Noname manuscript No.
(will be inserted by the editor)

Implementation and Evaluation of Global and Partitioned Scheduling in a Real-Time OS

Giovani Gracioli · Antônio Augusto Fröhlich · Rodolfo Pellizzoni · Sebastian Fischmeinster

Received: date / Accepted: date

Abstract In this work, we provide an experimental comparison between Global-EDF and Partitioned-EDF, considering the run-time overhead of a real-time operating system (RTOS). Recent works have confirmed that OS implementation aspects, such as the choice of scheduling data structures and interrupt handling mechanisms, impact real-time schedulability as much as scheduling theoretic aspects. However, these studies used real-time patches applied into a general-purpose OS. By measuring the run-time overhead of an RTOS designed from scratch, we show how close the schedulability ratio of task sets is to the theoretical hard real-time schedulability tests. Moreover, we show how a well-designed object-oriented RTOS allows code reuse of scheduling components (e.g., thread, scheduling criteria, and schedulers) and easy real-time scheduling extensions. We compare our RTOS to a real-time patch for Linux in terms of the task set schedulability ratio of several generated task sets. In some cases, Global-EDF considering the overhead of the RTOS is superior to Partitioned-EDF considering the overhead of the patched Linux, which clearly shows how different OSs impact hard real-time schedulers.

Keywords Real-time scheduling · multicore processors · real-time operating systems · global EDF · partitioned EDF

G. Gracioli and A. A. Fröhlich
Federal University of Santa Catarina, Florianópolis, Brazil
Phone/Fax: +55 (48) 3721-9516
E-mail: {giovani,guto}@lisha.ufsc.br

R. Pellizzoni and S. Fischmeinster
University of Waterloo, Waterloo, Canada
Phone/Fax: +1 (519) 888-4567 / +1 (519) 746-3077
E-mail: {rpellizz,sfischme}@uwaterloo.ca

1 Introduction

Multicore processors are being increasingly used in real-time system domains due to the evolution and integration of features and consequently the need for more processing power. In automotive environments, for instance, new safety functionalities like "automatic emergency breaking" and "night view assist" should read and fusion data from sensors, process the video, and give warnings when an obstacle is detected on the road under real-time constraints (Mohan et al, 2011). In addition, adding more functionalities to the system costs in terms of power consumption, heat dissipation, and space (e.g., cables) (Cullmann et al, 2010). Thus, multicore processors become an alternative to decrease these costs and integrate features in a single processing unit, instead of several electronic control units (ECUs) spread over the vehicle. Especially for hard real-time (HRT) systems, in which deadlines must be always met, the use of a well-designed real-time operating system (RTOS) and multicore real-time schedulers are crucial to provide HRT guarantees.

Traditional real-time multicore scheduling can be classified in two categories: global and partitioned-based schedulers (Carpenter et al, 2004). In this context, Global Earliest-Deadline-First (G-EDF) and Partitioned-EDF (P-EDF) are examples of each category. Although suboptimal, G-EDF has less frequent preemptions and migrations and consequently, less run-time overhead, than optimal global schedulers (Brandenburg et al, 2008), such as proportional fairness-based schedulers (Baruah et al, 1996; Srinivasan et al, 2003; Levin et al, 2010). Furthermore, optimal multicore real-time schedulers are difficult to implement in practice, and G-EDF proved to have reasonable run-time overhead for a moderate number of processors[1] (Bastoni et al, 2010b), as those processors found in the embedded HRT domain. Also, G-EDF is interesting for mixed-criticality systems, where applications with different criticality levels co-exist in the same system, because it ensures tardiness bounds for soft real-time (SRT) applications as long as the total system utilization is at most m, which m is the number of processors (Leontyev and Anderson, 2007). P-EDF, on the other hand, has a limitation due to similarity to the bin-packing problem: heavy utilization tasks[2] strongly affect task partitioning heuristics. P-EDF is superior to G-EDF for HRT scenarios, mainly because the G-EDF schedulability tests are too pessimistic (Bastoni et al, 2010b).

Additionally, considering the RTOS point of view, implementation issues, such as scheduling data structures and interrupt handling, impact schedulability as much as scheduling theoretic issues (Brandenburg and Anderson, 2009; Bastoni et al, 2010b). In general, all the recent works that have measured the influence of run-time overhead in SRT and HRT multicore schedulers used a Linux-based infrastructure to support their studies (Brandenburg and Anderson, 2009; Bastoni et al, 2010b; Lelli et al, 2012). Despite being a good development platform, real-time Linux-based studies suffer from the inherent

[1] In this paper we interchangeably use the terms processor and core.
[2] A task that has a utilization higher than 0.5 is considered a heavy task.

non real-time behavior of Linux, which affect the run-time overhead observed in these works. This leads us to a few questions: what is the difference between Linux-based real-time patches and an RTOS designed from scratch in terms of run-time overhead? Is this difference significant for HRT applications? What is the influence of this run-time overhead when incorporated into G-EDF and P-EDF schedulability analyses?

Aiming at investigating these questions, we compare the Embedded Parallel Operating System (EPOS) (Fröhlich, 2001; EPOS, 2012) to LITMUS^{RT} (Calandrino et al, 2006) in terms of run-time overhead and the impact of both OSes on the schedulability ratio of generated task sets for G-EDF and P-EDF schedulers. We measure the run-time overhead of EPOS and LITMUS^{RT} using a modern 8-core processor. In addition, we extend the run-time overhead analysis and compare the ideal (i.e., without overhead) G-EDF and P-EDF using eight state-of-the-art G-EDF schedulability tests and three P-EDF partitioning heuristics up to 100 processors for HRT systems.

In summary, the main contributions of this paper are:

- We show how a well-designed object-oriented component-based RTOS (EPOS) allows code reuse of system components (e.g., scheduler, thread, semaphore, etc) and easy global and partitioned real-time scheduling extensions. To the best of our knowledge, EPOS is the first open-source RTOS that supports global schedulers. We believe that EPOS can be used to conduct research for multicore HRT related areas due to higher predictability and smaller overhead compared to real-time patches for Linux.
- We show that the RTOS run-time overhead, when incorporated into G-EDF and P-EDF schedulability tests, can provide HRT guarantees close to the theoretical schedulability tests. Moreover, in some cases, G-EDF considering the overhead in EPOS is superior to P-EDF considering the overhead in LITMUS^{RT}, contradicting the theoretical tests.
- A comparison in terms of task set schedulability ratio between P-EDF and G-EDF, considering also the OS overhead, for HRT tasks. P-EDF has obtained the same or better performance than G-EDF for all analyzed scenarios. In our experiments, P-EDF and G-EDF had the same behavior for task sets composed only of heavy tasks, mainly because of G-EDF's schedulability test bounds. We observed a slightly improvement in G-EDF for this heavy tasks scenario compared to related work (Calandrino et al, 2006), due to the use of up to date G-EDF schedulability tests (Bertogna and Cirinei, 2007; Baruah et al, 2009).
- We measure cache-related preemption and migration delay (CPMD) on a modern 8-core processor, with shared level-3 cache, using hardware performance counters. We use the obtained values to compare P-EDF and G-EDF through the weighted schedulability metric (Bastoni et al, 2010a).

The rest of this article is organized as follows. Section 2 discusses related work. Section 3 defines the task model and presents the main concepts used in this paper. Section 4 presents in details the real-time support on EPOS. Section 5 shows the evaluation between G-EDF and P-EDF as well as the

influence of run-time overhead into schedulability of both algorithms. Section 6 discusses the main findings. Finally, Section 7 concludes the article.

2 Related Work

In this section we discuss related work. We organize the discussion in three topics: multicore real-time scheduling, run-time overhead and implementation tradeoffs, and cache-related preemption and migration delay. For each topic we present the main works available in the literature.

2.1 Multicore Real-time Scheduling

Recently, several scheduling algorithms have been proposed to provide real-time guarantees for multicore applications. They are usually either global or partitioned approaches. Considering the P-EDF algorithm, some works propose new task partitioning, splitting, or admission control techniques (Kato and Yamasaki, 2007; Masrur et al, 2010; Burns et al, 2012). Additionally, semi-partitioning algorithms combine characteristics from partitioning and global scheduling: they allow few tasks to migrate between the processors to improve the system utilization (Anderson et al, 2005; Kato and Yamasaki, 2008; Bletsas and Andersson, 2009). Other works compare different real-time schedulability tests for G-EDF (Goossens et al, 2003; Baker, 2005, 2003; Baruah, 2007; Baker and Baruah, 2009; Bertogna et al, 2005; Bertogna and Cirinei, 2007; Baruah et al, 2009). In general, the performance evaluation in theses works considers the ratio of schedulable task sets. For example, Bertogna and Baruah compared the main G-EDF schedulability tests for HRT scenarios up to 8 processors (Bertogna and Baruah, 2011). Baker compared three G-EDF schedulability tests with P-EDF approaches also in terms of the ratio of schedulable task sets (Baker, 2005).

Global schedulers based on different concepts have also been proposed. Cho et al. proposed a new abstraction for task execution on multiprocessors, named the time and local remaining execution-time plane (T-L plane) (Cho et al, 2006). The entire scheduling over time is the repetition of T-L planes of various sizes. Other global scheduling algorithms are based on the proportional fairness, such as PFair and its variants (Baruah et al, 1996; Anderson and Block, 2000; Anderson et al, 2003; Srinivasan et al, 2003; Levin et al, 2010). Proportional Fairness-based schedulers are optimal, since they correctly schedule any feasible intra-sporadic task system on m processors. On the other hand, proportional fairness algorithms incur more run-time overhead, because of higher scheduling decision and migration rates (Srinivasan et al, 2003).

Other real-time schedulers focus on different processor architectural aspects, such as memory hierarchy. Calandrino and Anderson proposed a cache-aware scheduling algorithm (Anderson et al, 2006; Calandrino and Anderson, 2008). In their approach, there are two scheduling phases: (i) all tasks that

may induce significant memory-to-L2 traffic are combined into groups off-line; and (ii) at run-time, the system uses a scheduling policy that reduces concurrency within groups. The authors introduce the concept of *megatask*, which represents a task group treated as a single schedulable entity. FP_{CA} is another cache-aware scheduling algorithm that divides the shared cache space into partitions (Guan et al, 2009). Tasks are scheduled in a way that at any time, any two running tasks' cache spaces (e.g., a set of partitions) are non-overlapped. A task can execute only if it gets an idle core and enough cache partitions. The authors proposed two schedulability tests, one based on a linear problem (LP) and another one as an over-approximation of the LP test. Tasks are not preemptive and the algorithm is blocking, i.e., it does not schedule lower priority ready jobs to execute in advance of higher priority even though there are enough available resources.

