of the several thresholds;
- Power usage reaching its cap;
- Driver forcing the FSRM.

Whenever one of these events happen, the divided clock and the FSRM get updated following to their associated configuration. The clock divisor can be set to 1, 2, 4, 8 or 16 while the FSRM can range from 0 (use the divided clock all the time) to 255 (use the normal clock all the time). However, even though each temperature threshold can specify an independent clock divisor, they have to share the same FSRM.

Some preliminary tests have been performed on an nv84 to lower the clock to the maximum when PGRAPH is idle and this resulted in a about 20% power reduction of the computer while the impact on the framerate of a video game was not measurable. Some cards do not enable this feature by default, possibly suggesting that it may lead to some instabilities. This is however really promising and will be further investigated.

In the case of the power limiter, another mode can be selected to dynamically update the FSRM. This allows to lower the frequency of PGRAPH as little as possible in order to stay in the power budget. This mode uses 2 windows, one including the other entirely. Each window will increase the FSRM whenever the power reading is lower than the lowest threshold of the window and decrease the FSRM when the reading is above the highest threshold of the window. The increase and decrease values are independent for both windows and can be set arbitrarily. However, the outer window is supposed to increase or decrease the FSRM rapidly while the inner window is supposed to make finer adjustments.

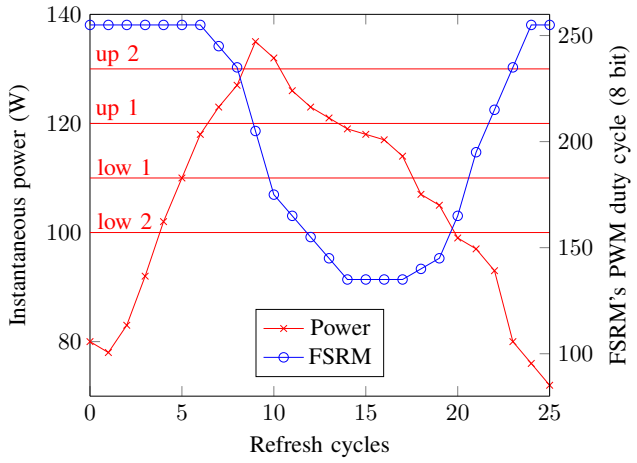


Figure 6. Example of the power limiter in the dual window mode

An example can be seen in figure 6 where the outer window is set to [130W, 100W] while the inner window is set to [120W, 110W]. The outer window will increase the FSRM by 20 when the power is lower than 100W and will decrease it by 30 when the power is above 130W. The inner window will increase the FSRM by 5 when the power is between 120 and 130W and will decrease it by 10 when the power is between 100 and 110W. The FSRM is limited to the range [0, 255].

E. Power limiter on Fermi and newer

As no power reading is possible by default on GPUs of the Fermi family and newer, the power limiter cannot be used. This came to me as a surprise as NVIDIA started advertising the power limitation on Fermi. This suggests that they may have implemented another way to read and lower the power consumption of their GPUs. I unfortunately do not have access to a card with such feature but the independent and proprietary tool GPU-Z [28] proposes a way to disable this power cap, as can be seen on figure 7.

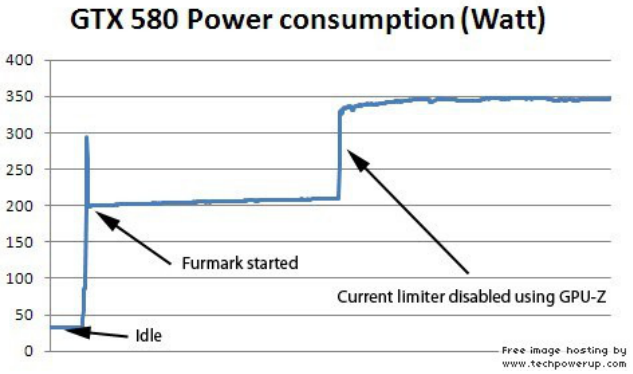


Figure 7. Effect of disabling the power limiter on the Geforce GTX 580. Copyrights to W1zzard from techpowerup.com.

The first spike would seem to suggest a very slow response to exceeding the power budget. It is thus possible that NVIDIA would use its driver to poll the voltage controller's power reading and decide to reclock the card to a lower performance level. Since GPU-Z is proprietary, more reverse engineering will be needed to understand and document this feature.

V. PDAEMON : An RTOS embedded in your GPU

PDAEMON is an engine introduced on nva3, a late GPU from the Tesla family. This engine is fully programmable using the instruction set (ISA) FμC (Flexible MicroCode).

FμC was introduced in nv98 for PCRYPT, an engine meant to offload some cryptographic operations. FμC was then reused for PDAEMON and many more engines of the Fermi family. This feature-rich ISA is now being used to implement some of the interface between the host and the hardware of some engines thus bringing a lot more flexibility to the hardware's interfaces with the host. An assembler has been written for this ISA and is available under the name envyas in the envytools repository [12]. A FμC LLVM backend project was also started but never gained traction and was later deleted.

PDAEMON is an engine meant to offload some operations usually performed by the host driver. It is clocked at 200 MHz, has a memory management unit (MMU), has access to all the registers of the GPU and direct access to PTHERM. It also supports timers, interrupts and can redirect the interrupts from the GPU to itself instead of the host. Several independent communication channels with the host are also available. Surprisingly, it also has performance counters to monitor some engines' activity along with its own. In other words, PDAEMON is a fully-programmable "computer" with low-latency access to the GPU that can perform more efficiently whatever operation the host can do. However, it cannot perform heavy calculations in a timely fashion because of its limited clock frequency.

For more information about PDAEMON's capabilities, please read hwdocs/pdaemon.txt and all the files from hwdocs/fuc/ found in the envytools repository [12].

A. Usages of PDAEMON

PDAEMON's usage by NVIDIA has not been fully reverse-engineered yet. However, the uploaded microcode contains the list of the processes executed by the RTOS developed by or for NVIDIA. Here are some of the interesting processes:

- Fan management;
- Power gating;
- Hardware scheduling (for memory reclocking);
- Power budget enforcement;
- Performance and system monitoring.

Its usage list could also be extended to cover:

- Parsing the VBIOS;
- Implementing full DVFS support;
- Generating power-usage graphs.

B. Benefits of using PDAEMON

PDAEMON has clearly been developed with the idea that it should be able to do whatever the host system can do. One of the practical advantage of using PDAEMON to implement power management is that the CPU does not need to be awoken as often. This lowers the power consumption as the longer the CPU is allowed to sleep, the greater the power savings are.

Another benefit of using PDAEMON for power management is that, even when the host system has crashed, full power management is still available. This has the benefit of checking the thermal and power budget of the GPU while also lowering the power consumption of crashed system.

VI. The GPU as an autonomic-ready system

In 2001, IBM proposed the concept of autonomic computing [29] to aim for the creation of self-managing systems as a way to reduce their usage complexity. The idea was that systems are getting more and more complex and, as such, require more knowledge from the technicians trying to maintain them. By having self-managing systems, the user could write a high-level policy that would be enforced by the system itself, thus hiding complexity.

As an example, a modern NVIDIA GPU could perform the following self-functions:

self-configuration: The GPU is responsible for finding the optimal configuration to fill the user-requested tasks.

self-optimization: Using performance counters, the GPU can optimise performance and also lower its power consumption by using DVFS.

self-healing: As the GPU can monitor its power consumption and temperature, it can also react to destructive behaviours by lowering the frequency of the clocks.

self-protection: Isolation between users can be provided by the GPU (integrity and confidentiality) while availability can be achieved by killing long-running jobs ran by users. The GPU can also store secrets like HDCP [30] keys or even encrypt/decrypt data on the fly using PCRYPT.

Although the aim of autonomic computing is primarily oriented towards lowering human maintenance cost, it can also be extended to lower the development cost. By having self-managing subsystems for non-functional features, integration cost is lowered because of the reduced amount of development needed to make it work on a new platform. Ideally, a complete system could be assembled easily by using unmodified autonomic subsystems and only a limited amount of development would be needed to make their interfaces match.

This approach does not make sense in the IBM-PC-compatible personal computer market as the platform has barely evolved since its introduction in the way components interact (POST procedure, x86 processors and high speed busses). This renders the development of drivers executed on the CPU cheaper than having a dedicated processor for the driver's operations. However, in the System-On-Chip (SoC) world where manufacturers buy IPs (intellectual property blocks) and assemble them in a single-chip system, the dedicated-processor approach makes a lot of sense as there are no single processor ISA and operating system (OS) that the IP manufacturer can depend on. In this context, the slimmer the necessary driver for the processor and operating system, the wider the processor and OS choice for the SoC manufacturer.

This is the approach that have chosen Broadcom for his Videocore technology which is fully controllable by a clearly-documented Remote Procedure Calls (RPC) interface. This allowed the Raspberry Pi foundation to provide a binary-driver-free Linux-based OS on their products [31]. However, the Open Source driver only uses this RPC interface is thus not a real driver as it is just some glue code. This led the graphics maintainer of Linux to refuse including this driver in Linux as the code running in Videocore is still proprietary and bugs there are unfixable by the community [32].

Broadcom's Videocore is a good example of both the advantage and the drawbacks of autonomic systems. Adding support for it required very limited work by the Raspberry Pi foundation but it also means the real driver is now a black box running on another processor which cannot be traced, debugged, or modified easily. From a research perspective, it also means it will become harder to study the hardware

and propose improvements to the software. In the case of NVIDIA, this situation could happen if PDAEMON's code was not readable and writable by the host system.

VII. Conclusion

In this paper, innovative power management features of NVIDIA GPUs have been partially documented through reverse engineering. Even though the reverse engineering work is incomplete, it is already clear that what has been implemented in NVIDIA GPUs enhances the state of the art and that a GPU can be a great test bed for doing power management research. Moreover, it is already known that these features can be combined in order to create a self-managed system, following IBM's vision on autonomic computing. It is my hope that this article will spur more interest in the research community to study, document and improve GPU power management.

Future work will focus on creating a stable reclocking process across all the modern NVIDIA GPUs in order to create a testbed for experimenting with DVFS algorithms. More work will also be done on power and clock gating which are the two main power management features which have not been documented yet.

Acknowledgment

The author would like to thank everyone involved in graphics on Linux or other FLOSS operating systems for laying out the foundations fors the work done in Nouveau. I would however like to thank in particular Marcin Kościelnicki for reverse engineering and writing most of the documentation and tools found in envytools. I would also like to thank the Nouveau maintainer, Ben Skeggs, for figuring out many VBIOS tables along with the clock tree on most cards and reviewing the code I wrote for Nouveau DRM. I should also thank Roy Spliet for working with me on figuring out how to calculate the memory timings based on the parameters found in the memory timings table for the nv50 family and proofreading this paper. Finally, I would like to thank my Ph.D. supervisor Francine Krief for funding me and letting me work on this project.

References

[1] EPA, "EPA report to congress on server and data center energy efficiency," Tech. Rep., 2007. [Online]. Available: http://hightech.lbl.gov/documents/data_centers/epa-datacenters.pdf

[2] aaltouniversityace, "Aalto talk with linus torvalds [full-length]," Jun. 2012. [Online]. Available: https://www.youtube.com/watch?v=MShbP3OpASA&t=2894s

[3] Y. Abe, H. Sasaki, M. Peres, K. Inoue, K. Murakami, and S. Kato, "Power and performance analysis of GPU-accelerated systems," in *Proceedings of the 2012 USENIX conference on Power-Aware Computing and Systems*, ser. HotPower'12. Berkeley, CA, USA: USENIX Association, 2012, p. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387869.2387879

[4] "Titan, world's #1 open science supercomputer." [Online]. Available: http://www.olcf.ornl.gov/titan/

[5] Nouveau Community, "Nouveau, the community-driven open source NVIDIA driver," 2006. [Online]. Available: http://nouveau.freedesktop.org/wiki/

[6] NVIDIA, "xf86-video-nv : NVIDIA's open source driver," 2003. [Online]. Available: http://cgit.freedesktop.org/xorg/driver/xf86-video-nv/

[7] X.org, "Direct rendering infrastructure." [Online]. Available: http://dri.freedesktop.org/wiki/

[8] M. Peres, "[nouveau] [RFC] initial power management vbios parsing, voltage & clock setting to nouveau." Sep. 2010. [Online]. Available: http://lists.freedesktop.org/archives/nouveau/2010-September/006499.html

[9] PCI-SIG, "PCI express," 2013. [Online]. Available: http://www.pcisig.com/specifications/pciexpress/resources/

[10] NXP, "I2C-Bus: what's that?" [Online]. Available: http://www.i2c-bus.org/

[11] C. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU architecture," *IEEE Micro*, vol. 31, no. 2, pp. 50–59, 2011.