In this paper, we extended the G-EDF versus P-EDF comparison made by previous related works. We compare eight G-EDF schedulability tests with three P-EDF partitioning techniques (first-fit decreasing, best-fit decreasing, and worst-fit decreasing) using synthetic task sets composed of different periods and utilizations in a scenario with 100 processors.

2.2 Run-time Overhead and Implementation Tradeoffs

During the last years, a group of real-time extensions were proposed to the Linux kernel. Linux/Resource Kernel (Linux/RK) is an implementation of the Portable RK in the Linux kernel (Oikawa and Rajkumar, 1999). Portable RK is a framework that abstracts the resource management (e.g., CPU, network, and disk bandwidth reservation) from a specific OS. Moreover, Portable RK provides an API (Application Program Interface) to users, thus a developer when porting the framework for a different OS, only needs to deal with the differences in the OS APIs (Oikawa and Rajkumar, 1999). However, for implementing the Portable RK in the Linux kernel, the authors inserted a number of *callback hooks* to send relevant scheduling events to the Portable RK framework. The authors measured several run-time overhead sources in the framework, but did not provide a comparison with other Linux-based real-time implementations. Current research topics in Linux/RK include multicore processors, integration with real-time java, and resource reservation strategies.

SCHED_DEADLINE is an implementation of a real-time scheduling class using the Linux scheduling class mechanism (Faggioli et al, 2009). The scheduling class supports EDF and Constant Bandwidth Server (CBS) (Abeni and Buttazzo, 2004) algorithms to provide temporal isolation among tasks. Furthermore, SCHED_DEADLINE uses the control groups (*cgroups*) API to natively support multicore platforms and hierarchical scheduling.

ExSched is a scheduler framework that enables the implementation of different (real-time) schedulers as external plugins without modifying the OS (Åsberg et al, 2012). ExSched hides all OS dependencies, which allows different OSes to reuse the same plugin implementation. An API is responsible

for all communication between user applications and the target OS, occurring in more execution time overhead. The authors implemented some multicore real-time scheduling algorithms, such as partitioned and global fixed-priority schedulers, and compared the run-time performance of ExSched multicore implementations to the SCHED_FIFO Linux scheduler in terms of schedulability ratio of generated task sets. For some cases, fixed-priority schedulers in ExSched had worse performance than the Linux SCHED_FIFO scheduler.

Advanced Interactive Real-time Scheduler (AIRS) is another real-time extension for Linux designed on top of SCHED_DEADLINE scheduling class (Kato, 2012). AIRS increases the system performance when multiple interactive real-time applications, like multimedia applications, run on a multicore processor. AIRS proposes two new concepts: Flexible CBS (F-CBS) that improves the CPU bandwidth reservation and the Window-constrained Migration and Reservation (EDF-WMR) scheduler that improves the absolute CPU bandwidth available for multicore real-time applications (Kato, 2012). In a comparison carried out by the authors, ARIS delivered higher quality to simultaneous multiple videos than the existing real-time Linux extensions (e.g., AIRS, SCHED_DEADLINE, Linux/RK, and $LITMUS^{RT}$) (Kato, 2012).

Adaptive Quality of Service Architecture (AQuoSA) is a patch for Linux that allows the interception of scheduling events and consequently, the implementation of external schedulers (Palopoli et al, 2009). The main objective of AQuoSA is to provide Quality of Service (QoS) to time-sensitive applications. A resource reservation module implements different algorithms, such as CBS, IRIS and GRUB. The module also implements an EDF scheduler to manage the internal queues. Generally, all the above related works are either patches or extensions applied to the Linux kernel. In consequence, all of these works suffer from the inherent non real-time aspect of Linux, higher run-time overhead (mainly the OS-independent frameworks), and are tightly coupled to the Linux infrastructure. Differently, our work shows how an RTOS designed from scratch allows a better code reuse and a smaller run-time overhead than Linux-based approaches.

Several multicore real-time schedulers were evaluated considering run-time overhead using the **LI**nux **T**estbed for **MU**ltiprocessor **S**cheduling in **R**eal-**T**ime systems ($LITMUS^{RT}$ – see Section 3.2 for an overview). Calandrino et al. measured run-time overheads of G-EDF, P-EDF, and two variants of the PFair algorithm (Calandrino et al, 2006). Brandenburg et al. investigated the scalability in terms of number of processors in partitioned and global schedulers (Brandenburg et al, 2008). Brandenburg and Anderson discuss how the implementation of the ready queue, quantum-driven versus event-driven scheduling, and interrupt handling strategies affect a global real-time scheduler, considering the different run-time overhead sources in each implementation (Brandenburg and Anderson, 2009). The results indicate that implementation issues can impact schedulability as much as scheduling-theoretic tradeoffs. Moreover, in their case study, the system achieved the best performance by using a fine-grained heap, event-driven scheduling, and dedicated interrupt handling. An empirical comparison of G-EDF, P-EDF, and Clustered-EDF (C-

EDF) on a 24-core Intel platform, assuming run-time overhead (e.g., release, context switch, and scheduling) and cache-related preemption and migration (CPMD) delay, has concluded that P-EDF outperforms the other evaluated algorithms in HRT scenarios (Bastoni et al, 2010b). Moreover, the same study suggests the use of "less global" approaches (P-EDF and C-EDF-L2, which cluster at the cache level 2) in contrast of "more global" approaches (G-EDF and C-EDF-L3) for HRT applications. Bastoni et al. investigated implementation tradeoffs in semi-partitioned schedulers (Bastoni et al, 2011). Mollison and Anderson proposed a userspace scheduler implemented on top of Linux that supports C-EDF (Mollison and Anderson, 2012). The authors measured OS overhead, including releasing, scheduling and switching context, and compared the obtained values with LITMUS^{RT}. They concluded that the overheads of both implementations are roughly comparable.

Following the same research line, Lelli et al. compared the performance of partitioned, clustered, and global Rate Monotonic (RM) and EDF scheduling algorithms on top of Linux, focusing on SRT applications (Lelli et al, 2012). The authors concluded that the migration overhead is not more costly than a context switch, using an AMD Opteron with 48 cores NUMA platform. In addition, the clustered variants were more efficient than the global approaches mainly due to reduced run-time overhead, as also noted by Bastoni et al (2010b). Other studies created micro-benchmarks to quantify context switch overhead on specific processors and/or situations, e.g., hardware interrupts, program data size, and cache performance (Mogul and Borg, 1991; David et al, 2007; Li et al, 2007; Tsafrir, 2007).

In general, the related works that measured run-time overhead in real-time multicore schedulers used Linux as OS. We extended these works by comparing LITMUS^{RT} (that uses Linux) with EPOS RTOS in terms of run-time overhead and schedulability ratio. Moreover, to the best of our knowledge, EPOS is the first open-source research RTOS that supports global schedulers in the context of multicore real-time systems. We believe that the OS can be extensively and easily used to conduct research in the area, due to higher predictability and smaller overhead, obtaining more precise results for HRT scenarios.

2.3 Cache-related Preemption and Migration Delay

Cache-related Preemption and Migration Delay (CPMD) is the delay caused by the loss of cache affinity after a preemption/migration. Basically, there are two ways of estimating CPMD: through offline static analyses and through online empirical experiments. Static analysis techniques estimate CPMD by analyzing the program code/data of the preempted and preempting tasks. Then, the analysis determines which data or instructions are reused after a preemption (Negi et al, 2003; Stärner and Asplund, 2004; Staschulat and Ernst, 2005; Yan and Zhang, 2008; Hardy and Puaut, 2009; Altmeyer et al, 2012).

Schedule-Sensitive and *Synthetic* are two methods to measure CPMD through experimentation (Bastoni et al, 2010a). In the first method, the sys-

tem records the delays online by executing tests and collecting the measured data through the use of a time-stamp counter. The drawback of this method is the impossibility of controlling when a preemption or migration happens, what causes many useless data to be collected (Bastoni et al, 2010b). The second method tries to overcome this problem by explicitly controlling preemptions and migration of a task, and thus measuring the delays. The evaluation shows that the CPMD in a system under load is only predictable for working set sizes that do not trash the L2 cache (Bastoni et al, 2010a).

On our work, we measure the CPMD by using hardware performance counters online through an RTOS in an 8-core modern processor. We use the obtained CPMD values to compare P-EDF and G-EDF through the weighted schedulability metric (Bastoni et al, 2010a). For high CPMD, P-EDF and G-EDF tend to have the same performance.

3 Task Model and Background

In this work we consider the periodic task model, in which a task set τ is composed of n tasks, $\{T_1, T_2, \ldots, T_n\}$, that are scheduled on m identical processors or cores $\{P_1, P_2, \ldots, P_m\}$. Each task T_i, where $i \leq n$ and $i \geq 1$, has a period p_i and a worst-case execution time (WCET) e_i. A task T_i releases a job at every p_i time units. r_i^j denotes the release time of the jth job of T_i, named T_i^j. The relative deadline of the task T_i is equal to its period: $d_i = p_i$. The relation e_i/p_i defines the utilization of a task T_i, called u_i. The sum of all tasks' utilizations defines the total system utilization ($\sum_{i=1}^{n} u_i$).

We focus on HRT applications. A HRT system is considered schedulable if and only if no job misses its deadline. System designers use schedulability tests to prove that a task set τ is schedulable or not considering a specific real-time scheduling algorithm. Our empirical evaluation in this work considers two real-time scheduling algorithms: G-EDF and P-EDF.

The G-EDF scheduling algorithm organizes the jobs in ascending order by deadlines and schedules the jobs according to this order. There is only one global ready queue from which the scheduler chooses ready jobs to be scheduled based on available processors. Consequently, a job can be preempted due to the release of a higher priority job and can be later resumed on another processor, resulting in a job migration. Migrations and preemptions cause delay with relation to the cache affinity, since a preempted job may evict the preempting task's cached data. Several schedulability tests for the G-EDF scheduling provide a HRT bound for a task set (Goossens et al, 2003; Baker, 2005, 2003; Baruah, 2007; Baker and Baruah, 2009; Bertogna et al, 2005; Bertogna and Cirinei, 2007; Baruah et al, 2009).

The P-EDF scheduling algorithm statically partitions tasks of a task set τ into available processors using a partitioning heuristic, such as first-fit decreasing, and schedules the tasks on each processor using the EDF scheduling (Liu, 2000), separately. The partitioning problem is equivalent to the bin packing problem when a task set has deadlines equal to periods. The bin packing

problem is a NP-hard problem (Garey and Johnson, 1990), and so is tasks partitioning. This means that partitioning heuristics can produce a solution that is not optimal. P-EDF does not allow migrations among processor, consequently there is no migration overhead. Moreover, there are cases (task sets with heavy tasks) that a task set cannot be partitioned. For example, consider the scheduling of five tasks with the same utilization of 0.51 on four processors. There is no partitioning algorithm able to allocate these five tasks into four processors.

3.1 OS Overhead

In this section we summarize the main sources of run-time overhead and provide an overview of main OS implementation strategies.

There are two basic ways to implement a scheduler in an OS: using event-driven scheduling or quantum-based scheduling (Liu, 2000). In the former, the OS performs every scheduling decision after an event, such as a job release or job completion. In the latter, the OS performs every scheduling decision at a timer interrupt. The hardware timer period (a *tick*) defines the interval of two interrupts. However, the quantum-driven scheduling can have precision problems. For instance, in a system that generates an interrupt every 10 ms, a 15 ms task period interval may have to wait for 20 ms to be released. On the other hand, a periodic timer can be very precise if every interrupt coincides with a job releasing (Fröhlich et al, 2011).

In addition, the data structure responsible for ordering the tasks in the scheduler also plays an important role on the performance. The use of a list or heap as ready queue affects the time to insert and remove tasks and consequently, impacts the real-time scheduling.

The design of timer interrupts in the system is also relevant. In a multicore processor, a single core can handle timer interrupts or timer interrupts can be distributed across all cores (each core handles its own timer interrupts). Moreover, each timer interrupt can be periodic, based on tick counting, or single-shot. The periodic approach generates an interrupt at the hardware timer period rate, while the single-shot approach only generates an interrupt when there is an event to be released. However, the single-shot approach requires a timer reprogramming at each interrupt to set up the timer with the next event interval. The single-shot approach can also fall back to tick counting if the requested event period is greater than the maximum hardware timer period. In general, single-shot tends to cause less interference on the system (Fröhlich et al, 2011).

Additionally, the processor architecture is also a source of overhead for real-time applications. For example, the way that the DRAM controller handles concurrent accesses, the implementation of bus arbiters, and different cache replacement algorithms significantly impact the execution time of an application. Although important, processor architecture overhead is difficult to be controlled by the OS and will not be discussed in this work.

In summary, the main sources of overhead in an RTOS are:

– **Context switching:** the process of storing and restoring CPU context of the running task and the next schedulable task. Its time is largely dependent on the CPU (e.g., registers, stack, etc) and OS.
– **Tick counting:** it is the delay spent to handle a timer interrupt and count a tick, in case of periodic timer, or to reprogram the timer, in case of single-shot timer.
– **Task releasing:** it is the delay required to release a task after a timer interrupt.
– **Scheduling:** it is the delay taken to choose a new task to be ran. Scheduling overhead includes operations such as inserting and removing from the run queue and changing the state of a task, from ready to waiting for example.
– **Inter-process interrupt latency:** an inter-process interrupt (IPI) is necessary when a job is released on one core and must be executed on another core because it has the lowest priority running task. An IPI is issued to call a reschedule operation on another core. In our platform (Intel i7 processor, see Table 1) for example, the WCET of an IPI is 0.3 µs (see Section 5).
– **Preemption and migration delay:** the CPMD is the delay caused by a preemption that incurs a loss of cache affinity after resuming the preempted job (Bastoni et al, 2010a). Predicting CPMD on processors with complex shared caches hierarchy is a difficult problem. Related works commonly use empirical approximations and/or static analyses (Wilhelm et al, 2008; Yan and Zhang, 2008; Hardy and Puaut, 2009; Bastoni et al, 2010a). This delay is highly dependent on the working set size (WSS) of a task.

Each job incurs a scheduling and context switch overhead twice, a releasing overhead once, and a preemption/migration delay at most once (Liu, 2000; Brandenburg and Anderson, 2009).

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

$LITMUS^{RT}$ is a real-time extension to Linux. LITMUS$^{\text{RT}}$ is implemented as a plugin and allows different multiprocessor scheduling algorithms to be implemented as plugin components using the available Linux infrastructure (Calandrino et al, 2006). The current LITMUS$^{\text{RT}}$ version (2012.1) is based on Linux 3.0 and supports G-EDF, P-EDF, C-EDF, and PFair scheduling algorithms.

LITMUS$^{\text{RT}}$ adds a new and highest priority scheduling class to the traditional Linux scheduler. As a consequence, LITMUS$^{\text{RT}}$ always executes the highest priority jobs in advance of the Linux original scheduler (Brandenburg et al, 2008). LITMUS$^{\text{RT}}$ changes the Linux scheduler to invoke the plugin initialization functions, scheduling and tick handlers at run-time. At each timer interrupt interval (1 ms), the processor invokes the tick handler. Linux invokes the scheduler handler at every scheduling decision to select the next task to

be executed (Brandenburg and Anderson, 2009). In addition, LITMUS^{RT} provides a userspace library to create real-time tasks. Initially, tasks are created as non real-time and specific functions are called to initialize real-time settings and per-task data structures and put a task in real-time mode.

3.3 Hardware Performance Counters

Hardware Performance Counters (HPCs) are special registers available in most of the modern microprocessors through a hardware Performance Monitoring Unit (PMU). HPCs offer support for counting or sampling several micro-architectural events in real time, such as cache misses and retired instructions (Sprunt, 2002). However, HPCs are typically difficult to use due to limited hardware resources (for example Intel Sandy Bridge supports event counting with four event counters per core-thread) and complex interfaces (e.g., low-level and specific to micro-architecture implementation) (Azimi et al, 2005).

Nevertheless, it is possible to use multiplexing techniques to overcome the limitation in the number of HPCs (May, 2001; Sprunt, 2002) or specific libraries that make the use of HPCs easier (Dongarra et al, 2003; Gracioli and Fröhlich, 2011), yet adding a low overhead to the application. HPCs can be used together with OS techniques, such as scheduling and memory management, to monitor and identify performance bottlenecks to perform dynamic optimizations (Azimi et al, 2009). In multicore systems, for instance, it is possible to count the numbers of snoop requests, last-level cache misses, and evicted cache lines.

4 Real-time Support on EPOS

The Embedded Parallel Operating System (EPOS) is a multi-platform, object-oriented, component-based, embedded system framework (Fröhlich, 2001; EPOS, 2012). Platform-independent system components implement traditional OS services, such as threads and semaphores. Hardware Mediators implement platform-specific support (Polpeta and Fröhlich, 2004). Hardware mediators are functionally equivalent to device drivers in Unix, but do not build a traditional Hardware Abstraction Layer (HAL). Instead, hardware mediators sustain the interface contract between software and hardware by means of static metaprogramming techniques and inlining code that “dilute” mediator code into components at compile time (no calls, no layers, no messages; mostly embedded assembly). EPOS has been used in several academic and industry research and development projects in the last years, such as software-defined radio (Mück and Fröhlich, 2011), wireless sensor networks (Wanner and Fröhlich, 2008), and energy-efficient applications (Fröhlich, 2011). In the next subsections we summarize the real-time support on EPOS, highlighting the advances of this work.

4.1 Scheduling

Figure 1 shows the three main components that form the real-time scheduling support on EPOS. The `Thread` class represents an aperiodic task and defines its execution flow, with its own context and stack. The class implements traditional thread functionalities, such as suspend, resume, sleep, and wake up operations. The `Periodic_Thread`[3] class extends the `Thread` class to provide support for periodic tasks by aggregating mechanisms related to the periodic task re-execution. The `wait_next` method performs a `p` operation on a semaphore that forces the thread to sleep during its defined period. The `Alarm` class, on the other hand, performs a `v` operation to release and wake up the thread. The periodic thread constructor creates the alarm. Section 4.2 shows the activities performed on each periodic thread operation (sleep and wake up).

Fig. 1 UML class diagram of the real-time scheduling components on EPOS.

The `Scheduler` class and `Criterion` sub classes define the structure that realizes task scheduling. Usually, object-oriented OS scheduler implementations use a hierarchy of specialized classes of an abstract scheduler class. Sub classes specialize the abstract class to provide different scheduling poli-

[3] An EPOS periodic thread is conceptually implemented as a real-time periodic task.

cies (Marcondes et al, 2009). EPOS reduces the complexity of maintaing such hierarchy and promotes code reuse by detaching the scheduling policy (here represented by the `Criterion` sub classes) from its mechanism (e.g., data structure implementations as lists and heaps). The data structure in the scheduler class uses the defined scheduling criterion to order the tasks accordingly. At compile time the thread's `Trait`[4] class defines the scheduling criterion (for example `typedef Scheduling_Criteria::GEDF Criterion` defines the scheduling criterion as G-EDF). The `Scheduler` consults the information provided by the criterion class to define the appropriate use of lists and operations.

Each criterion class basically defines the priority of a task, which is later used by the scheduler to choose a task (`operator ()`), and other criterion features, as preemption and timing, for instance. In this work, we extended the EDF criterion to support G-EDF. We created a flag, `GLOBAL_SCHEDULER`, that informs the scheduler whether the criterion is global or not. If the criterion is global, the scheduler uses a specialized version of the `Scheduling_Queue` class, with only one global queue in the system. It is important to highlight that the `Scheduler`, `Scheduling_Queue`, and all other list implementations are metaprogrammed, which means that the compiler solves all dependencies and consequently, there is no overhead at execution time. The current version of the EPOS G-EDF scheduler uses an ordered list.