[12] "Envytools - tools for people envious of nvidia's blob driver." [Online]. Available: https://github.com/pathscale/envytools

[13] Pekka Paalanen, "MMIO-trace," 2008. [Online]. Available: http://nouveau.freedesktop.org/wiki/MmioTrace/

[14] E. Le Sueur and G. Heiser, "Dynamic voltage and frequency scaling: the laws of diminishing returns," in *Proceedings of the 2010 international conference on Power aware computing and systems*, ser. HotPower'10. Berkeley, CA, USA: USENIX Association, 2010, p. 1–8. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924920.1924921

[15] R. Singhal, "POWER GATING TECHNIQUE TO REDUCE POWER IN FUNCTIONAL AND TEST MODES," U.S. Patent 20 100 153 759, Jun., 2010. [Online]. Available: http://www.freepatentsonline.com/y2010/0153759.html

[16] Z. Y. Zheng, O. Rubinstein, Y. Tan, S. A. Jamkar, and Y. Kulkarni, "ENGINE LEVEL POWER GATING ARBITRATION TECHNIQUES," U.S. Patent 20 120 146 706, Jun., 2012. [Online]. Available: http://www.freepatentsonline.com/y2012/0146706.html

[17] S. C. LIM, "CLOCK GATED PIPELINE STAGES," Patent WO/2007/038 532, Apr., 2007. [Online]. Available: http://www.freepatentsonline.com/WO2007038532A2.html

[18] K. M. Abdalla and R. J. Hasslen III, "Functional block level clock-gating within a graphics processor," U.S. Patent 7 802 118, Sep., 2010. [Online]. Available: http://www.freepatentsonline.com/7802118.html

[19] Radeon Community, "Radeon - an open source ATI/AMD driver." [Online]. Available: http://xorg.freedesktop.org/wiki/radeon

[20] Matthew Garrett, "Radeon reclocking," Apr. 2010. [Online]. Available: http://mjg59.livejournal.com/122010.html

[21] X.org, "X.org developer conference 2011," Sep. 2011. [Online]. Available: http://www.x.org/wiki/Events/XDC2011

[22] NVIDIA, "PerfKit." [Online]. Available: https://developer.nvidia.com/nvidia-perfkit

[23] ——, "CUDA profiling tools interface." [Online]. Available: https://developer.nvidia.com/cuda-profiling-tools-interface

[24] H. Nagasaka, N. Maruyama, A. Nukada, T. Endo, and S. Matsuoka, "Statistical power modeling of GPU kernels using performance counters," in *Green Computing Conference, 2010 International*, 2010, pp. 115–122.

[25] Samuel Pitoiset, "Samuel pitoiset's open source blog," 2013. [Online]. Available: https://hakzsam.wordpress.com/

[26] H. Cha, R. J. Hasslen III, J. A. Robinson, S. J. Treichler, and A. U. Diril, "Power estimation based on block activity," U.S. Patent 8 060 765, Nov., 2011. [Online]. Available: http://www.freepatentsonline.com/8060765.html

[27] J. Fishburn, "Clock skew optimization," *IEEE Transactions on Computers*, vol. 39, no. 7, pp. 945–951, 1990.

[28] W1zzard, "TechPowerUp GPU-Z v0.7.1," 2013. [Online]. Available: http://www.techpowerup.com/downloads/2244/techpowerup-gpu-z-v0-7-1/

[29] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[30] Digital Content Protection, LLC, "High-bandwidth digital content protection (HDCP)." [Online]. Available: http://www.digital-cp.com/

[31] Alex Bradbury, "Open source ARM userland," Oct. 2012. [Online]. Available: http://www.raspberrypi.org/archives/2221

[32] Dave Airlie, "raspberry pi drivers are NOT useful," Oct. 2012. [Online]. Available: http://airlied.livejournal.com/76383.html

The State of COMPOSITE*

Jiguo Song, Qi Wang, Gabriel Parmer
The George Washington University
{jiguos,interwq,gparmer}@gwu.edu

I. THE TAO OF COMPOSITE.

COMPOSITE is a component-based operating system that has been under development since 2006 with design goals including configurability, predictability, and reliability. Unlike many previous component-based operating systems that focus on kernel-based configurability, COMPOSITE implements most system policies, mechanisms and abstractions as user-level, hardware-protected, fine-grained units of functionality that are harnessed through well-defined interfaces. COMPOSITE's structure is most similar to μ-kernels: "A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e., permitting competing implementations, would prevent the implementation of the system's required functionality"[1]. COMPOSITE philosophically expands on this in two ways:

1) *Component-based policy definition.* We strive to eliminate policies from the kernel, thus including only mechanisms. This enables for both customized resource management, and for designers to trade between complexity and TCB size, for flexibility and capability. Though the line between policy and mechanism is not clean [2], functionality common to most modern microkernels including scheduling and structured memory mapping is moved to user-level components where it can be redefined. Unlike exokernels [3] we avoid distributed management of resources, instead centralizing the policy into specific manager components. To enable flexibility of resource management (diversity of policy), *resource managment abstraction* is enabled via inter-component protocols to hierarchically control scheduling, manage memory, or perform I/O [4]. This support enables concurrent execution of multiple virtual environments that trade between heightened isolation with customized resource management, and resource sharing.

2) *System behavior via composition of fine-grained components.* One of the most successful component-based systems is the UNIX command line, based on the composition of simple programs into pipelines of complex functionality. COMPOSITE emphasizes the composition of complex systems from fine-grained components. The structure of this composition is a general DAG, and the functional protocols between components are encoded in explicit interfaces. Though a pervasive *separation of concerns, and extensive interface-level polymorphism*, developers have significant leeway in programming a system all the way down to resource management policies *at the composition-level*.

Mutable Protection Domains enables protection boundaries between components to be dynamically raised and lowered [5] to trade protection and performance. Collections of components can be collapsed into the same protection domain to mimic the structure of μ-kernels, monolithic systems, or exokernels. This fine-grained control over protection domains enables the generic study of system structure.

COMPOSITE's focus. COMPOSITE provides unique opportunities due to its component-based structure. These include:

- *Configurability.* Supports concurrent execution of divergent virtual environments ranging from a separation-kernel environment emphasizing strong barriers between high- and low-criticality tasks, to a simple web-server that serves both static and dynamic content composed of 25 components. In the server, different communication protocols, altering the data source, or even changing the interrupt scheduling to more aggressively avoid livelock, can all be done by changing the composition of components. Though extensively decoupled, this web-server performs at least as well as Apache.
- *Predictability.* All aspects of the system are designed around the goal of bounded-latency. Notably, COMPOSITE places an emphasis on the end-to-end bounded latency of invocations across a possibly long chain of components. This solves by design the dependency problem that complicates scheduling in many component-coordination systems. This end-to-end predictability is currently being extended to multi-core systems.
- *Reliability.* By pervasively memory isolating components, fault propagation is significantly constrained in COMPOSITE. COMPOSITE enables even the lowest-level system components to fail, and will predictably reconstitute their state with overhead on the order of 10s of μ-seconds.

II. CURRENT STATE OF COMPOSITE

COMPOSITE is a research OS, and current goals do not include executing existing applications. Including external libraries, COMPOSITE is 160K lines of code (LOC) including a 7K LOC kernel, and 30K in components (minus third-party libraries). The system includes some POSIX support, some scripting language support (via LUA), and networking via LWIP.

Who should use COMPOSITE? COMPOSITE is in a state of constant development, and is not yet appropriate for production environments. Researchers investigating some combination of OS structure, resource management, parallelism, and real-time execution could benefit from the system. Developers interested in expanding the corpus of components are always welcome.

Online presence. The development mailing list, and more information can be found at http://composite.seas.gwu.edu/. The source is available at https://github.com/gparmer/Composite.

This material is based upon work supported by the National Science Foundation under Grants No. CNS 1137973, CNS 1149675, and CNS 1117243. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] J. Liedtke, "On micro-kernel construction," in *Proceedings of the 15th ACM Symposium on Operating System Principles*. ACM, December 1995.
[2] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf, "Policy/mechanism separation in hydra," in *Proceedings of SOSP*, 1975.
[3] D. R. Engler, F. Kaashoek, and J. O'Toole, "Exokernel: An operating system architecture for application-level resource management," in *Proceedings of SOSP*, 1995.
[4] G. Parmer and R. West, "HiRes: A system for predictable hierarchical resource management," in *Proceedings of RTAS*, 2011.
[5] ——, "Mutable protection domains: Adapting system fault isolation for reliability and efficiency," in *ACM Transactions on Software Engineering (TSE)*, July/August 2012.

Priority Inheritance on Condition Variables

Tommaso Cucinotta
Bell Laboratories, Alcatel-Lucent Ireland
Email: tommaso.cucinotta@alcatel-lucent.com

***Abstract*—In this paper, a mechanism is presented to deal with *priority inversion* in real-time systems when multiple threads of execution synchronize with each other by means of mutual exclusion semaphores coupled with the programming abstraction of *condition variables*. Traditional priority inheritance solutions focus on addressing priority or deadline inversion as due to the attempt to lock mutual exclusion semaphores, or deal exclusively with specific interaction patterns such as client-server ones. The mechanism proposed in this paper allows the programmer to explicitly declare to the run-time environment what tasks are able to perform a notify operation on a condition over which other tasks may be suspended through a wait operation. This enables developers of custom interaction models for real-time tasks to exploit their knowledge of the application-specific interaction so as to potentially reduce priority inversion. The paper discusses issues in the realization of the technique, and its integration with existing priority inheritance mechanisms on current operating systems. Also, the paper briefly presents the prototyping of the technique within the open-source RTSim real-time systems simulator, which is used to highlight the potential advantages of the exposed technique through a simple simulated scenario.**

I. Introduction and problem presentation

Priority inversion is a well-known problem in the literature of real-time systems occurring every time a task execution is delayed due to the interference of lower priority tasks. This problem is well-known to happen whenever a higher-priority task tries to acquire a lock on a mutual-exclusion semaphore (shortly, a *mutex*) already locked by a lower-priority task. Clearly, the lock owner task needs to release the lock before the more urgent one can proceed and this delay is unavoidable. However, if a third task with a middle priority between these two is allowed to preempt the lower-priority task holding the lock, then the release of the lock is delayed even further, adding an unnecessary delay to the execution of the higher-priority task, waiting for the lock to be released.

For example, Figure 1 shows a sequence in which three tasks, A, B and C are scheduled on the same processor by using a fixed-priority scheduler, and A and C synchronize on a mutex M for the access to some common data structure. Task C runs while no other higher-priority task is ready to run. Then, it locks the mutex (operation L(M) in the picture) but, before being able to release it (operation U(M) in the picture), it is preempted by the higher-priority task A that just woke up. Task A executes for a while, then it tries to lock the same mutex already locked by C, thus it suspends allowing C to continue execution. Unfortunately, C cannot execute for much time, because the middle-priority task B wakes up at this point, preempting C as due to its higher-priority. Even though B has a higher-priority than C, we know that C holds

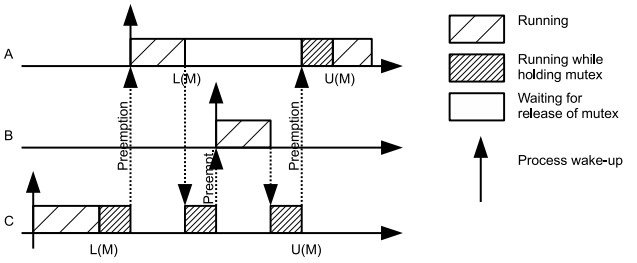


Figure 1. Sample priority inversion scenario. Task A has the highest priority, task C the lowest, and task B has a middle priority between them.

a lock for which the highest-priority task A is waiting, thus B should not be allowed to preempt C in such a case. Therefore, the time for which B keeps executing, delaying the release of the lock by C, constitutes an avoidable additional delay for the task A.

This problem has been addressed in a number of ways in the literature, for example by the well-known Basic Priority Inheritance and Priority Ceiling mechanisms [1], [2]. The related literature is discussed in Section II later.

A mutex is commonly used for synchronization of tasks in conjunction with the *condition variables* programming abstraction, a mechanism that allows a task to suspend its execution waiting for a condition to be satisfied on certain variables. The typical example is the one of a reader from a queue of messages waiting for someone to write into the queue when it is empty. When the reader tries to read an element from the queue but finds it empty, it suspends itself till the number of elements in the queue becomes greater than zero (as a consequence of a writer task pushing one element). In such a case, the reader typically blocks on a condition variable with an operation that atomically suspends the task and releases the mutex (e.g., by using the POSIX [3], [4] `pthread_cond_wait()` call) used for critical sections operating on the queue. The writer, on the other hand, after insertion of an element in the queue, notifies possible readers through a notify operation (e.g., the POSIX `pthread_cond_notify()` call).

In such cases, a form of unnecessary priority inversion may still occur. Consider for example the scenario depicted in Figure 2. Task A communicates with a set of other tasks C, D and E via a message queue Q, through which it expects to receive some message from them. Now, imagine that, for whatever reason, in the set of tasks potentially producing messages for A, there is also a task C with a priority lower than

the one of A. In the shown scenario, A tries to read atomically from the queue (operation R(Q) in the picture), thus it locks the queue mutex, but releases it immediately after detecting that the queue is empty, blocking on a condition variable. Task C then executes, but, before it finishes its computations for writing into the message queue Q, it gets preempted by a third unrelated task B with a priority level higher than B but lower than A. The time for which B runs constitutes an unnecessary delay for the execution of the higher-priority task A, which is waiting for B to write something into the shared queue.

With the Priority Inheritance protocol, whenever a task blocks trying to acquire a lock on a mutex that is already locked by another task, the former task can temporarily donate its priority to the latter task, in order to let it progress quicker (avoiding unneeded preemptions) towards releasing the lock. However, for the scenario depicted in Figure 2, even if concurrent access to the queue Q is protected by a mutex exhibiting Priority Inheritance, the mechanism cannot help. Indeed, in this case Task C is not holding the lock of the Q mutex while it is progressing towards completing the computations that will lead to the production of a data item to be pushed into the message queue Q. Only as part of the atomic push and pop operations into and from the queue, does a task acquire the mutex lock protecting access to the queue. Therefore, Priority Inversion can merely fire in the short time instants of execution of the atomic operations, but not during the generally longer execution of the tasks.

Abstracting away from the specific example of shared message queues, generally speaking, consider a set of real-time tasks synchronizing through the use of mutex and condition variables. Then, if a task that needs to wait for a condition to become true may be unnecessarily delayed by lower-priority tasks, then a form of priority inversion can occur. Indeed, if the task(s) responsible for letting the mentioned condition become true run(s) at a lower-priority in the system, and a third task with a middle priority level wakes up, said third task may preempt the execution of those lower-priority tasks, thus delaying the achievement of the condition for which the higher-priority task is waiting. In this case, the traditional mechanism of Basic Priority Inheritance cannot help, because the higher-priority task waiting for the condition to become true drops the mutex lock before suspending through a wait operation on the condition variable, and the lower-priority task(s) that need to progress in their computations so as to perform the notify operation on the condition variable do not hold any mutex lock while they are computing. Furthermore, the run-time has generally no information about the (application-specific) interaction among the tasks, so it cannot infer automatically the needed task dependency information.

In some cases, real-time tasks might interact through higher-level mechanisms that allow the run-time to actually know, when a task suspends, which other tasks may actually cause the resume of the suspended task. For example, this is the case of the client-server interaction model of the Ada language run-time, in which a client task invokes explicitly an Entry of a server task, i.e., it pushes an element into the server input queue and suspends till it receives a response. In such a case, the run-time knows which particular server has to perform work on behalf of which client(s), so it can correctly apply Priority Inheritance, as shown in the seminal work on the topic by Sha et al. [5]. However, in the general case of tasks interacting by application-specific synchronization protocols realized through mutex and condition variables, such information is not readily available to the run-time.

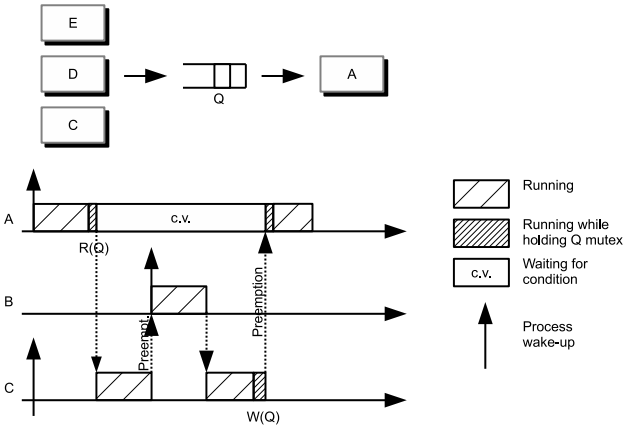


Figure 2. Priority inversion scenario with task A receiving data from a lower-priority task C through a shared message-queue Q realized with a mutex and a condition variable. Task B has middle-priority between A and C.

A. Paper contribution

In this paper, a mechanism is proposed to let the run-time be aware of the possible dependencies among tasks within a real-time system, expanding the functionality of the programming abstraction of condition variables. With the new mechanism, called PI-CV, the programmer may declare what are the tasks that may help a condition become true, over which other tasks may be waiting. Exploiting such dependency information, the run-time can trigger the necessary priority inheritance that is needed to avoid priority inversion. PI-CV alleviates the problem of priority inversion in cases in which developers code into the system custom, application-specific communication and synchronization logic through mutex and condition variables.

There are various scenarios in which the introduced mechanism may be useful and indeed improve responsiveness of real-time software components. For example, in the literature of real-time systems, it is very common to see real-time applications modeled as Directed Acyclic Graphs (DAGs) of computations which are triggered periodically or as a result of external events. Each node in the DAG can start its computations once its inputs are available (see Figure 3), which in turn are produced as output of the computations of other nodes. The mechanism is particularly useful in contexts in which producers and consumers of data share common data structures in shared memory (serializing the operations on it by means of semaphores and synchronizing among each other by means of condition variables), but at the same time they possess different

priority or criticality. This situation is very common in real-time systems. For example, we can easily find co-existence of both the main real-time code, characterized by stringent timing constraints, and other side software components that are needed for monitoring or configuration purposes. Often it happens that some (often bidirectional) information flow is needed between these two worlds (e.g., the monitoring code needs to retrieve information about the status of the real-time code, and the real-time code needs to reconfigure itself according to the configuration passed by reconfiguration code).

II. Related work

The literature on the management of shared resources for real-time systems is huge. In this section, the main works related to the problem of priority inversion are shortly recalled.

During the International Workshop on Real-Time Ada Issues, back in 1987, Cornhill and Sha reported [6] various limitations of the Ada language when dealing with priority-based real-time systems. Specifically, a high-priority task could be delayed indefinitely by lower priority tasks under certain conditions. Shortly afterward, the same authors formalized [7] what are the correct interactions between client and server tasks in form of assertions on the program execution. The Ada run-time was not respecting those assertions, thus allowing tasks to undergo unnecessary priority inversion. In the same work, Priority Inheritance was informally introduced as a general mechanism for bounding priority inversion. Later, Sha et al. [2], [1] described better their idea formalizing the two well-known Basic Priority Inheritance (BPI) and Priority Ceiling (PCP) protocols. While BPI allows a task to be blocked multiple times by lower priority tasks, with PCP a task can be blocked at most once by lower-priority tasks, so priority inversion is bounded by the execution time of the longest critical-section of lower-priority tasks; also, PCP prevents deadlock. A possible realization of PCP for the Ada language has been described by Goodenough and Sha [8], and by Borger and Rajkumar [9]. Also, Locke and Goodenough discussed [10] some practical issues in applying PCP to concrete real-time systems.

Various extensions to PCP have been proposed, for example to deal with reader-writer locks [11], multi-processor systems [12], [13], [14] and dynamically recomputed priority ceilings [15]. Furthermore, Baker introduced [16] Stack Resource Policy (SRP), extending PCP so as to handle multi-unit resources, dynamic priority schemes (e.g., EDF), and task groups sharing a single stack ("featherweight" processes), treated on its own as a resource with a zero ceiling. Also, Gai et al. investigated [17] on minimizing memory utilization when sharing resources in multiprocessor systems. More recently, Lakshmanan et al. [18] further extended PCP for multi-processors grouping tasks that access a common shared resource and co-locating them on the same processor.

Schmidt et al. investigated [19] on various priority inversion issues in CORBA middleware, and proposed an architecture (TAO) not suffering of such problem. Priority inversion has also been considered by Di Natale et al. in a proposal [20] for schedulability analysis of real-time distributed applications, where, despite the use of PCP for scheduling tasks on the CPUs, non-preemptability of packet transmissions causes unavoidable priority inversion when a higher-priority packet reaches the transmission queue while a low-priority packet is being transmitted.

When scheduling under the Constant Bandwidth Server (CBS) [21], Lamastra et al. proposed [22], [23] the BandWidth Inheritance (BWI) protocol, allowing a task owning a lock on a mutex not only to inherit the (dynamic) priority of the highest priority waiting task (if higher than its own), but also to account for its execution within the reservation of the task whose priority is being inherited. This allows to keep the *temporal isolation* property ensured by the CBS, in the sense that non-interacting task groups cannot interfere on each other's ability to meet their timing constraints. Later, Faggioli et al. [24] discussed various issues and optimizations in the implementation of the protocol in the Linux kernel, and specifically as an add-on to the AQuoSA scheduler [25]. An extension of BWI to multi-processor systems has been proposed again by Faggioli et al. [26], where the implementation of the technique [27] was prototyped this time on the LITMUS-RT [28] real-time test-bed.

Block et al. proposed FMLP [29], a resource locking protocol for multi-processor systems allowing for unrestricted critical-section nesting and efficient handling of the common case of short non-nested accesses.

Guan et al. dealt [30] with real-time task sets where interactions among tasks are only known at run-time depending on which particular branches are actually executed.

Many other works exist in the literature [31], [32], [33], [34], [35], [36], [37] on variants of the above resource-sharing protocols and their analysis. An overview of them can be found in [27]. Recently, techniques to mitigate priority inversion have also been applied in the context of scheduling virtual machines communicating with each other [38]. A very interesting recent work by Abeni and Manica [39] adapts BWI to trigger priority inheritance on client-server interactions, and presents a schedulability analysis for that particular type of scenario. The mechanism being presented in this paper is more generic as it can be used with custom inter-thread communications. Though, the analysis presented by Abeni may constitute a valuable starting point for the analysis of the generic scenarios addressed by the present paper.

The above reviewed literature on resource sharing in real-time systems focuses essentially on dealing with priority inversion (and applying various types of priority inheritance mechanisms) in two main scenarios: 1) tasks interacting by the use of shared memory and critical sections, serialized through mutexes; 2) tasks interacting in a client-server fashion, where the server task executes operations on behalf of various clients.

In this paper, a general priority inheritance mechanism is presented, useful when tasks interact by using condition variables associated with mutexes. These are generally used in the implementation of custom shared data types supporting custom communication and synchronization protocols in concurrent

systems. In such a case, when a task, after entering a critical section, suspends itself through a wait operation on a condition variable, it also releases the mutex associated with the critical section. At this point, some other task running in the system may be the one responsible for the notify operation on the same condition variable, waking up the task(s) suspended on it. However, without further information, the run-time cannot generally know which task(s), among the currently ready-to-run ones, may perform such a notify operation. If the interacting tasks have different priorities, then the system may undergo avoidable priority inversion. With the mechanism proposed in this paper, the run-time is informed by the tasks about which other tasks may possibly help and accelerate the wake-up of a task suspended on a condition variable, thus enabling the avoidance of this kind of priority inversion. The mechanism can also be composed with existing priority inheritance schemes for lock-based interactions. Furthermore, it can also be used for realizing priority inheritance in client-server interactions, in Ada-rendezvous style. However, it can also be used in arbitrary, application-specific interactions programmed through mutual exclusion semaphores and condition variables.

Note that Hart and Guniguntala [40] made changes to the GNU `libc` pthreads library and kernel in order to support efficient wake-up of multiple tasks waiting on a condition variable (as due to a `pthreads_cond_broadcast()`) used in connection with an rt-mutex, so as to avoid the "thundering herd" effect, and guaranteeing the correct wake-up order (considering also priority inheritance). Such changes relate to the support for priority inheritance in rt-mutexes and they are not to be confused with the mechanism being proposed in this paper.