With this separation of concerns among scheduler, criterion, and thread, it is straightforward to add new scheduling policies into the system. Moreover, in cases where a scheduling policy requires specific scheduling treatment, a new scheduler may be created by extending the existing schedulers through metaprogramming specialization techniques (Czarnecki and Eisenecker, 2000).

4.2 Periodic Thread Operations

As stated before, the `Periodic_Thread` implementation uses an `Alarm` and a `Semaphore` to guarantee the re-execution of a thread. In this case, the `Semaphore` performs the sleep and wake up periodic thread operations, instead of preventing concurrent accesses. After executing a code, the periodic thread calls the `wait_next` method to wait until the next period. The `wait_next` method is a call to the `p` method of the periodic thread's `Semaphore`. Figure 2 depicts the UML sequence diagram of the sequence of calls starting from the `wait_next` method. The `begin_atomic` method prevents concurrent accesses by accessing a `spinlock` and disabling interrupts. The thread `dispatch` method releases the `spinlock` and enables the interrupts later. The `Semaphore` passes a `Queue` as an argument to the thread `Sleep` method. This method suspends the running thread, inserts this thread into the semaphore queue, chooses another thread to be ran by calling the `Scheduler chosen` method, and switch the context between them by calling the `dispatch` method. The

[4] A trait class is a template class that associates information of a component at compile time.

scheduler `suspend` method removes and updates the head from the scheduling ordered list and the `chosen` method gets the new head.

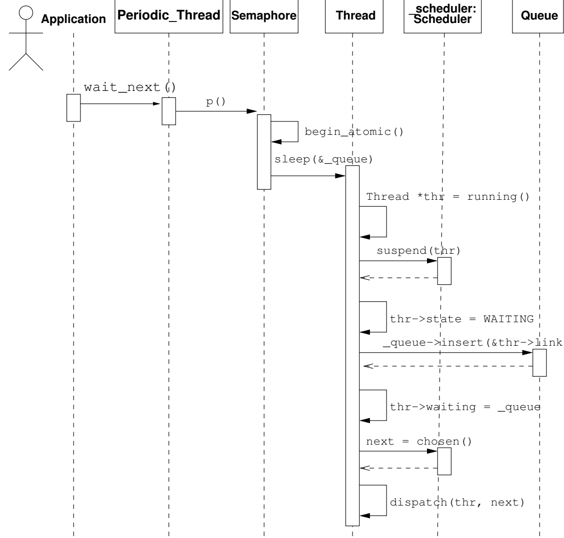

Fig. 2 UML sequence diagram of thread sleep method.

The `Alarm` class is responsible for counting the time until a periodic thread can be released. To release a thread, the `Alarm` calls the `v` method of the `Semaphore` class. Figure 3 depicts the UML sequence diagram of the wake up operation. The **alt** label means an if/else condition and the **opt** label means an if clause. Again, the `begin_atomic` method protects shared data by locking a `spinlock` and disabling interrupts. The thread `dispatch` method, called by the `reschedule` method, releases the `spinlock` and enables the interrupts. The `Semaphore` calls the `wakeup` method, which removes the `Thread` blocked on the semaphore's `Queue` and calls the scheduler to reinsert the thread into the scheduler list (`resume` method) according to the defined scheduling criterion (`Criterion` sub classes in Figure 1). In the end, the `wakeup` method calls the thread `reschedule` to choose the highest priority task to be ran.

Figure 4 shows the thread `reschedule` UML sequence diagram. We inserted a condition to test if an IPI is necessary in case of using a global scheduler. The system fires an IPI when the CPU that is executing the reschedule is not the CPU that is executing the lowest priority thread on the system. Then, the Interrupt Controller (IC) hardware mediator sends an IPI through the `ipi_send` method to switch the context on that CPU. A method in the `Scheduler` informs the lowest priority CPU. Moreover, we do not send an IPI when the lowest priority thread on another CPU is an idle thread, since an idle thread yields the CPU when there is a thread to be scheduled and the time to send an IPI (about 0.3 µs on our platform, shown in Section 5.2) is slower than

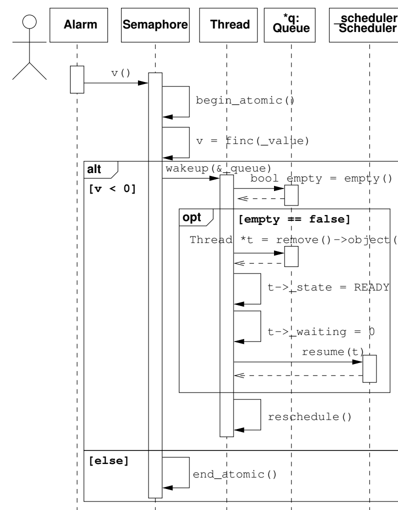

Fig. 3 UML sequence diagram of thread wake up method.

the idle thread yield cycle. It is important to highlight that the idle thread only calls the locking mechanism (e.g., spinlock and disable interrupts) when there is a thread to be scheduled. Consequently, there is no influence in the system. If an IPI is not needed, the `reschedule` method just calls the `choose` method from the `Scheduler` to choose the next thread and switch the context between the running thread and the chosen thread. Note that the scheduling criterion statically sets the `GLOBAL_SCHEDULER` flag. As a consequence, the compiler processes the if condition in the `reschedule` method at compile time and thus does not incur in run-time overhead.

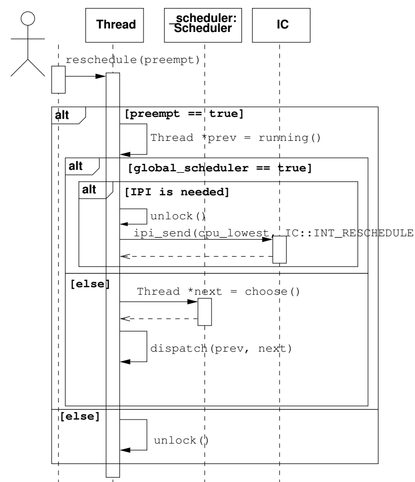

Fig. 4 UML sequence diagram of thread reschedule method.

4.3 Context Switching

The thread `dispatch` method is responsible for switching context between the previous and chosen threads. The method verifies if a context switch needs to be performed by checking if the chosen thread is not the same as the running thread. If both threads are different, the method changes the state of the running thread to "ready", the chosen thread state to "running", and calls the CPU hardware mediator `switch_context` method to perform the context switch. Figure 5 presents the UML class diagram of the CPU hardware mediator. The mediator handles the most dependencies of process management. The inner class `CPU::Context` defines the execution context for each process architecture. The method `CPU::switch_context` is responsible for the switching context, receiving the old and new contexts. The CPU mediators also implement several functionalities as enabling and disabling interrupts and test and set lock operations. Each process architecture defines a set of registers and specific addresses, but the same interface remains. Thus, it is possible to keep the same operations for platform-independent components, such as threads, synchronizers, and alarms.

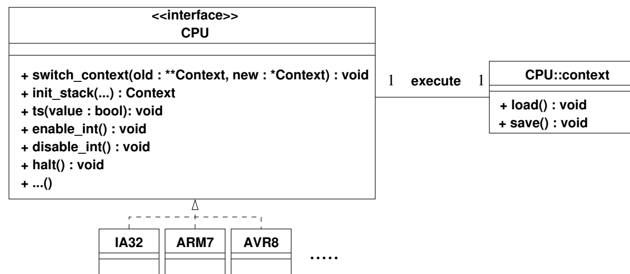

Fig. 5 CPU hardware mediator UML class digram.

4.4 Alarm and Timer Interrupt Handler

The `Alarm` component handles timed-based events. The component uses a `Timer` hardware mediator class that abstracts the hardware timer. In a periodic event model, EPOS sets the hardware timer with a constant (configurable) frequency. When a new alarm event is registered, its interval is converted to timer *ticks*, with T = I / F, where T is the number of ticks, I is the desired interval, and F is the timer frequency (Fröhlich et al, 2011). The alarm inserts all created events in an ordered and relative request queue. Thus, operations on the queue only affect its head, because all queue elements are relative to the first element. The `Alarm` component registers an interrupt handler that increments the tick counter, thus promoting every alarm in the event queue by a tick, at every hardware timer interrupt. The `Alarm` interrupt handler

releases the head queue's event handler if there are no more ticks to count to that event. The event handler, when using a `Periodic_Thread`, releases the thread by calling the `p` semaphore method.

We changed the described alarm handler in two ways. First, we distributed the handler across all CPUs. Thereby, when the application creates a `Periodic_Thread`, the alarm constructor assigns the event handler of that `Periodic_Thread` to a specific CPU. Each different handler manages its own event list separately. Second, we changed the handler to release all events that reach 0 ticks in the same alarm interrupt handler. This way, we guarantee that all events are released without any tick delay (i.e., wait one or more ticks to be released) and that the OS always executes the m highest priority threads.

4.5 Summary of Real-time Extensions and Overhead Sources

Figure 6 summarizes the sources of run-time overhead in EPOS. In Figure 6(b), message 1 and the `Alarm handler` method form the tick counting and thread release overheads. The `Thread dispatch` and `CPU::switch_context` methods constitute the context switching overhead. Finally, message 2 and the `Thread sleep` method in Figure 6(a) and messages 2 and 3 and `Thread wakeup/reschedule` methods in Figure 6(b) form the scheduling overhead.

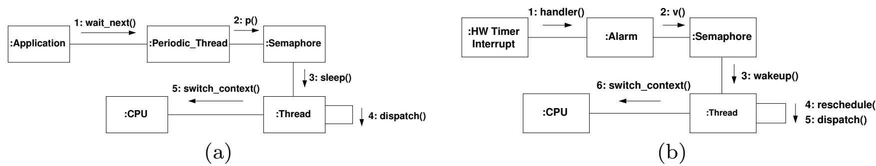

Fig. 6 UML communication diagram summarizing the sources of run-time overhead in EPOS. (a) Operations initiated by the periodic thread sleep operation. (b) Operations initiated by the hardware timer.