To the best of my knowledge, there are no alternatives for dealing with the specific type of problem of priority inversion as described above, in presence of condition variables. Commonly known alternatives to using semaphores and locks at all, include recurring to lock-free data structures, and solutions based on the Transactional Memory programming paradigm [41]. Lock-free programming is well-known to be more complex and difficult to master, than traditional lock-based programming. The advantage of the presented technique is that it allows applications developers to keep designing code using traditional synchronization primitives, i.e., mutual exclusion semaphores and condition variables, but they can improve the responsiveness of their applications with the very little additional effort to sort out which are the helper tasks for the condition variables they use (or, sometimes, the helper tasks may be automatically identified in proper libraries, see Section IV-B later). On the other hand, the Transactional Memory programming paradigm is particularly useful in presence of non-blocking operations on shared data structures, i.e., operations that would not lead to the suspension of the calling task in order to wait for a condition to become true, as it happens with condition variables, thus it does not constitute an alternative to the presented technique. A thorough and detailed comparison among these communication and synchronization techniques is outside the scope of this paper.

III. Priority inheritance on condition variables

In what follows, without loss of generality, the term *task* will be used to refer to a single thread of execution in a computing system, being it either a thread in a multi-threaded application, or a process. Also, without loss of generality, the term *priority* will be used to refer to the right of execution (or "urgency" level) of a task as compared to other tasks from the CPU(s) scheduler viewpoint. This includes both the priority of tasks whenever they are scheduled according to a priority-based discipline and their deadline whenever they are scheduled according to a deadline-based discipline (and their time-stamp whenever they are scheduled according to other policies based for example on virtual times, such as the Linux CFS [42]). However, the described technique is not specifically tied to these scheduling disciplines and it can be applied in presence of other schedulers as well. Furthermore, it should be clarified that this paper deals with how to dynamically change the priorities of tasks within a system, which is orthogonal with respect to how said tasks are scheduled on the available processors. Specifically, analyzing the consequences of the introduced technique on schedulability of real-time systems in presence of multi-core and/or multi-processor systems is out of the scope of this paper.

The mechanism of priority inheritance on condition variables (PI-CV) proposed in this paper works as follows:

- it is possible (but not mandatory) to programmatically associate a condition variable with the set of tasks able to speed-up the verification of the condition; these tasks will be called *helper tasks*; the set of helper tasks associated with a condition variable can be fixed throughout the lifetime of the condition variable, or be dynamically changed at run-time, according to the application needs;
- whenever a higher-priority task executes a wait operation on a condition variable, having a non-null set of helper tasks, *it temporarily donates its priority to all the lower-priority helper tasks*, so as to "speed-up" the verification of the condition associated with the condition variable;
- *as soon as the condition variable is notified, the dynamically inherited priority is revoked*, restoring the original priority of the helper tasks;
- *the mechanism can be applied transitively*, if one or more helper tasks suspends on other condition variables;
- *the mechanism can be nicely integrated with traditional (Basic) Priority Inheritance*, resulting in priority being (transitively) inherited from a higher priority task to a lower priority one either because the former waits to acquire a lock held by the latter, or because the former suspended through a wait operation on a condition variable for which the latter is a helper task.

Whenever a higher-priority task is suspended waiting for some output produced by lower-priority tasks, PI-CV allows the lower-priority tasks to temporarily inherit the right of execution of the higher-priority task with respect to the tasks scheduler. In order for the mechanism to work, it is necessary to introduce a few interface modifications to the classical

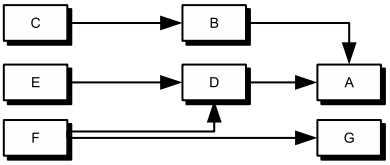


Figure 3. General interaction scenario where priority inheritance on condition variables may be applied transitively. Task F is waiting on a condition variable having tasks D and G registered as helpers.

condition variables mechanism as known in the literature, so that the run-time environment (e.g., the Operating System) knows which lower-priority tasks should inherit the priority of a higher-priority task suspending its execution waiting for a condition to become true. The interface may allow the mechanism of priority inheritance on condition variables to be enabled selectively on a case-by-case basis (per-condition variable and per-semaphore), depending on the application and system requirements (see below).

Priority inheritance may be applied transitively, when needed. For example, if Task A blocks on a condition variable donating temporarily its priority to Task B, and Task B in turn blocks on another condition variable donating temporarily its priority to Task C, then Task C should inherit the highest priority among the one associated with all the 3 tasks. Also, priority inheritance for condition variables can be integrated with traditional priority inheritance (or deadline inheritance) as available on current Operating Systems, letting the priority transitively propagate either due to an attempt of locking a locked mutex, or to a suspension on a condition variable with associated one or more helper tasks.

In other words, consider a blocking chain of tasks $(\tau_1, \tau_2, \ldots, \tau_n)$ where each task τ_i $(1 \leq i \leq n-1)$ suspended on the next one τ_{i+1} either trying to acquire a lock (enhanced with priority or deadline inheritance) already held by τ_{i+1}, or waiting on a condition variable (enhanced with PI-CV as described in this document) where τ_{i+1} is registered among the helper tasks. All the tasks in such a blocking chain are suspended, except the last one (that is eligible to run). This last task inherits the priority of any of the tasks in any blocking chain terminating on it, i.e., any task in the direct acyclic graph of blocking chains that terminate on it. For example, consider the scenario shown in Figure 3, where each arrow from a task to another means that the former is suspended on the latter due to either a blocking lock operation or a wait on a condition variable where the latter task is one of the helpers. Task A inherits the highest priority among tasks B, C, D, E, F, while G inherits the priority of F, if all of the suspensions happen through mutex semaphores enriched with priority inheritance or condition variables enriched with PI-CV. In the depicted scenario, note that F is waiting on a condition variable where both D and G are registered as helpers. This allows both of them to inherit the priority of F, until the condition is notified.

A. Reservation-based scheduling

Also, whenever a task is associated by the scheduler with a maximum time for which it may execute within certain time intervals, as in reservation-based scheduling [43], [44] (e.g., the POSIX Sporadic Server [3] or the CBS), the inheritance mechanism may behave in such a way that the helper task executing as a result of its priority having been boosted by the described priority inheritance mechanism, will account its execution towards the execution-time constraints of the task from which the priority was inherited (i.e., the budget of its server). For example, referring to the Bandwidth Inheritance (BWI) protocol [22], it is straightforward to think of the corresponding extension. In a BWI-CV protocol, whenever a task inherits the priority of a higher-priority task, the ready-to-run tasks at the end of the blocking chains (involving both attempts to acquire locks and wait operations on condition variables with associated other helper tasks) also execute in the server of the highest-priority task that is donating its priority to them, depleting its corresponding budget. Namely, the server to consider for budget accounting purposes should be the one associated with the highest-priority task, among the ones in the Direct Acyclic Graph (DAG) of all the blocking chains terminating on the said ready-to-run task.

B. Multi-processor systems

Note that PI-CV can be applied to single-processor as well as to multi-processor and multi-core systems. PI-CV merely allows the programmer to declare which are the helper tasks for each given condition variable at each time throughout the program life-time, and the run-time applies priority inheritance as described above. The specifics about how exactly tasks are scheduled in a multi-processor environment are outside the scope of this paper.

C. Schedulability analysis

PI-CV is presented in this paper without any particular associated schedulability analysis technique nor formal proof. As the mechanism allows for reducing priority inversion, it is expectable that the worst-case and/or average-case interference terms in schedulability analysis calculations, as coming out considering the specifics of the scheduling policy being employed on a system, have a shorter duration. This is shown by simulation in a simple scenario later in Section VI.

Similarly to the traditional Priority Inheritance mechanism available on current Operating Systems, PI-CV may reduce unneeded priority inversion in certain scenarios, leading to an improved responsiveness of the highest priority activities within a system. Also, when combined with resource reservations along the lines of BWI [22], [23], a BWI-CV mechanism should be capable of guaranteeing temporal isolation among non-interacting task groups. However, a theoretical analysis would be useful to provide a strong assessment on the (worst-case) responsiveness of the various real-time activities, including understanding whether it will be possible to meet all deadlines for higher-priority tasks that may benefit from PI-CV, as well as for lower-priority ones that may worsen their behavior, in presence of interactions based on condition variables. Further development of these concepts is left as future work.

IV. Implementation notes

From an implementation standpoint, the proposed mechanism may be made available to applications via a specialized library call that can be used by a task to declare which other tasks are the potential helpers towards the verification of the condition associated with a condition variable. For example, in an implementation leveraging the pthreads library implementation, this can be realized through the following C library calls:

```
int pthread_cond_helpers_add
  (pthread_cond_t *cond, pthread_t *helper);
int pthread_cond_helpers_del
  (pthread_cond_t *cond, pthread_t *helper);
```

These two functions add or delete the `helper` thread to the pool of threads (empty after a `pthread_cond_init()` call) that can potentially inherit the priority of any thread waiting on the condition variable `cond` by means of a `pthread_cond_wait()` or `pthread_cond_timedwait()` call. The condition variable may be associated with a list of helper threads, and a kernel-level modification needs to ensure that the highest priority among the ones of all the waiters blocked on the condition variable is dynamically inherited by the registered helper thread(s), whenever higher than their own priority (and also that this inheritance is transitively propagated across both condition variables and traditional mutex supporting Priority Inheritance). Whenever the `pthread_cond_notify()` or `pthread_cond_broadcast()` function is called, the correspondingly woken-up thread(s) will revoke donation of their own priority.

A. Message queues

In a possible usage scenario, the proposed mechanism can be associated with a message queue in shared memory protected by a mutex for guaranteeing atomic operations on the queue, and a condition variable used to wait for the queue to become non-empty (if the queue has a predetermined maximum size, then another condition variable may similarly be used to wait for the queue to become non-full). In such a scenario, whenever initializing the condition variable, a writer task will declare itself as a writer associating its `pthread_t` to the condition variable, i.e., declaring explicitly that its execution will lead to the verification of the condition associated to that condition variable (non-empty queue). This can be done with a call to the above introduced `pthread_cond_helpers_add()` function after the condition variable initialization. Therefore, whenever a reader task will suspend its execution via a `pthread_cond_wait()` call on the condition variable, the associated writer(s), if there are any of them ready for execution, will dynamically inherit the priority of the suspended reader if higher than their own priority. This will inhibit third unrelated middle-priority tasks to preempt the low-priority writers, protecting from the mentioned Priority Inversion problem.

In a possible scenario in which there is a pipeline of multiple tasks using the just mentioned PI-CV-enhanced message queue

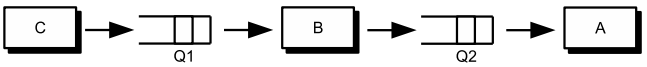


Figure 4. Pipeline interaction model.

implementation, it is possible to see the transitive inheritance propagation. Consider, for example, the scenario depicted in Figure 4, where A receives data from B through a message queue Q2, and B receives data from C through another message queue Q1. In such a case, when A attempts a read from Q1 but it suspends because it finds the queue empty, its priority may be donated to B. However, if B suspends on its own because it attempts a read from Q2 but it finds it empty, then C inherits not only the priority of B, but also the one of A (i.e., C runs with the highest priority – be it priority or deadline or other type of time-stamp – among A, B and C).

B. Client-server interactions

The described PI-CV mechanism may be leveraged to realize client-server interactions with the correct management of priorities whenever a server executes on behalf of a client with possibly other clients waiting for its service(s). In a possible implementation, clients and servers interact through message queues, synchronized through mutexes and condition variables. A server accepts requests from clients through a single server request queue. Each client may receive the desired reply from the server through a dedicated client-server reply queue. Each client may explicitly declare the server as the helper task for the condition variable associated to the client-server reply queue being non-empty. After posting a new request in the server request queue, a client suspends on the condition variable of its dedicated client-server reply queue. This allows the OS to automatically let the server inherit the maximum priority among (its own priority and) the priorities of any client waiting for its service(s).

Also, if the mutex protecting the message queues are all enhanced with traditional priority inheritance (e.g., the POSIX `PTHREAD_PRIO_INHERIT` attribute), the two mechanisms compose with each other towards reducing priority inversion.

Note that, in such scenario, it would be easy to provide a proper programming abstraction for client-server messaging that declares *implicitly* which are the helper tasks for the condition variables of the dedicated reply message queues. When dealing with real-time scheduling and the correct set-up of scheduling parameters, it is often convenient for developers if the Operating System or middleware services exhibit self-tuning capabilities [45], [46].

Effectiveness of PI-CV in the context of client-server interactions is further explored in Section VI, reporting a few simulation results. These have been obtained by means of the implementation of PI-CV described in what follows.

V. SIMULATION

The described PI-CV mechanism has been prototyped within the open-source RTSim real-time systems simulator[1]. RTSim [47] allows for simulation of a multi-processor system running a set of real-time tasks. Various scheduling mechanisms are available within the framework, including fixed priority, deadline-based scheduling and resource-reservation mechanisms [43], [44] (e.g., the POSIX Sporadic Server [3] or the Constant Bandwidth Server [21]). The simulated real-time tasks can be programmed with a simple language that includes, among others, instructions for simulating:

- computations for a fixed amount of time units (`fixed()` instruction), or for a probabilistically distributed time;
- basic locking instructions (`lock(M)` and `unlock(M)`) allowing for simulations of critical sections protected by a mutex `M`, corresponding to the POSIX `pthread_mutex_lock()` and `pthread_mutex_unlock()` functions.

The simulator also includes simulation of a few protocols for shared resources, which can be associated with the locking primitives, such as priority ceiling, traditional priority inheritance on mutexes, BWI [22] and others.

RTSim has been extended with the following modifications:

- condition variables have been supported through a new `CondVar` object type that can be referenced in two new dedicated task statements: `wait(M,CV)` and `signal(M,CV)`, which act on the mutex `M` and CondVar `CV`, with semantics corresponding to the POSIX `pthread_cond_wait()` and `pthread_cond_signal()` function calls, respectively (note that, when a task is suspended via `wait()`, the mutex `M` is released, and that `signal()` wakes up only the highest-priority task among the waiters);
- a new `Counter` object type has been added, with the associated instructions `inc()`, `dec()` and `set()`, with obvious meaning;
- as RTSim lacks of conditional statements, a new `waitc(M,CV,SZ)` instruction has been added which suspends the calling task performing a `wait()` only if the specified counter is zero;
- the support for priority inheritance has been completed for the case of arbitrarily nested critical sections;
- the PI-CV mechanism as described above has been integrated, in a way that also integrates transitive inheritance among mutex lock and condition variable wait primitives.

Helper tasks for condition variables must be set-up statically before the simulation begins. Furthermore, the PI-CV and traditional priority inheritance mechanisms can be individually enabled or disabled for the whole simulation.

With such elements, it is possible to simulate for example the synchronization among two tasks due to one of them writing onto a shared queue and the other one waiting for reception of a message, using a counter `SZ` just to keep track of the queue size. To simulate a periodic or sporadic writer that computes for 6 time units, then it pushes a message onto a queue with an atomic operation lasting for 2 time units, then it waits for the next cycle, one would use the code in Figure 5 left. To simulate a reader/waiter that computes for 2 time units, then it waits for a message to be made available onto the queue, where checking the message availability takes 1 time unit, and extracting it from the queue takes 2 time units, then it completes the cycle with further 3 time units of computation, one would use the code in Figure 5 right.

```
fixed(6);
lock(M);
  fixed(2);
  inc(SZ);
unlock(M);
signal(M,CV);
```

```
fixed(2);
lock(M);
  fixed(1);
  waitc(M,CV,SZ);
  fixed(2);
  dec(SZ);
unlock(M);
fixed(3);
```

Figure 5. Task code for send (left) and receive (right) via a shared queue.

The above scheme can be used, for example, to reproduce the scenario in Figure 2.

Additional instructions have been introduced to deal with more dynamic behaviors, as required in a real client-server interaction, in which the server cannot know in advance what client it will receive a request from, thus it cannot know in advance which client queue it will have to push the answer into. To this purpose, the following further modifications have been realized in RTSim:

- a new `Queue` type has been added, abstracting a message queue functionality, in which no messages are actually exchanged by tasks, but RTSim remembers how many messages have been posted by means of the usual `push(Q)` and `pop(Q)` operations; also, the further operations `pushptr(Q,RQ,RCV,RM)` and `popptr(Q,P_RQ,P_RCV,P_RM)` are used for more complex client-server interactions, where a client can post into the queue information on which queue the reply should be directed to (see code below); the `Queue` type is purposely non-synchronized, so as to leave freedom to specify the synchronization by composing the other primitives as needed;
- a new `waitq(M,CV,Q)` operation waits for the specified queue to be non-empty, performing a `wait()` operation on the specified CV and releasing the specified mutex, if needed;
- a new `Pointer` type has been added, capable of pointing to mutex, condition variable and queue objects; whenever RTSim expects the name of any of said objects, the name of a pointer pointing to an object of the same type can be used instead, in dereferenced notation (using a "*" prefix); namely, the operation `lock(*pM)` unlocks the mutex that is referenced by the pointer `pM`; this is useful in combination with the `popptr` and `pushptr` functions, as clarified in the example below.

As in the original RTSim code base, there are no instructions to declare mutex, condition variable, queue and pointer objects in

[1]More information is available at: http://rtsim.sourceforge.net.

```
fixed(1);
lock(ServerM);
  fixed(2);
  pushptr(ServerQ,ClientQ,
          ClientCV,ClientM);
unlock(ServerM);
signal(ServerM,ServerCV);

fixed(1);
lock(ClientQ);
  fixed(2);
  waitq(ClientCV,ClientM,
        ClientQ);
  pop(ClientQ);
unlock(ClientM);
```

```
fixed(1);
lock(ServerM);
  fixed(2);
  waitq(ServerM,ServerCV,
        ServerQ);
  popptr(ServerQ,pClientQ,
         pClientCV,pClientM);
unlock(ServerM);

fixed(5);

fixed(1);
lock(*pClientM);
  fixed(2);
  push(*pClientQ);
unlock(*pClientM);
signal(*pClientM,*pClientCV);
```

Figure 6. Task code for client (left) and server (right) using PI-CV.

the tasks code, but these have to be created by using the RTSim API before adding code to the tasks. The above elements can be used to code a client-server interaction, as shown in Figure 6.

As it can be seen, the level of detail for the simulation may be kept to a minimum, neglecting details related to the functional aspects of the simulated tasks, but catching the main behavioral aspects that may impact their response-times.

The presented modifications to the RTSim open-source simulator have been submitted for clearance to be released in public and be freely made available to other researchers. However, at this time it is not clear whether this will be possible or not.

VI. Simulated Results

Using the implementation of PI-CV within RTSim as described in the previous section, an evaluation has been done by simulating a simple scenario with 3 client tasks using the same server task and running on a single-processor platform. For example, the server task might be representative of some OS service available through proper RPC calls, realized in terms of shared in-memory data structures protected by synchronizing access through mutexes and condition variables. In the simulated scenario, each client task is periodic, it spends a fixed time processing (see Table I), then it invokes the server by pushing a message onto the server receive queue, then it waits for a response to be placed by the server onto the client own receive queue. The server, on the other hand, is not periodic. It has been given the lowest possible priority within the system. It waits for an incoming message on its receive queue, then it computes for a fixed amount of time, then it pushes a message back onto the receive queue of the caller task, and it repeats forever. Task periods have been generated randomly. The overall experiment duration has been set to 200000 simulated time units, amounting to roughly 300 activations for each task. The overall set of used parameters for the 3 clients is summarized in Table I. The parameters have been roughly chosen to create a scenario in which the advantages of the proposed technique could be easily highlighted. Other overheads such as context switch or scheduling overheads have not been simulated. A more realistic simulation, including a careful tuning of the overheads

Parameter	Client1	Client2	Client3
Task period	676	683	687
Overhead of `lock()/unlock()`	1	1	1
Overhead of `wait()/signal()`	2	2	2
Overhead of `push()/pop()`	2	2	2
Overhead of `pushptr()/popptr()`	2	2	2
Job own computation	50	50	50
Server call computation	20		
Experiment duration	200000		

Table I
Task parameters for the simulated scenario (all values are expressed in the simulated time units).

and parameters around a real platform and OS and possibly a real application, is surely valuable future work to be done.

Client-server interactions have been simulated in RTSim following the code structure exemplified in the previous section making use of the `Queue` type and of the `Pointer` type for the server. Two simulations have been done, one with only the traditional priority inheritance on all mutexes, and the other one with also the PI-CV mechanism on all condition variables (the two mechanisms acted in an integrated fashion as explained above).

Figure 7.(a) reports the obtained Cumulative Distribution Functions (CDFs) of the response time (i.e., the difference between the job finishing and arrival times) of the highest and lowest priority clients for the experiment, in the two cases of with and without PI-CV. It is clearly visible that, when using PI-CV, the highest priority client greatly benefits of PI-CV, reducing its average and maximum response-times, at the expense of the lowest priority client for which both metrics become worse, as expected. As a result, PI-CV allows for avoiding unnecessary priority inversion. For completeness, Figures (b) and (c) report the CDFs for all the 3 clients in the two cases.

Figure 8 reports the cumulative simulated time units for which each task was assigned each priority value. Note that, in RTSim, the priority with numeric value 1 corresponds to the highest priority in the system. As it can be seen, the server task is assigned, for a significant part of the simulation, one of the clients priority levels, while it is serving requests on their behalf. Also, Client3 is assigned for a small time (note the logarithmic scale on the vertical axis) the boosted priority levels assigned to Client1 and Client2. This may be due to two factors. First, Client3 competes on the mutex protecting access to the server queue, thus whenever Client1 or Client2 wait for it to release the mutex before posting their message, the Client3 priority is correspondingly boosted to the level of Client1 or Client2 by the traditional priority inheritance mechanism. Second, whenever Client1 or Client2 submit a request to the server and start waiting for the response, but the server is still serving Client3, and no mutex is being held by any task, PI-CV boosts the priority of Client3 to the level of Client1 or Client2, depending on who is actually waiting on the condition variable.

It has to be noted that the effectiveness of PI-CV and its quantitative impact on the tasks performance depends essentially on how much time a task spends wait()-ing on a

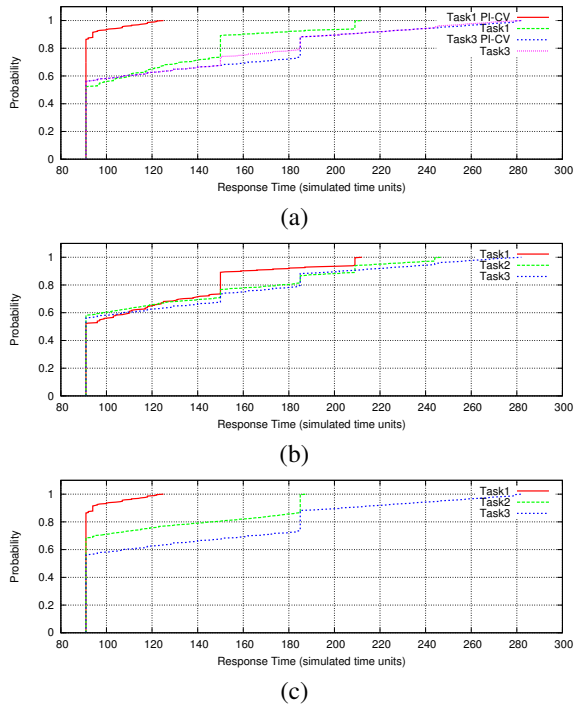


Figure 7. Response time CDFs of the response time of: (a) the highest-priority client (Task1) and the lowest-priority client (Task3) in the two cases of with and without PI-CV; (b) the 3 clients when executing without PI-CV and (c) with PI-CV.

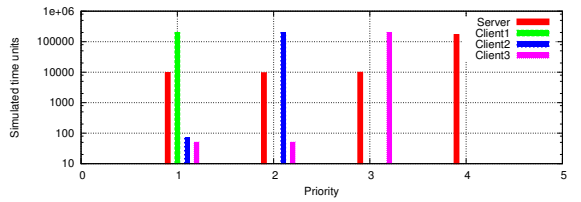


Figure 8. Cumulative simulated time units spent by each task into each priority value (1 is the highest priority, 4 is the lowest priority in the system).

condition variable for which helper tasks are defined, i.e., how much time is needed for the corresponding notify() to occur. This time is of course very application-specific. Comparing with traditional priority inheritance on mutex semaphores, in that case the effectiveness of the mechanism depends on how much time a task spends in a critical section with a mutex locked, which is *also* very application-specific. Though, the time spent with a mutex locked may be expected to be lower than the one spent wait()-ing for a notify() by some other task. Therefore, whenever it is possible to identify dependency relationships among real-time tasks, the presented PI-CV mechanism may be exploited to avoid situations of priority inversion expected to be of longer durations.

VII. Conclusions

In this paper, a mechanism has been presented for enhancing real-time systems with priority inheritance in presence of mutual exclusion semaphores and condition variables. The new mechanism, called PI-CV, alleviates the problem of priority inversion in cases in which developers code into the system custom, application-specific interaction and communication/synchronization logic by means of mutex and condition variables. With PI-CV, the programmer may declare what are the tasks that may help a condition to become true, over which other tasks may be waiting. Exploiting such dependency information, the run-time (Operating System) can correspondingly trigger the needed priority inheritance among tasks, mixing with traditional priority inheritance (e.g., as available through the `PTHREAD_PRIO_INHERIT` attribute in POSIX).