In summary, we carried out the following extensions into the original EPOS real-time support:

– **G-EDF scheduling criterion:** we have added the G-EDF scheduling criterion by extending the EDF criterion and inserting a new flag representing a global scheduler (`GLOBAL_SCHEDULER`). This flag informs the scheduler that the scheduling criterion is global and thus, allows the scheduler to choose an appropriate scheduling list implementation. We also added a new scheduler specialization class that chooses the implementation of the global scheduling list.
– **Distributed alarm handler:** we have performed two modifications in the original EPOS alarm handler. First, we distributed the handler in all available CPUs. Each handler has its own private relative and ordered list. Second, we changed the handler to release all periodic threads that reached

0 ticks at the same interrupt. In this way, we are sure that the OS always executes the m highest priority periodic threads.
- **Support for global schedulers:** we have extended the IC hardware mediator to support IPI messages and added the IPI call into the thread `reschedule` method. The IPI message allows the implementation of virtually any global scheduler.

5 Evaluation

This section shows the comparison between G-EDF and P-EDF. First, we discuss the generation of task sets and the evaluation methodology. Then, we show the comparison between EPOS and LITMUS^{RT} (version 2012.1, see Table 1) overheads and the empirical evaluation of the schedulability ratio.

5.1 Experiments Description

To measure the OS overhead and the schedulability ratio of G-EDF and P-EDF, we randomly generated task sets with different distributions similar to Baker (2003, 2005) and Brandenburg and Anderson (2009). For generating tasks periods (all values are in ms), we used a uniform distribution between [3, 33] (short), [10, 100] (moderate), and [50, 250] (long). For generating tasks utilizations, we used a uniform distribution between [0.001, 0.1] (light), [0.1, 0.4] (medium), and [0.5, 0.9] (heavy), and a bimodal distribution (combining two uniform distribution) between [0.001, 0.1] and [0.5 ,0.9], with probabilities 8/9 and 1/9 (light), 6/9 and 3/9 (medium), and 4/9 and 5/9 (heavy), respectively. There are in total 18 combinations of periods and utilizations. Based on the generated task's period and utilization, we defined the task's WCET (before adding the OS overhead).

For measuring the overhead associated to OS activities, we fixed the number of tasks to 5, 15, 25, 50, 75, 100, and 125, and used the light uniform utilization with short periods to generate the tasks. Each task sums a local variable in a loop of 50 repetitions. Then, we applied three sufficient schedulability tests (Goossens et al, 2003; Baker, 2005; Bertogna et al, 2005). A task set was considered feasible if it had passed in at least one test. We then used the generated task sets to measure the overhead of EPOS and LITMUS^{RT}. We executed each task set for 100 times on the Intel i7-2600 processor (see Table 1) for LITMUS^{RT} and 10 times for EPOS[5], and extracted the sampled WCET for each overhead from these executions. We used these numbers of tasks to measure the run-time overhead, because they represent the range of tasks in our generated synthetic task sets.

[5] This is due a technical reason. EPOS does not have file system neither I/O drivers for our platform. The only way to collect data is printing on the screen after each execution. Although the comparison seems unfair, the standard deviations in EPOS are low compared to LITMUS^{RT}.

Table 1 Intel i7-2600 processor features and LITMUS^{RT} version.

Clock speed	3.4 Ghz
Cores	4
Hyper-threading (SMT)	2 per core (8 logical cores)
L1 cache	4 x 64 KB 8-way set associative (32 KB separate data and instructions caches)
L2 cache (non-inclusive)	4 x 256 KB 8-way set associative (unified)
L3 cache (inclusive)	8 MB 16-way set associative (unified)
LITMUS$^{\mathrm{RT}}$ version	2012.1 kernel 3.0

To determine the schedulability of a task set that considers OS overhead, we first inflated the WCET of each task by adding the measured overheads, as described in Section 3.1, and then applied eight sufficient schedulability tests[6] (Goossens et al, 2003; Baker, 2005, 2003; Baruah, 2007; Baker and Baruah, 2009; Bertogna et al, 2005; Bertogna and Cirinei, 2007; Baruah et al, 2009). The additional overhead for a task is dependent on the number of tasks in the task set. We verified the number of tasks and associated it with the measured overhead interval. For example, the additional overhead for a task set with 20 tasks is the measured overhead for 15 tasks, since the task number is between 15 and 25.

As in (Brandenburg and Anderson, 2009), we did not use the Baruah's test (Baruah, 2007) for light uniform utilization due to high processing time caused by its pseudo-polynomial behavior. We considered a task set schedulable if it passed at least one test. We created tasks until reaching a fixed utilization cap (from 2 to 8, in steps of 0.1). For each utilization, we defined a slack related to the utilization cap. For example, a slack of 0.05 specifies that a utilization cap U of a task set τ is always between the interval U - 0.05 and U. Thus, we make sure that the utilization values are always between two consecutive caps. The slacks used were 0.07, 0.07, and 0.1 for light, medium, and heavy utilization distributions, respectively. We generated 1000 task sets for each utilization cap. For the P-EDF algorithm, we first partitioned the task set using three partitioning algorithms (first-fit decreasing, best-fit decreasing, and worst-fit decreasing (Johnson, 1973)) and then applied the EDF test (Liu and Layland, 1973) for each partition (eight in total). A task set is schedulable if all the eight partitions pass in the test and at least one partitioning algorithm correctly partitions the task set.

[6] We used the open-source implementation of the eight G-EDF schedulability tests available at `http://www.cs.unc.edu/~bbb/diss/`. We changed the code to allow the tests to be executed in parallel in a cluster and extended it to support also P-EDF partitioning heuristics and EDF uniprocessor test. The new code is also available online at `http://epos.lisha.ufsc.br`.

5.2 Run-time Overhead

We measured the context switch, release, tick, IPI, and scheduling overheads for EPOS and LITMUS^{RT} on the Intel i7-2600 processor. To record the overheads, in EPOS, we use the processor Time-Stamp Counter (TSC) and in LITMUS^{RT}, the tracing support accomplished by Feather-Trace (Brandenburg and Anderson, 2007).

Context switch overhead. For EPOS, we configured a test case composed of two threads a and b. Thread a sets the TSC before switching the context (Thread `dispatch` method, as described in Section 4.3) and thread b reads the TSC and calculates the difference, resulting in the total context switch time for thread a. Thus, we can isolate the exact time that a context switch takes. We measured in total 5,000,000 context switches and extracted the worst-case[7] and average times from these executions. For LITMUS^{RT}, we measured the worst-case and average times by using Feather-Trace and running the task sets with fixed number of tasks, as presented previously. The total execution time is about 16 seconds, since the greatest period of all periodic tasks is 33 ms and each periodic thread repeats for 500 times.

Figure 7 shows the average and worst-case context switch overhead. The x-axis shows the number of threads and the y-axis the measured execution time in µs. The greater the execution time, the higher the run-time overhead. The error bars represent the observed standard deviation. The average context switch overhead for EPOS is 0.03 µs and the worst-case is 0.3 µs. For LITMUS^{RT}, the average context switch overhead is about 1.2 µs both for P-EDF and G-EDF. For the worst-case context switch overhead, there is a high variation of the observed execution times, from 9.4 µs to 29.56 µs in G-EDF and from 2 µs to 26.95 µs for P-EDF.

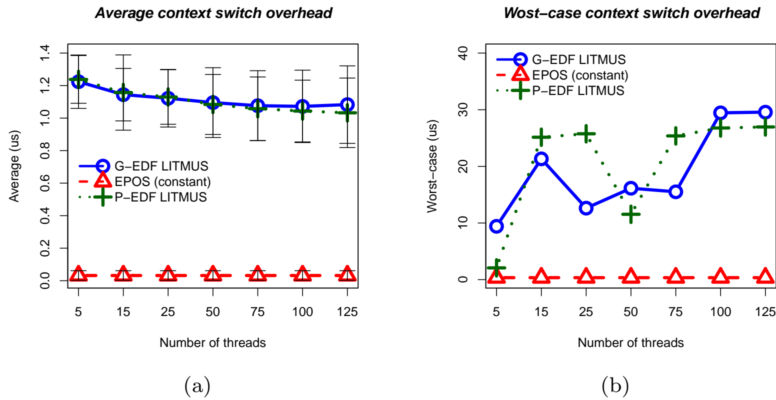

Fig. 7 Average (a) and worst-case (b) context switch overhead.

[7] From now on, whenever we refer to worst-case is the observed worst-case from the experiments.

EPOS is up to 76.3 times faster than the LITMUS^{RT} context switch function. For algorithms with a high rate of preemptions and context switches, such as fairness-based schedulers (Baruah et al, 1996; Anderson and Block, 2000; Anderson et al, 2003; Srinivasan et al, 2003; Levin et al, 2010), the use of an RTOS with low context switch overhead certainly improves the task set schedulability ratio, as will be shown in Section 5.4. It is important to highlight that we removed few (from two to eight) outliers from LITMUS^{RT} measurements. For example, we obtained a worst-case context switch time of up to 2.000 µs. This interference may be due to a set of factors, such as warm-up effects in the instrumentation code and non-deterministic aspects of Linux (Brandenburg et al, 2008). This result corroborates the theory that Linux suffers interference from other system's parts, harming the predictability needed in HRT systems.

IPI latency. For measuring the IPI latency in EPOS, we used a similar approach to the previous context switch overhead measurement. We designed a test case in which a thread running on a core sends an IPI to another core. This thread sets the TSC before sending an IPI and reads it after the IPI is delivered. We are sure that the IPI was delivered, because the `IC` hardware mediator method waits until the requested interrupt to be delivered by checking a register flag. We measured the worst-case and average times for 5,000,000 IPIs. The observed worst-case IPI time for EPOS was 0.41 µs and the average case was 0.009 µs with a standard deviation of 0.003 µs. The observed worst-case is comparable to the worst-case context switch overhead. Additionally, we measured the IPI latency in LITMUS^{RT} for 125 tasks. In LITMUS^{RT}, the worst-case IPI latency was 69.52 µs and the average case was 15.67 µs with a standard deviation of 18.15 µs. This difference between EPOS and LITMUS^{RT} is mainly caused by the `IC` hardware mediator and EPOS' design. The `ipi_send` mediator method is "diluted" into the application code at compile time. There are no software layers between the application and OS, only embedded assembly code. Moreover, we used the EPOS library mode, in which the system is linked with the application, avoiding the overhead of system calls. EPOS also supports a kernel mode, which creates a system call layer between the application and OS. The system designer can choose the best configuration that fits the application requirements. Our focus is on performance, that is the reason we chose the library mode. Low IPI latency is important because it delivers the interrupt message faster, which affects the preemption delay in the core that is receiving the interrupt message.

Scheduling overhead. We measured the G-EDF and P-EDF scheduling overheads in EPOS and LITMUS^{RT}. The `CPU_Affinity` scheduler, described in Section 4.1, implements the P-EDF algorithm in EPOS by assigning priorities according to the EDF scheduling. The `sleep`, `wakeup`, and `reschedule` thread methods, including the list operations and thread state changes, as demonstrated in the UML sequence diagrams of Section 4.2, represent the scheduling overhead in EPOS. In EPOS, we measured the scheduling overhead using the processor's TSC and in LITMUS^{RT}, using the Feather-Trace.

Figure 8 shows the average and worst-case scheduling overhead for both P-EDF and G-EDF in EPOS and LITMUS^{RT}. The x-axis shows the number of threads and the y-axis the measured execution time in µs. The error bars represent the observed standard deviation. For instance, the average scheduling overhead of EPOS for 100 tasks is about 0.5 µs with a standard deviation of about 0.2 µs. In Figure 8(b), the observed worst-case scheduling overhead for EPOS increases after 75 threads, mainly in G-EDF scheduler. This is due to the scheduling list: insertion and removal operations take more time, because there are more threads in the list (time complexity in the worst-case of $O(n)$). For P-EDF, on the other hand, as the threads are evenly distributed across the cores and each core has its own ready scheduling list, we did not observe a considerable variation in the overhead. For LITMUS^{RT}, as the number of threads increases, the unpredictability of Linux increases as well. We again removed few outliers in the LITMUS^{RT} measurements, as explained before. For G-EDF, EPOS was up to 7 times faster than LITMUS^{RT}, while for P-EDF it was up to 21.05 times faster.

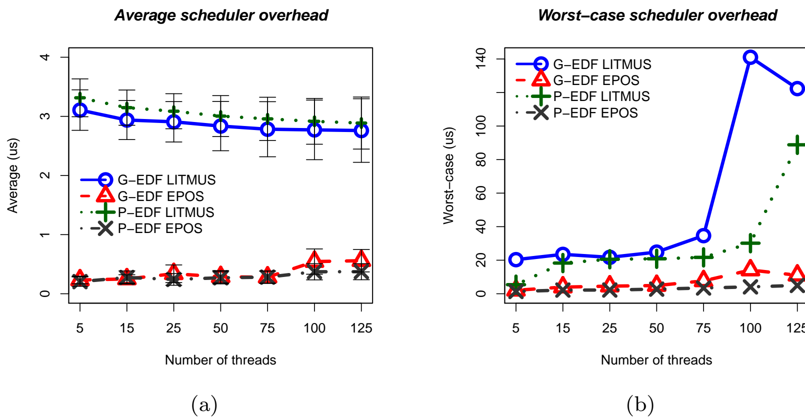

Fig. 8 Average (a) and worst-case (b) scheduling overhead.

Tick counting and thread release overheads. Tick counting and thread release are two different sources of run-time overhead. However, we combined the two overheads because EPOS performs both tick counting and task releasing in the same alarm `handler` method (see Section 4.4). Moreover, we did not measure separate overheads for the P-EDF and G-EDF schedulers, because tick counting and thread releasing operations are the same for both schedulers.

Figure 9 shows the average and worst-case tick counting and release overheads. The x-axis shows the number of threads and the y-axis the measured execution time in µs. The error bars represent the observed standard deviation. For example, the average overhead for 125 threads in EPOS is about 0.3 µs with a standard deviation of about 0.5 µs. Considering the worst-case

tick counting and release overhead, in Figure 9(b), the observed worst-case times increase according to the number of threads in the system: the alarm `handler` releases more threads in the same tick, affecting the overhead. On the other hand, the overhead in LITMUS^{RT} increases considerably after 75 threads. One of the contributing factor for this is also the greater number of threads in the kernel data structures. Furthermore, the use of a slower data structure in EPOS (list) against the LITMUS^{RT} heap implementation is not critical, because EPOS performs less operations before and after inserting or removing elements from the data structure. As the number of threads increases, the standard deviation of EPOS increases as well. This is because the greater time difference between an interrupt that only counts a tick and an interrupt that releases several threads.

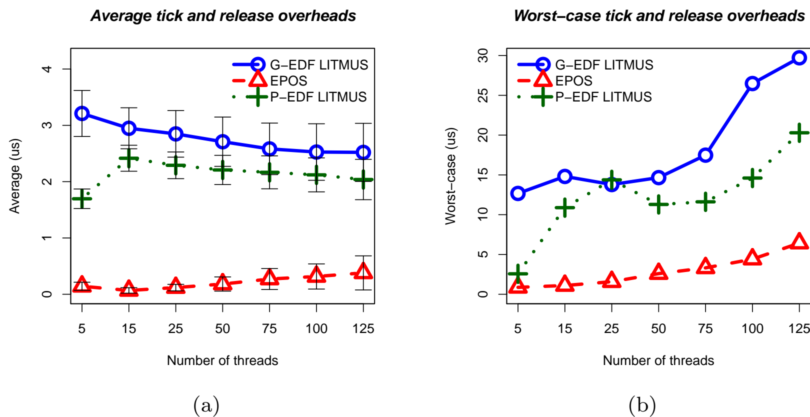

Fig. 9 Average (a) and worst-case (b) tick counting and thread release overheads.

5.3 Preemption/Migration Overhead

We measured the CPMD using the EPOS hardware performance counter API (Gracioli and Fröhlich, 2011). We configured HPCs to count hardware events that together form three metrics, represented in Equations 1, 2, and 3. The metrics calculate the impact of L1, L2, and L3-cache misses in terms of cycles spent serving them (Intel Corporation, 2011, 2012). The Last-Level Cache Miss Impact (LLC MI) is the number of all memory accesses that missed the LLC multiplied by the number of cycles spent to serve one LLC miss:

$$\text{LLC MI} = 200 * \text{mem load uops retired that miss LLC} \tag{1}$$

The Last-Level Cache Hit Impact (LLC HI) is the sum of all memory accesses that hit the LLC with no bus snoop needed, all memory accesses that

hit the LLC and required a cross-core snoop hit, and all memory accesses that hit the LLC and had a hit modified response from another core multiplied by the processor cycles spent to serve each hardware event:

$$\begin{aligned} \text{LLC HI} = 31 * \text{mem load uops retired that hit LLC} + \\ 43 * \text{mem load uops with LLC hit and snoop hit} + \\ 60 * \text{mem load uops with LLC hit and hitm response} \end{aligned} \quad (2)$$

The L2-cache Hit Impact (L2 HI) multiplies the number of all memory accesses that hit the L2-cache by the processor cycles spent in one hit:

$$\text{L2 HI} = 12 * \text{mem load uops that hit L2} \quad (3)$$

The sum of the three equations gives us the total impact of all accesses that missed in the L1-cache. However, the platform presents a hardware limitation to perform this calculation. Intel i7-2600 processor offers only four programmable and three-fixed hardware counters per each core-thread, while the metrics need five counters. We eliminated from Equation 2 the event that counts the LLC hit and had a cross-core snoop hit modified response, because there is no data sharing in our test application (see Figure 10) and this event only captures cache coherency activities.

Figure 10 shows part of the application code used to measure the CPMD. We vary the Working Set Size (WSS) from 4 KB to 10 MB, which provides a reasonable relation to the size of the three cache memory levels (see Table 1). The `Perf_Mon` component configures the hardware counters and then reads them into a buffer. At every iteration, the application calls the `wbinvd` instruction to write back all modified cache lines to main memory and to invalidate the internal caches. Thus, we emulate a situation where a thread entirely loses its cache affinity after a preemption/migration. We used the `CPU_Affinity` scheduler and at every thread period we changed the affinity to force a CPU migration.

We executed the test application for ten times and extracted the worst-case values and the average cases for each WSS from these executions. We also considered a theoretical worst-case bound: the cache line size (64) divides the WSS, and the processor cycles to treat an LLC (200) multiplies the division resulting value. Figure 11 shows the calculated worst-case bound, the sampled worst-case, and the average CPMD for our test application. On the x-axis, we vary the WSS and on the y-axis, we present the CPMD in µs and in logarithm scale. Comparing the sampled worst-case with the calculated worst-case bound, the hardware pre-fetcher considerably improves the CPMD: it brings data to the LLC that is later accessed, which does not cause an LLC miss. For WSS of 10 MB, it is possible to observe a smaller difference between the sampled and calculated worst-case bound values, because the application thrashes the cache. Additionally, the CPMD difference between the average and sampled worst-case is not high, and the average cases have a low standard

```
1  int job(int factor, int id)
2  {
3      Perf_Mon perf;
4      int array[WSS];
5      int sum = 0;
6      for(int i = 0; i < ITERATIONS; i++) {
7          Periodic_Thread::wait_next();
8          asm("wbinvd");
9          perf.cpmd();
10         for(int k = 0; k < factor; k++) {
11             for(int j = 0; j < WSS; j++)
12               sum += array[j];
13             if(k == 0)
14               perf.get_cpmd(threads[id]->_buffer);
15         }
16     }
17 }
```

Fig. 10 CPMD application code.

deviation (error bars that represent standard deviation are almost imperceptible). This shows that hardware performance counters can provide a correct view of the application behavior.

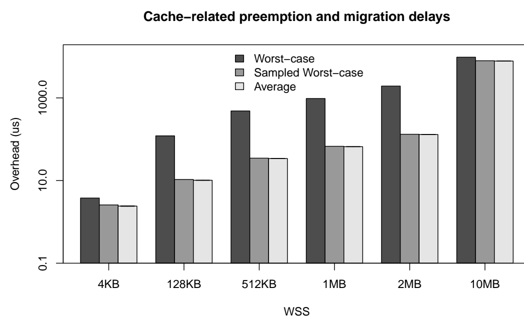

Fig. 11 Cache-related preemption and migration delay varying the WSS in microseconds. Note that the y-axis uses a logarithm scale.

5.4 Schedulability Tests Evaluation

We present below the empirical comparison between G-EDF and P-EDF. We first show the comparison considering the run-time overhead measured in eight processors and then we extend the results to 100 processors, considering only the ideal P-EDF and G-EDF tests (i.e., without overhead).