Whether or not it is meaningful that a higher-priority task suspends waiting for possible lower-priority tasks to provide some output, is something belonging to the application logic, and outside the scope of this paper. Whenever priorities of tasks can be meaningfully fine-tuned ahead of time, PI-CV might not be needed at all. However, PI-CV is applicable and useful in all those situations in which a task might need interactions with multiple tasks of different priorities (that go beyond the simple synchronization by mutual exclusion semaphores but need to recur to condition variables). This is a situation that might occur frequently in the design of real-time and embedded systems, for example for OS or middleware services that are shared across all real-time tasks within the system, as shown in the simulated scenario of Section VI.

Even though the proposed technique has been prototyped within the RTSim open-source simulator for real-time systems, possible future work includes the implementation of the proposed technique within a real OS (e.g., by extending the pthreads library and kernel functionality on Linux and integrating the technique with the SCHED_DEADLINE deadline-based scheduler [48]) showing its usefulness in concrete application contexts. For example, the Jack low-latency audio development framework allows for realizing arbitrary DAGs of inter-connected components and filters, in the audio processing pipeline. As the framework was already modified [49] for using a deadline-based scheduler, it would be interesting, in a multi-processor context, to leverage the PI-CV mechanism in order to let all the resource reservations involved in the audio processing pipeline automatically synchronize over (inherit) the common deadline of delivery of the audio frames to the speakers. Other directions for future work go of course along the direction of extending existing schedulability analysis techniques in presence of PI-CV. For example, the analysis presented in [39] might be extended and generalized for such purpose.

References

[1] L. Sha, R. Rajkumar, and J. P. Lehoczsky, "Priority inheritance protocols, an approach to real-time synchronization," Tech. Rep. CMU-CS-87-181, Carnegie-Mellon University, November 1987.

[2] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *Computers, IEEE Transactions on*, vol. 39, pp. 1175 –1185, sep 1990.

[3] *IEEE Std 1003.1-1990, IEEE Standard for Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) [C Language]*, 1990.

[4] U. Drepper and I. Molnar, "The Native POSIX Thread Library for Linux," tech. rep., Red Hat Inc., February 2001.

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

[6] D. Cornhilll, L. Sha, and J. P. Lehoczky, "Limitations of Ada for real-time scheduling," in *Proceedings of the first international workshop on Real-time Ada issues*, IRTAW '87, (New York), pp. 33–39, ACM, 1987.

[7] D. Cornhill and L. Sha, "Priority inversion in Ada," *Ada Lett.*, vol. VII, pp. 30–32, Nov. 1987.

[8] J. B. Goodenough and L. Sha, "The Priority Ceiling Protocol: A Method for Minimizing the Blocking of High-Priority Ada Tasks," Tech. Rep. CMU/SEI-88-SR-4, Carnegie-Mellon University, March 1988.

[9] M. W. Borger and R. Rajkumar, "Implementing Priority Inheritance Algorithms in an Ada Runtime System," Tech. Rep. CMU/SEI-89-TR-015, Carnegie Mellon University, April 1989.

[10] C. D. Locke and J. B. Goodenough, "A practical application of the ceiling protocol in a real-time system," in *Proceedings of the second international workshop on Real-time Ada issues*, IRTAW '88, (NY), pp. 35–38, ACM, 1988.

[11] L. Sha, R. Rajkumar, and J. Lehoczky, "A priority driven approach to real-time concurrency control," tech. rep., CMU, July 1988.

[12] R. Rajkumar, L. Sha, and J. Lehoczky, "Real-time synchronization protocols for multiprocessors," in *Real-Time Systems Symposium, 1988., Proceedings.*, pp. 259 –269, dec 1988.

[13] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," in *Proceedings of the International Conference on Distributed Computing Systems*, pp. 116–123, 1990.

[14] C.-M. Chen and S. K. Tripathi, "Multiprocessor priority ceiling based protocols," tech. rep., College Park, MD, USA, 1994.

[15] M.-I. Chen and K.-J. Lin, "Dynamic priority ceilings: a concurrency control protocol for rt systems," *RTSJ*, vol. 2, pp. 325–346, Oct. 1990.

[16] T. P. Baker, "Stack-based scheduling for realtime processes," *Real-Time Syst.*, vol. 3, pp. 67–99, Apr. 1991.

[17] P. Gai, G. Lipari, and M. D. Natale, "Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip," in *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, RTSS '01, (Washington, DC, USA), pp. 73–, IEEE Computer Society, 2001.

[18] K. Lakshmanan, D. d. Niz, and R. Rajkumar, "Coordinated task scheduling, allocation and synchronization on multiprocessors," in *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, RTSS '09, (Washington, DC, USA), pp. 469–478, IEEE Computer Society, 2009.

[19] D. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Alleviating Priority Inversion and Non-Determinism in Real-Time CORBA ORB Core Architectures," in *Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium*, RTAS '98, (Washington, DC, USA), pp. 92–, IEEE Computer Society, 1998.

[20] M. Di Natale and A. Meschi, "Guaranteeing end-to-end deadlines in distributed client-server applications," in *Real-Time Systems, 1998. Proceedings. 10th Euromicro Workshop on*, pp. 163 –171, jun 1998.

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

[22] G. Lamastra, G. Lipari, and L. Abeni, "A bandwidth inheritance algorithm for real-time task synchronization in open systems," in *Real-Time Systems Symposium, 2001. (RTSS 2001). Proceedings. 22nd IEEE*, pp. 151 – 160, dec. 2001.

[23] G. Lipari, G. Lamastra, and L. Abeni, "Task synchronization in reservation-based real-time systems," *IEEE Trans. Comput.*, vol. 53, pp. 1591–1601, Dec. 2004.

[24] 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.

[25] 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.

[26] D. Faggioli, G. Lipari, and T. Cucinotta, "The multiprocessor bandwidth inheritance protocol," in *Proc. of the 22nd Euromicro Conference on Real-Time Systems (ECRTS 2010)*, pp. 90–99, 2010.

[27] D. Faggioli, G. Lipari, and T. Cucinotta, "Analysis and implementation of the multiprocessor bandwidth inheritance protocol," *Real-Time Systems*, vol. 48, pp. 789–825, 2012. 10.1007/s11241-012-9162-0.

[28] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, "Litmus-rt: A testbed for empirically comparing real-time multiprocessor schedulers," in *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, RTSS '06, (Washington, DC, USA), pp. 111–126, IEEE Computer Society, 2006.

[29] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson, "A flexible real-time locking protocol for multiprocessors," in *Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on*, pp. 47 –56, aug. 2007.

[30] N. Guan, P. Ekberg, M. Stigge, and W. Yi, "Resource sharing protocols for real-time task graph systems," in *Proc. of the 23rd Euromicro Conference on Real-Time Systems*, (Porto, Portugal), July 2011.

[31] B. B. Brandenburg and J. H. Anderson, "Optimality results for multiprocessor real-time locking," in *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, RTSS '10, (Washington, DC, USA), pp. 49–60, IEEE Computer Society, 2010.

[32] M. Bertogna, N. Fisher, and S. Baruah, "Resource-sharing servers for open environments," *Industrial Informatics, IEEE Transactions on*, vol. 5, pp. 202 –219, aug. 2009.

[33] M. Behnam, I. Shin, T. Nolte, and M. Nolin, "Sirap: a synchronization protocol for hierarchical resource sharing real-time open systems," in *Proceedings of the 7th ACM and IEEE international conference on Embedded software*, 2007.

[34] R. I. Davis and A. Burns, "Resource sharing in hierarchical fixed priority pre-emptive systems," in *Proceedings of the IEEE Real-time Systems Symposium*, 2006.

[35] A. Easwaran and B. Andersson, "Resource sharing in global fixed-priority preemptive multiprocessor scheduling," in *Proceedings of IEEE Real-Time Systems Symposium*, 2009.

[36] G. Macariu, "Limited blocking resource sharing for global multiprocessor scheduling," in *Proc. of the 23rd Euromicro Conference on Real-Time Systems (ECRTS 2011)*, (Porto, Portugal), July 2011.

[37] M. M. van den Heuvel, R. J. Bril, and J. J. Lukkien, "Dependable Resource Sharing for Compositional Real-Time Systems," in *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*, pp. 153–163, IEEE, Aug. 2011.

[38] S. Xi, C. Li, C. Lu, , and C. Gill, "Limitations and solutions for real-time local inter-domain communication in xen," tech. rep., Oct 2012.

[39] L. Abeni and N. Manica, "Analysis of client/server interactions in a reservation-based system," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, (New York, NY, USA), pp. 1603–1609, ACM, 2013.

[40] D. Hart and D. Guniguntalay, "Requeue-pi: Making glibc condvars pi-aware," in *Proceedings of the Eleventh Real-Time Linux Workshop*, pp. 215–227, 2009.

[41] A. Dragojević *et al.*, "Why STM can be more than a research toy," *Commun. ACM*, vol. 54, pp. 70–77, Apr. 2011.

[42] J. Corbet, "CFS group scheduling." http://lwn.net/, July 2007.

[43] C. W. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves for multimedia operating systems," Tech. Rep. CMU-CS-93-157, Carnegie Mellon University, Pittsburgh, May 1993.

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

[45] T. Cucinotta, F. Checconi, L. Abeni, and L. Palopoli, "Self-tuning schedulers for legacy real-time applications," in *Proceedings of the 5^{th} European Conference on Computer Systems (Eurosys 2010)*, (Paris, France), European chapter of the ACM SIGOPS, April 2010.

[46] T. Cucinotta, L. Abeni, L. Palopoli, and F. Checconi, "The Wizard of OS: a Heartbeat for Legacy Multimedia Applications," in *Proceedings of the 7^{th} IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia 2009)*, (Grenoble, France), October 2009.

[47] L. Palopoli, G. Lipari, G. Lamastra, L. Abeni, G. Bolognini, and P. Ancilotti, "An object-oriented tool for simulating distributed real-time control systems," *Software: Practice and Experience*, vol. 32, no. 9, pp. 907–932, 2002.

[48] J. Lelli, D. Faggioli, T. Cucinotta, and G. Lipari, "An experimental comparison of different real-time schedulers on multicore systems," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2405 – 2416, 2012. Automated Software Evolution.

[49] T. Cucinotta, D. Faggioli, and G. Bagnoli, "Low-latency audio on linux by means of real-time scheduling," in *Proceedings of the Linux Audio Conference (LAC 2011)*, (Maynooth, Ireland), May 2011.

Deterministic Fast User Space Synchronization

Alexander Züpke
RheinMain University of Applied Sciences, Wiesbaden, Germany
SYSGO AG, Klein-Winternheim, Germany
Email: alexander.zuepke@{hs-rm.de,sysgo.com}

***Abstract*—The Fast User Space Mutex (Futex) mechanism in Linux is a lightweight way to implement thread synchronization objects which handle the uncontended case in user space and rely on the operating system kernel only for suspension and wakeup on lock contention. However, the implementation in Linux today has certain drawbacks that make it unsuitable for use in hard real-time or mixed-criticality systems.**

This paper addresses these issues and presents a novel approach for a futex implementation that guarantees bounded worst case execution time (WCET) in all operations and increases the required level of determinism for safety critical applications. The presented approach of blocking mutexes and condition variables for an unbounded number of threads has linear memory usage and does not require a fine granular in-kernel memory allocator, thus being suitable for real-time operating systems on hardware platforms with low memory or partitioning constraints.

I. Introduction

This paper discusses *fast and lightweight* user space synchronization objects for systems with real-time constraints. We describe a robust and deterministic implementation of *mutexes* and *condition variables*. While mutexes are used to enforce *mutual exclusion* to guarantee exclusive access in resource sharing, condition variables provide a concept for suspension and wakeup-on-notification for producer-consumer problems. Unlike semaphores, which can handle both aspects by the same mechanism, mutexes and condition variables have clearly defined semantics for ordering in the POSIX API [1].

We use a two-tier approach for mutexes and condition variables: The first stage handles the uncontended case and relies on atomic operations in user space. The second stage covers the contended case and uses syscalls (system calls) into the operating system kernel for suspension and wake up. The presented approach is optimized for *best case* usage by omitting expensive syscalls on mutexes with low contention. Similar approaches where the kernel is entered only on contention are used by the *Futex* concept in Linux [2], in Microsoft Windows *Critical Sections* [3], and *Benaphores* in BeOS [4]. As our approach bases upon the Linux concept, we compare both concepts and explain the differences required for determinism.