Schedulability experiments considering the run-time overhead. Figure 12 shows the task set schedulability ratio for short periods and the

six combinations of uniform and bimodal utilization distributions. In the x-axis, we vary the utilization cap and in the y-axis, we present the ratio of schedulable task sets. A ratio of 0.6, for instance, means that 60% of the total generated task sets are schedulable.

In Figure 12(a), that shows the results for the light uniform utilization, P-EDF is able to partition and schedule all task sets. The partitioning heuristics performed well, because tasks utilizations are very low. For the bimodal light utilization, P-EDF did not have the same behavior due to few heavy tasks: for instance, in Figure 12(b), the P-EDF schedulability ratio starts to decrease around the utilization cap of 7.0.

For all analyzed distributions, except in the uniform heavy utilization, G-EDF is worse than P-EDF. When all tasks in a task set have utilizations between 0.5 and 0.9 (in the uniform heavy utilization), the partitioning heuristics can only partition a task set with a total number of tasks equal to the processor number (eight). Figure 12(f) exemplifies this situation, where around the utilization cap of 5.1, task sets have more than eight tasks. Consequently, the schedulability ratio starts to drop. For G-EDF, instead, the HRT bounds in the sufficient schedulability tests limit the schedulability ratio.

In the light uniform utilization, when the number of tasks in a task set is greater than in the medium and heavy utilization distributions, the run-time overhead impact is more significant. For example, in the uniform light utilization (Figure 12(a)), P-EDF considering the overhead in LITMUS$^{\mathrm{RT}}$ is worse than G-EDF and G-EDF considering the overhead in EPOS.

Figure 13 shows the task set schedulability ratio for moderate periods and the six combinations of uniform and bimodal utilization distributions. We observe a reduction in the overhead impact for all distributions compared to shorter periods: as the periods become larger, the proportion among periods and overheads becomes smaller. Figure 13(a) exemplifies this situation, where the lines for G-EDF with EPOS and LITMUS$^{\mathrm{RT}}$overheads are closer to the theoretical G-EDF than the previous graph with short periods (Figure 12(a)). Moreover, for the same light uniform distribution, P-EDF with EPOS and LITMUS$^{\mathrm{RT}}$ overheads also improved the schedulability ratio compared to the short period. For instance, the schedulability ratio of P-EDF with EPOS and LITMUS$^{\mathrm{RT}}$ overheads in moderate periods start to decrease from 1 in the utilization caps of 7.9 and 6.6, respectively. For short periods, in contrast, the schedulability ratio of P-EDF inflated by the overhead in EPOS starts to drop in the utilization cap of 7.8 and the schedulability ratio of P-EDF considering the overhead in LITMUS$^{\mathrm{RT}}$ in the utilization cap of 4.6.

Figure 14 shows the task set schedulability ratio for long periods and the six combinations of uniform and bimodal utilization distributions. Following the same trend, long period lengths reduce the overhead impact on the schedulability ratio. Moreover, for the light uniform utilization (Figure 14(a)), P-EDF inflated by the overhead in LITMUS$^{\mathrm{RT}}$ was better than the ideal G-EDF for the first time. Additionally, P-EDF inflated by the overhead in EPOS presents an improvement of 79% in the schedulability ratio for the utilization cap of

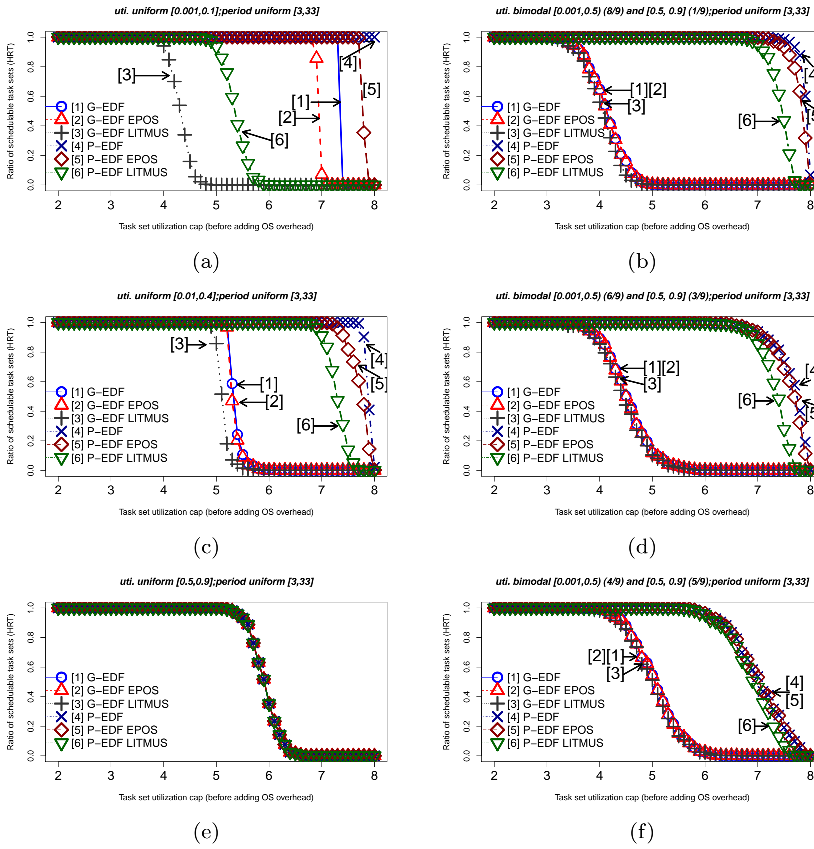

Fig. 12 Comparison between G-EDF and P-EDF with short periods (a) Uniform light (c) Uniform medium (e) Uniform heavy (b) Bimodal light (d) Bimodal medium (f) Bimodal heavy.

7.9 compared to the same utilization cap in the light uniform utilization with short periods.

Table 2 summarizes the schedulability ratio results of P-EDF and G-EDF, considering the three utilization variants (light, medium, and heavy), and the uniform and bimodal distributions. G-EDF and P-EDF have the same performance for heavy uniform utilizations. Considering the different periods (short, moderate, and long), the biggest difference in terms of schedulability ratio between G-EDF and P-EDF is in the bimodal light utilization. The presence of few heavy tasks profoundly affects the bound in the G-EDF schedulability tests. Moreover, in the light uniform utilization and short periods scenario, the impact of the run-time overhead on the schedulability ratio is more significant, because the proportion between period and overhead is smaller. Furthermore, G-EDF inflated by the overhead in EPOS is better than P-EDF inflated by the overhead in LITMUS^{RT}, differently of the theoretical tests, in which P-EDF is

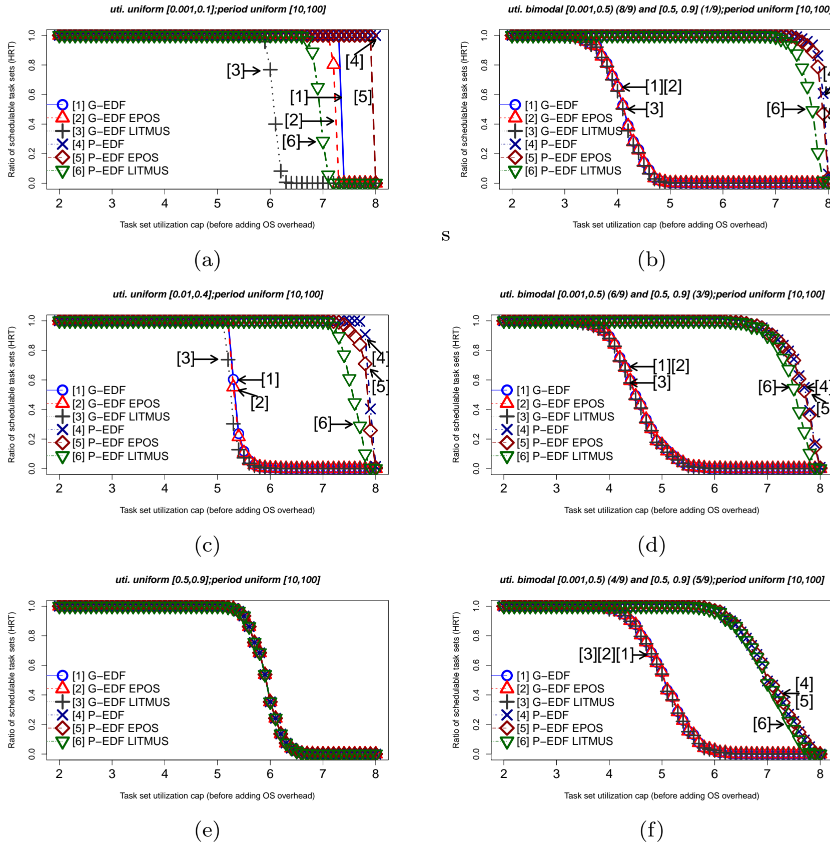

Fig. 13 Comparison between G-EDF and P-EDF with moderate periods (a) Uniform light (c) Uniform medium (e) Uniform heavy (b) Bimodal light (d) Bimodal medium (f) Bimodal heavy.

always better than G-EDF (except for heavy uniform utilizations). In the bimodal heavy utilization, the run-time overhead only changes the schedulability ratio for P-EDF and short periods.

Weighted schedulability. We used the weighted schedulability to account for CPMD (Bastoni et al, 2010a). As the schedulability ratio depends on two variables (i.e., utilization cap and CPMD), the weighted schedulability reduces the results to a two-dimensional plot without the use of the utilization cap.

Let D_c be a maximum CPMD incurred by any job and U be a utilization cap. $S(U, D_c)$ denotes the schedulability ratio for a given U and D_c, which is in the interval $[0, 1]$. Let Q be a set of utilization caps ($Q = \{2.0, 2.1, \ldots, m\}$). Then, Equation 4 defines the weighted schedulability for a D_c, $W(D_c)$ (Bastoni et al, 2010a).