The novelty and difference of the presented approach in comparison to other existing approaches lies in the way how suspension is handled in the operating system kernel: the kernel does not need to maintain an additional kernel object accompanying the user space object. Removing the kernel object also prevents possible out-of-memory errors on allocation of such objects, or resource shortage by overallocation.

The intended usage scenario covers real-time kernels, space- and time-partitioning, and mixed-criticality environments. A general safety requirement in such systems is, that an abuse of APIs in one partition must not interfere with a correct use other partitions, neither in a temporal sense that the WCET of unrelated partitions becomes indeterminable, nor in a spatial sense that it is possible to disturb operations of unrelated partitions, e.g. by exhausting kernel resources. The intended use case focuses on resource sharing inside a *single* partition utilizing one or more processors. The presented mutexes and condition variables are clearly *not* designed for resource sharing across different partitions or criticality levels in a mixed-criticality system, as they currently cannot handle priority inversion. Both mutexes and condition variables provide *fair* FIFO ordering.

We use the following terminology: a *process* is an instance of a computer program executing in an *address space*. The process comprises one or more *threads*, which are known by the operating system kernel[1] and can be independently scheduled on the processors assigned to the process at the same time. Multiple processes[2] have their own distinct address space each. Processes can share parts of their address spaces with others by using *shared memory*; a shared memory segment is usually *mapped* at different virtual addresses in each address space. A *waiting* thread suspends execution in the scheduler until the thread is *woken up* again on a predefined condition. In this paper, we do not use the terms *job* or *task* which have a different meaning in the field of real-time scheduling analysis. Also, we make no further assumption on scheduling algorithms or thread priorities.

The rest of this paper is organized as follows: Section II explains the futex mechanism in Linux and identifies problems that prevent deterministic use in real-time systems, leading to the requirements described in section III where we present our approach. Section IV discusses mutex and condition variable protocols. We discuss our approach in section V and conclude in VI.

II. Fast User Space Mutexes

A. Linux Implementation

The futex implementation in Linux [2] [5] came with the Native POSIX Thread Library (NPTL) [6] and allows to implement various POSIX-compliant high level synchronization objects such as mutexes, condition variables, semaphores, or readers/writer locks with low overhead in the system's C library in user space. One design goal was to reduce the syscall overhead for these locking objects when possible, thus the implementation uses atomic modifications on user space variables to handle uncontended locking and unlocking solely in user space, and a generic system call-based mechanism to suspend and wake threads in the kernel on lock contention. Basically, a futex is a 32-bit variable in user space, representing a certain type of lock (mutex, condition variable, semaphore) and its value is modified by a type-specific locking protocol.

[1] In contrast to *user-level* threading.

[2] In partitioned environments, a partition consists of one or more processes.

We give a short example of a simple mutex protocol on an integer variable representing the futex: let bit 0 represent the locked state of the mutex, and let bit 1 signal contention. Both bits cleared represents a free mutex. A thread can lock the mutex by atomically changing the lock value from 0x0 to 0x1 using a *Compare-and-Swap (CAS)* or *Load-Linked/Store-Conditional (LL/SC)* operation. A lock operation on an already locked mutex atomically sets bit 1 in the futex to indicate contention, then invokes the `FUTEX_WAIT` syscall to suspend the caller until the lock becomes available again. Symmetrically, when the current lock-holder sees contention during an unlock operation, it clears the locked bit and calls the `FUTEX_WAKE` syscall to wake the first waiting thread that then can acquire the lock itself by atomically setting bit 0 again.

During suspension, the futex syscall allocates an in-kernel futex object and stores the object reference in a hash table indexed by the address of the futex. The futex kernel object comprises the futex address, type, and a queue of waiting threads. The syscall enqueues further suspending threads on the same futex in the existing wait queue that matches the address and type information stored in the futex object. The futex kernel object is freed when the last waiting thread was woken up and the wait queue is empty again.

The third operation on futexes is `FUTEX_CMP_REQUEUE`. It is a special wait queue reassignment method which prevents *thundering herd effects* [2] on signaling of condition variables: instead of waking all threads and letting them compete to lock an associated mutex, the syscall transfers waiting threads from the condition variable's wait queue to the mutex' one. Linux additionally supports *priority inheritance* mutexes and condition variables for threads using POSIX real-time scheduling [7]. Finally, *robust futexes* provide a lightweight mean to notify pending waiters on a crash or deletion of the lock holder.

As futex wait queues are created on demand, Linux imposes no restrictions on the number of user space variables used for futexes. The kernel objects are created on contention only, and therefore the number of kernel objects is limited by the number of threads in the system (when all threads wait on a distinct futex) and by the available kernel memory.

B. Identified Problems and Possible Remedies

The key idea of allowing an unbounded number of futexes in user space and doing kernel operations only on contention appears to be reasonable also for real-time systems. However, the Linux implementation has certain drawbacks that hinder use in deterministic real-time systems:

1) Memory allocations of futex kernel objects require a fine granular in-kernel memory allocator. However, such memory allocations cause *fragmentation*, which is especially bad in space-partitioned environments, where kernel memory resources are limited.
2) Memory allocations can fail due to limited kernel memory. This leads to an extra burden for user applications to handle these kind of failures.
3) User space code controls the addresses of the futex variables which may cause hash collisions.
4) Operations on the wait queue should have deterministic timing for WCET analysis. This is especially important for priority-sorted wait queues. The Linux implementation [8] uses *priority-sorted linked lists* for priority inheritance mutexes and distinguishes 100 priority levels.

In all cases, the number of futex kernel objects is limited by the number of threads that can wait on a futex, which could be all threads in the system. Possible solutions (or reasonable compromises) for these issues depend on the usage scenario: if a fine granular memory allocator is available or out-of-memory failures are acceptable, issues 1 and 2 pose no problem. Alternatively, this could be solved by pre-allocation of the kernel objects, however this pessimistic approach wastes memory for futexes that never see contention. Also, moving the problems of memory allocations to other code sections of an application may not be useful in legacy, non real-time applications. Issues 3 and 4 can also be solved by limiting the number of threads that can utilize the futexes, or by limiting the number of waiting threads at a time, or by limiting the available priority levels. Removing the in-kernel memory allocations promises to solve the first three issues.

III. Deterministic Approach

Here, we present a novel approach without the need for a fine granular in-kernel memory allocator which can handle an unbounded number of threads. Based upon the previous discussion, we first define requirements for a reliable implementation, give an explanation of data structures and operations, and show how to handle wait queues without kernel objects.

A. Requirements

Besides obvious *functional* requirements regarding correct operation of mutexes and condition variables, an implementation must be *robust*, *deterministic* and support *partitioning*:

1) For mutexes, we define the user space operations `mutex_lock` and `mutex_unlock` to acquire and release a mutex lock of type `mutex_t`. For condition variables, `cond_wait` suspends the calling thread waiting on a condition variable of type `cond_t` until another thread wakes either one or all waiting threads by `cond_wake`. While calling a condition variable function, the caller is required to have a *support mutex* locked. This support mutex is released during suspension in `cond_wait`.
2) Support for an unbounded number of these user space objects. The number of objects is limited by the amount of available user memory.
3) No use of an in-kernel memory allocator. All required data in the kernel must be either kept in static *per partition* memory or *per thread* in TCBs (thread control blocks).
4) For robustness, the kernel must correctly handle invalid, unaligned, or unmapped address ranges passed in as references to user space objects.
5) The kernel must not expose loops with CAS or LL/SC operations on user space variables. Otherwise user code could, by updating a variable at the same time, force the kernel into unbounded retries to complete an operation.
6) Bounded operations on wait queues: *enqueue* (append at the end), *dequeue* (remove first) and *requeue* (append one queue to another) operations must be bounded in time, if possible with constant timing characteristics.
7) The kernel must be able to remove a given arbitrary thread from its wait queue in bounded time when handling timeouts, POSIX signals or thread deletion.
8) The lock *scope* defines the grade of required partitioning. A lock with *process local scope* can only be used in its defining process. Further, such locks must not interfere with locks of other unrelated processes. Locks with *global*

scope can be shared between processes[3].

9) For locks with different scope, internal locking of the wait queues must not interfere.

Of the listed requirements, Linux fulfills all objectives except 3 and 9. In the latter case, Linux uses a single hash for all processes. Due to hash collision, a process can delay unrelated ones. Additionally, the priority inheritance versions of Linux mutexes and condition variables violate objectives 5 and 6 by requiring CAS operations in the kernel and using a potentially long running sorting mechanism for the wait queue.

B. Operations and Data Structures

Without an in-kernel memory allocator, the required data to support locking objects in user space must be kept in user space or in the TCB. Wait queues need to be created on demand, so the key idea is to place a token identifying the associated wait queue into user space as well. For syscalls and other functions implemented in the kernel, we use names in capital letters. These operations and data structures are:

1) We define five basic kernel operations that closely resemble their Linux counterparts:
   - `MUTEX_WAIT`: suspend the current thread while waiting to acquire a mutex
   - `MUTEX_WAKE`: wake the next suitable thread from a mutex wait queue
   - `COND_WAIT`: suspend the current thread while waiting on a condition variable
   - `COND_REQUEUE`: move one or all threads waiting on a condition variable to the according mutex wait queue
   - `DEQUEUE`: remove a waiting thread from its mutex or condition variable wait queue

   The first four operations are used by the `mutex_lock`, `mutex_unlock`, `cond_wait`, `cond_wake` operations respectively. The fifth operation (`DEQUEUE`) is used internally by the kernel on timeout expiry, thread deletion, or other waking activities. This limited subset suspends or wakes a single thread at a time. The operations on wait queues are similarly restricted to enqueueing and removal of single threads, or to append complete wait queues to one another.
2) We place the *thread ID* of the head of the queue into user space, next to the futex variable, and a flag indicating if the thread is a legitimate head of a wait queue into TCB. The wait queue itself is kept in kernel space as we keep the internal pointers in the TCB as well. An *indexing* approach with an explicit *ID* identifying the wait queue instead would have required an allocation mechanism again.
3) In the TCB we further store information describing the lock object, i.e. the address in user space, type and scope.
4) We use the scope and the hashed address of the user space object to locate a suitable lock of the wait queue, as described in the following section.
5) For mutexes, we also encode the thread ID of the current lock holder in the futex value in user space.
6) For now, we only provide FIFO ordering of the waiters by using *doubly linked lists*.

By placing data like this, we do not need dynamic allocation of kernel objects. We explain the exact encoding of the current lock holder and wait queue head in section IV.

[3]For locks shared between processes of a single partition, an additional *partition local scope* would be required.

C. Locking of the Wait Queues

The in-kernel wait queues require internal locking during modifications. Skipping the discussion of trivial approaches like disabling interrupts on single processor systems or use of a single global kernel lock on multi processor systems, we must use a scope specific object and/or the address of a mutex or condition variable object as *key* to lock the wait queue.

Depending on the scope of the futex, per-process, per-partition, or global lock objects can be used to protect the wait queues. Per-process and per-partition locks ensure that futex operations of concurrently executed processes do not interfere with each other. For mixed-criticality systems, access to locks of higher critical partitions should be privileged.

Using the futex address as key, multiple locks could be used. The virtual address is sufficient to uniquely identify a futex in a single process.[4] For locking scopes beyond a single process, the physical address must be used. Eventually, hashing of the address leads to the right lock in a scope specific array.

We assume that these locks are located in scope specific data storage, e.g. process descriptor or global data, and exist during the lifetime of the scope. The actual number of hashed locks is a trade-off between scalability and memory usage. The choice depends on the targeted system environment.

IV. Locking Protocols

We now describe the necessary data structures and state transition protocols of a futex variable representing a mutex and a condition variable. We omit all error checking except for the sequences where we must retry an operation. APIs are reduced to the minimum necessary to describe the concept.

We assume the following scenario: Firstly, the target architecture has 32-bit atomic CAS or LL/SC instructions. Secondly, all threads can be referred by unique IDs of *less* than 32-bit, with the value zero denoting an invalid thread ID. Lastly, we limit the scope to a single process only, so that virtual addresses are used for locking of the wait queues[5].

A. Mutex Protocol

A mutex in user space comprises elements $\langle T, W, Q\rangle$ in two consecutive 32-bit integers, see figure 1. Let the *lock state* S be the first integer encoding T and W: The *waiters bit* W is a single bit indicating whether or not the wait queue is empty. The *lock holder's* thread ID T is encoded in the remaining bits. The second integer points to the *wait queue head* and holds the thread ID Q of the oldest thread in the wait queue or zero if the wait queue is empty, see step c in figure 2.

We use the notation $S = \langle W, T\rangle$ to describe the state of the mutex, and $Q = \{...\}$ to describe the wait queue as FIFO-ordered set, pointed to by the thread ID of the left-most thread on the queue. The mutex is *free without waiters* if both the values of S and Q are zero: $S = \langle 0, 0\rangle \wedge Q = \emptyset$, which is also the initial state of a mutex. Further, we let l, l' and l'' describe threads attempting to acquire the lock or suspended waiting on the lock. When used in conjunction with S or Q, l, l' and l'' describe the respective thread IDs. The rules and invariants to access S, Q, and the internal wait queue are:

1) The user changes only S using CAS.
2) The kernel accesses S, Q, and the wait queue only while holding the according wait queue lock.

[4]We neglect the case of mappings of the same physical memory to different virtual addresses. This will be detected and result in an address mismatch error.

[5]For shared locks, the lock object's physical address should be used.

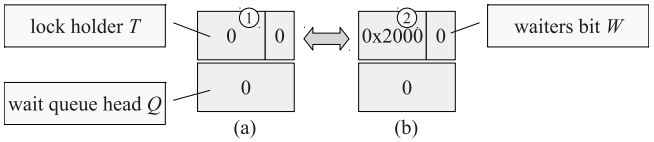


Fig. 1: Example of mutex state changes in the uncontended case: `mutex_lock` and `mutex_unlock` change T atomically, 0 denotes a free mutex and $0x2000$ refers to a lock holder's thread ID.

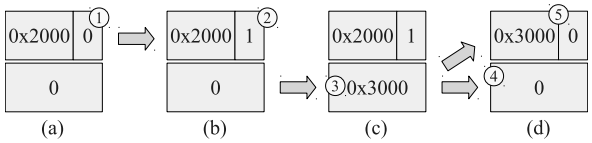


Fig. 2: Example of mutex state changes in the contended case: on contention, `mutex_lock` sets W in step b and calls the kernel. `MUTEX_WAIT` enqueues the calling thread $0x3000$ and updates Q in step c. A later `mutex_unlock` by thread $0x2000$ sees contention and let `MUTEX_WAKE` put in the first waiter as new lock holder in step d. As $0x3000$ was the only waiter, the wait queue becomes empty, Q and W are set to zero.

3) The user atomically sets T in S the first time on contention.
4) The kernel clears T if the wait queue is empty and there is no more contention.
5) If Q is zero, the wait queue is empty.
6) If Q is non-zero, it points to the head of the wait queue.
7) If the head of the wait queue changes, the kernel updates Q to the new head's thread ID.
8) If the wait queue becomes empty, the kernel sets Q to zero.
9) On suspension, the kernel adds the waiting thread to the wait queue (or creates a new one if it was empty before).
10) On wake up, the kernel removes the first thread from the wait queue.

In detail, the operations for mutexes are:

1) On an uncontended lock operation by thread l, `mutex_lock` atomically changes S from $\langle 0, 0\rangle$ to $\langle l, 0\rangle$. An uncontended unlock operation reverses this, see figure 1. The content of Q is ignored.
2) On a contended lock operation (non-zero S) by l', `mutex_lock` atomically sets the W bit in S if it is not yet set ($S = \langle l, 1\rangle$) before suspending in the kernel using `MUTEX_WAIT`, see step b in figure 2. It further passes the current value of S to the kernel as S'.
3) `MUTEX_WAIT`: After locking the wait queue of the mutex, the call checks if S' equals S in user space. This prevents *lost wakeup* errors, because the update of S and the syscall `MUTEX_WAIT` are not atomic.
   Before enqueueing l', the kernel reads Q first:
   - If Q is zero, the wait queue is empty and the kernel sets Q to l', creating a new wait queue, see figure 2, step c.
   - If Q is non-zero, the kernel tries to find the referenced thread and compares the mutex attributes to check if the referenced thread is an eligible wait queue head and waiting on the same mutex. It then inserts l' into the existing wait queue.

   Finally, the kernel releases the wait queue lock and suspends the calling thread in the scheduler.
4) On a contended (W set) unlock operation by l, `mutex_unlock`() relies on `MUTEX_WAKE` to wake the next waiting thread and pass the lock over to it.
5) `MUTEX_WAKE` locks the wait queue first, then checks Q:
   - If Q is zero, the wait queue is empty. Then it sets S to zero and the operation completes. This describes the race that leads to the lost wakeup error.
   - If Q is not zero, the call checks the wait queue; the denoted thread must be an eligible wait queue head and point to the same mutex.

   The kernel then removes the first thread from the queue to become the next lock holder. The call updates Q with the ID of the next thread in the wait queue, or zero if the queue is empty. Then it sets T to the ID of the new lock holder, with W set accordingly, see figure 2, step d. Finally, it unlocks the wait queue and wakes up the new lock holder.

It is necessary to let the kernel update S and Q consistently without using CAS, otherwise user code could force the kernel into endless CAS retries. The parts in user space retry their operation if either the CAS fails or the comparison of S' and S in the kernel fails. The shown implementation is a basic version only. A non-suspending `mutex_trylock` operation can be deduced from the first step. It is possible to busy-wait in `mutex_lock` for a certain time before suspending in the kernel or to add timeouts to the call.

B. Condition Variable Protocol

The condition variable protocol is similar to the mutex protocol. A condition variable comprises elements $\langle C, Q'\rangle$ in two consecutive integer variables. The *condition counter* C in the first integer is a single counter incremented on every wakeup operation. The second integer Q' describes a *wait queue head* again and identifies the longest waiting thread in the queue. To describe the mutex associated with the condition variable, we use $\langle S, Q\rangle$ again. The initial state of a condition variable is $C = 0 \wedge Q' = \emptyset$. For C and Q', the same rules apply as described for S and Q. The operations on condition variables are:

1) `cond_wait` is used to wait on the condition variable. The call reads C and provides it to `COND_WAIT` as C' to detect lost wakeup errors. Before calling the kernel, `cond_wait` releases the accompanying mutex.
2) `COND_WAIT` performs similar steps than `MUTEX_WAIT`: it locks the wait queue, compares C' with C in user space, enqueues the calling thread in the wait queue, and suspends the calling thread. Additionally, the call keeps a reference to the mutex for later `COND_REQUEUE`.
3) `cond_wake` wakes one or all threads. It first increments C and then calls `COND_REQUEUE`. As the caller is required to hold the mutex while calling `cond_wake`, the update of C is atomic to others.
4) `COND_REQUEUE` locks the wait queue of the condition variable and checks Q' as shown in figure 3:
   - If Q' is zero, the call bails out, as no threads are waiting.
   - Otherwise it removes either one or all threads from the wait queue (step 3). When all threads are to be migrated, the syscall keeps the complete queue as is and appends it at the end of the mutex wait queue later.

   After updating Q' accordingly, `COND_REQUEUE` unlocks the first wait queue. Then, by the reference to the mutex that was kept in the TCB of the waiters, the syscall locks the mutex wait queue (step 4), checks, appends previously

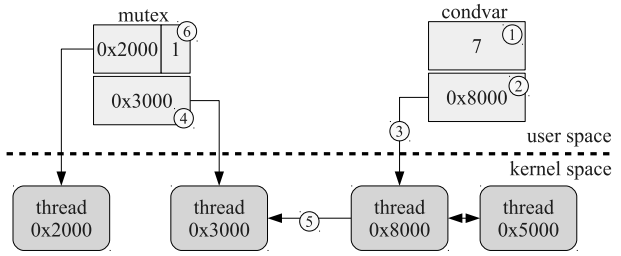


Fig. 3: Example of a COND_REQUEUE migrating all threads: The call disconnects the condition variable's wait queue (starting with thread $0x8000$) and sets Q' to zero. Then it attaches the wait queue to the one of the mutex with thread $0x3000$ as head.

removed waiters (step 5), and updates Q in user space if necessary. Additionally, the waiters bit S is set if necessary before unlocking.

It is also possible to design COND_REQUEUE in such a way that the caller is not required to hold the mutex. Then the first waiter is made the lock holder of the mutex S if it is currently free. However, this would require a CAS operation on S which could be forced into unbounded retries by user space code.

The wait operation exposes a race condition between the time it unlocks the mutex in user space and the time the kernel checks C. Lost wakeup issues are normally handled by the kernel comparing C and C', but if during that time exactly 2^{32} wakeup operations are performed, C overflows to exactly the same value and the check succeeds. Developers must be aware of this *ABA problem* [9].

C. Dequeue Operation

A generic *remove from wait queue* operation is required when waiting must be interrupted, for example when a timeout expires or the thread is going to be deleted. On DEQUEUE, the kernel does not know if the thread is currently suspended on a condition variable or mutex wait queue. Therefore it locks both wait queues at the same time. This can be done safely when both user space objects refer to different addresses and locks are taken in order of ascending addresses. After removing the thread from the lock, DEQUEUE checks if the thread was referenced in Q or Q' and updates Q or Q' accordingly. If Q becomes zero, the call clears the waiters bit in S as well.

V. Discussion

Here we discuss robustness and determinism aspects of our approach. Basically, we have to consider that user code may accidentally or deliberately manipulate either S, C, Q or Q'.

Until now, the encoding of the current lock holder in S is just of informative nature and has no impact on the operation of the kernel. More elaborate versions of the mutex user space implementation may implement deadlock detection or a recursion counter and may depend on S being correct, but the problem is solely a problem of user space. The same applies to C. The values of Q and Q' are of more importance to the kernel. If these are changed, the kernel cannot find the wait queues any more. However, the wait queue itself is kept in kernel space and therefore its integrity is not affected.

The kernel is robust against the following errors:

1) Q or Q' are set to zero. Threads can no longer be woken up on calls to mutex_unlock or cond_wake. But the kernel always allows to cancel the waiting operation by other means using DEQUEUE.
2) Q or Q' are set to another thread currently waiting in the queue. The kernel detects this by checking if the thread is an eligible wait queue head. For FIFO queueing, the check could be omitted, which then just affects the order in which threads are woken up. Otherwise it would take an unacceptable time of $O(n)$ to locate the original head.
3) Q or Q' are set to threads currently waiting in different wait queues, or threads not in waiting state, or invalid thread IDs. Again the kernel detects such conditions.
4) Q or Q' are set to threads out of the scope of the lock. The kernel can detect such errors by other means (e.g. communication capabilities, not shown).

We can consider that all these errors have the same impact, namely than an application may not release a locked mutex. The fault remains isolated in the application and can not harm the kernel or violate partitioning.

From a temporal point of view, none of the operations requires searching or exposes unbounded behavior. All loops are bounded to $O(1)$ for WCET analysis.

VI. Conclusion and Outlook

We have shown a novel approach to implement an unbounded number of mutexes and condition variables using a two-tier approach, with user space handling the uncontended case and the kernel handling contention, and described the low level protocols to be followed. We have also shown how to implement futexes without a fine granular memory allocator.

The presented approach is a foundation to implement other synchronizations means like semaphores, reader-writer locks or barriers on top [5], or to extend the implementations with a safety net of additional error checking or convenience functionality like recursive mutexes. It is also possible to implement timeout handling in all waiting operations or busy-wait on a mutex for a certain time before suspending in the kernel.

In future work, we would like to add support for priority ordered wait queues in fixed-priority scheduling use cases. Also we like to discuss the practicability of the approach when using dynamic priority scheduling like EDF. Finally, we would like to evaluate means to prevent *priority inversion*. At least, a *priority inheritance protocol* seems implementable.

References

[1] IEEE, "POSIX.1-2008 / IEEE Std 1003.1-2008 real-time API," 2008.

[2] H. Franke, R. Russell, and M. Kirkwood, "Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux," in *Proceedings of the 2002 Ottawa Linux Symposium*, 2002, pp. 479–495.

[3] "Initializecriticalsection function." [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/ms683472%28v=vs.85%29.aspx

[4] B. Schillings, "Be engineering insights: Benaphores," *Be Newsletters*, vol. 1, no. 26, May 1996.

[5] U. Drepper, "Futexes are tricky," White Paper, Nov. 2011. [Online]. Available: http://people.redhat.com/drepper/futex.pdf

[6] U. Drepper and I. Molnar, "The native posix thread library for linux," Red Hat, Inc, Tech. Rep., Feb. 2003.

[7] D. Hart and D. Guniguntalay, "Requeue-pi: Making glibc condvars pi-aware," in *Eleventh Real-Time Linux Workshop*, 2009, pp. 215–227.

[8] "Lightweight pi-futexes." [Online]. Available: http://lxr.linux.no/#linux+v3.8.7/Documentation/pi-futex.txt

[9] M. M. Michael, "ABA prevention using single-word instructions," IBM Thomas J. Watson Research Center, Tech. Rep. RC23089, Jan. 2004.