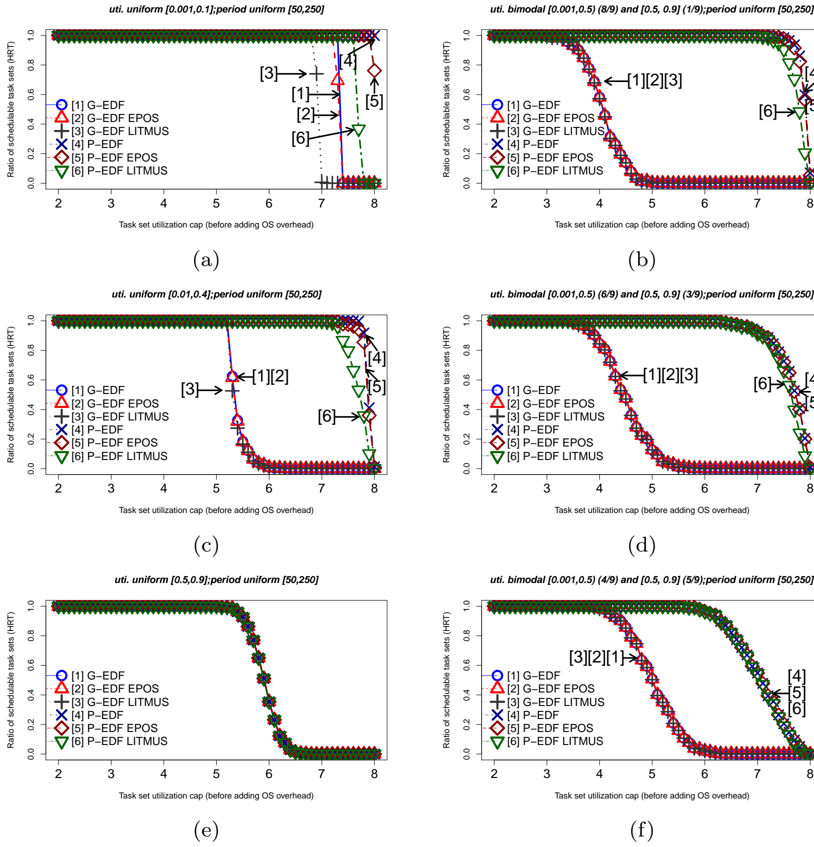

Fig. 14 Comparison between G-EDF and P-EDF with long periods (a) Uniform light (c) Uniform medium (e) Uniform heavy (b) Bimodal light (d) Bimodal medium (f) Bimodal heavy.

$$W(D_c) = \frac{\sum_{U \in Q} U \times S(U, D_c)}{\sum_{U \in Q} U} \tag{4}$$

We used the calculated worst-case bound CPMD values shown in Figure 11 as input to D_c. We assume that each task suffers the CPMD once — a single job is potentially preempted multiple times, but each job in the system can only cause one CPMD on one other job. We then inflated the WCET e_i of each task T_i by the CPMD and the run-time overhead measured before and applied the eight G-EDF schedulability tests and the three partitioning heuristics into the same previously generated task sets.

Figure 15 shows the obtained results for task sets with short periods. The x-axis defines the CPMD, while the y-axis presents the weighted schedulability metric. For instance, the weighted schedulability for a CPMD of 128 KB and P-EDF inflated by the overhead in EPOS is 0.82, which means that 82%

Table 2 Summary of the schedulability ratio comparison between G-EDF and P-EDF, considering also the run-time overhead in EPOS and $\text{LITMUS}^{\text{RT}}$.

<table>
<tr><th></th><th></th><th>P-EDF</th><th>G-EDF</th></tr>
<tr><td rowspan="2">Light utilization</td><td>Uniform</td><td>P-EDF and P-EDF considering the overhead in EPOS have the best performance among all analyzed scenarios.</td><td>G-EDF with EPOS overhead is better than P-EDF with $\text{LITMUS}^{\text{RT}}$ overhead.</td></tr>
<tr><td>Bimodal</td><td>All the three P-EDF variants are better than G-EDF.</td><td>P-EDF is better than G-EDF. The biggest difference in the schedulability ratio between the algorithms.</td></tr>
<tr><td rowspan="2">Medium utilization</td><td>Uniform</td><td>The run-time overhead is smaller than in light utilization due to the lower number of tasks in the task sets.</td><td rowspan="2">All G-EDF scenarios are worse than P-EDF. Moreover, G-EDF with the OS run-time overhead is equal or close to the ideal G-EDF.</td></tr>
<tr><td>Bimodal</td><td>P-EDF considering the overhead of EPOS and $\text{LITMUS}^{\text{RT}}$ is close to the ideal P-EDF.</td></tr>
<tr><td rowspan="2">Heavy utilization</td><td>Uniform</td><td colspan="2">G-EDF and P-EDF have the same performance.</td></tr>
<tr><td>Bimodal</td><td>The run-time overhead changes the schedulability ratio only for short periods. P-EDF is better than G-EDF.</td><td>The run-time overhead does not change the schedulability ratio.</td></tr>
</table>

of all generated task sets are schedulable. G-EDF and P-EDF lines serve as a reference for analyzing the results, since they have no overhead. For the uniform heavy utilizations distribution (Figure 15(e)), we observe the same trend as before: G-EDF and P-EDF have the same performance for all CPMD values. For small WSSs (4 KB and 128 KB), P-EDF is superior to G-EDF. As the WSS increases, the difference between P-EDF and G-EDF decreases, which can be clearly seen in Figure 15(c). For WSSs greater than 512 KB, P-EDF and G-EDF tend to be equal mainly due to the high CPMD values. Figures 15(a), (b), and (c) show the cases where the difference between the weighted schedulability of EPOS and $\text{LITMUS}^{\text{RT}}$ is more significant: there are more tasks in a task set, which favors EPOS due to the smaller run-time overhead.

For moderate and long periods the graphs are similar to the previously ones: as the periods become larger, the proportion between CPMD and the period becomes smaller. As a consequence, the weighted schedulability ratio starts to drop only for higher CPMD values. For example, Figure 16 shows the weighted schedulability results for light uniform utilization and long periods. Compared to the light uniform utilization and short periods, the weighted schedulability reaches 0 only for WSS of 10 MB, instead of 1 MB as for short periods.

Extended schedulability evaluation for 100 processors. We extended the empirical comparison between the ideal G-EDF and P-EDF to 100 processor. We used the same period and utilization distributions as described early to generate the task sets. However, we used different utilization slacks:

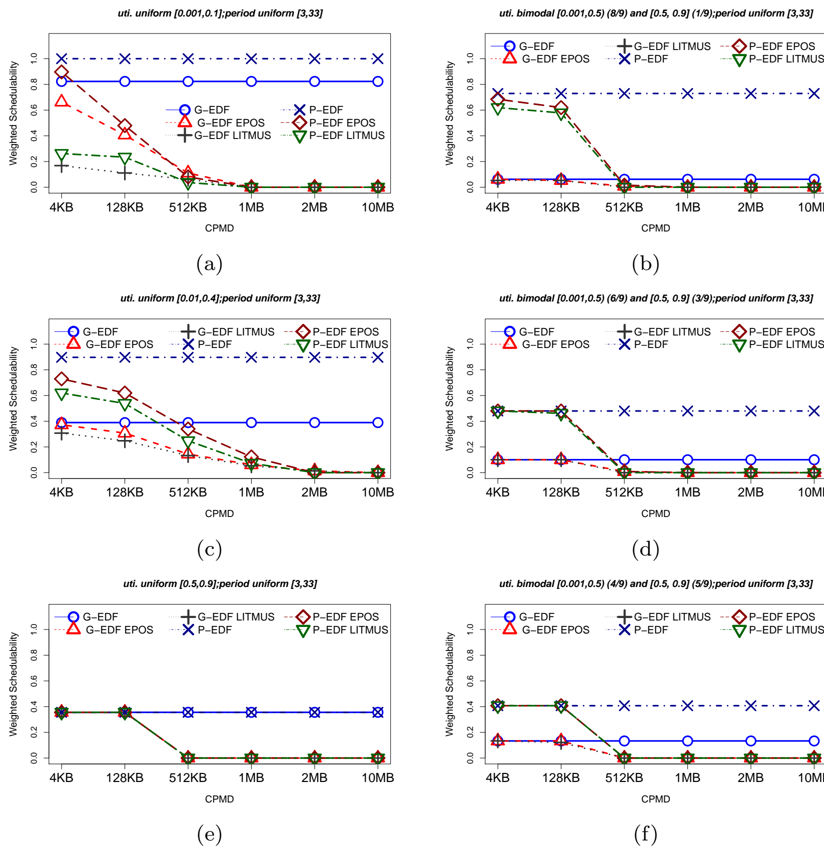

Fig. 15 Weighted schedulability for short periods.

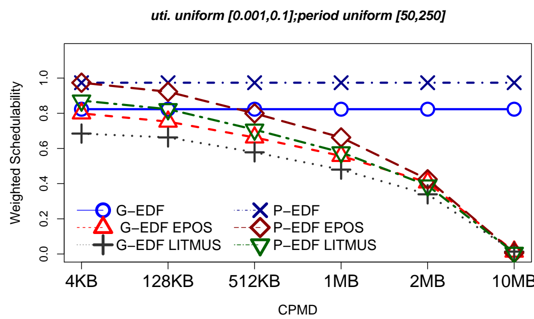

Fig. 16 Weighted schedulability for light uniform distribution and long periods.

0.5, 0.5, and 1 for light, medium, and heavy utilization distributions, respectively. Then, we executed the eight G-EDF sufficient schedulability tests and the three P-EDF partitioning techniques in the SHARCNET cluster (SHAR-

CNET, 2012), varying the utilization cap from 2, 3, 4,..., to 100. The total computation time for all tests and task sets generation was more than 1.1 year.

Figure 17 shows the obtained task set schedulability ratio for uniform utilization distributions. Each row in Figure 17 shows different utilization distributions (uniform, medium, and heavy) and each column different period distributions (short, moderate, and long). The x-axis defines the utilization cap and the y-axis the ratio of schedulable task sets. A ratio of 0 means that none of the task sets is schedulable, while a ratio of 1 means that all task sets are schedulable. P-EDF is always better than G-EDF, except for the heavy distribution. For the heavy distribution (Figures 17(g), (h), and (i)) G-EDF had the same results as P-EDF, for the same reason as explained early: the number of tasks in a task set affects the partitioning heuristics and the HRT bounds in the G-EDF sufficient tests limit the schedulability.

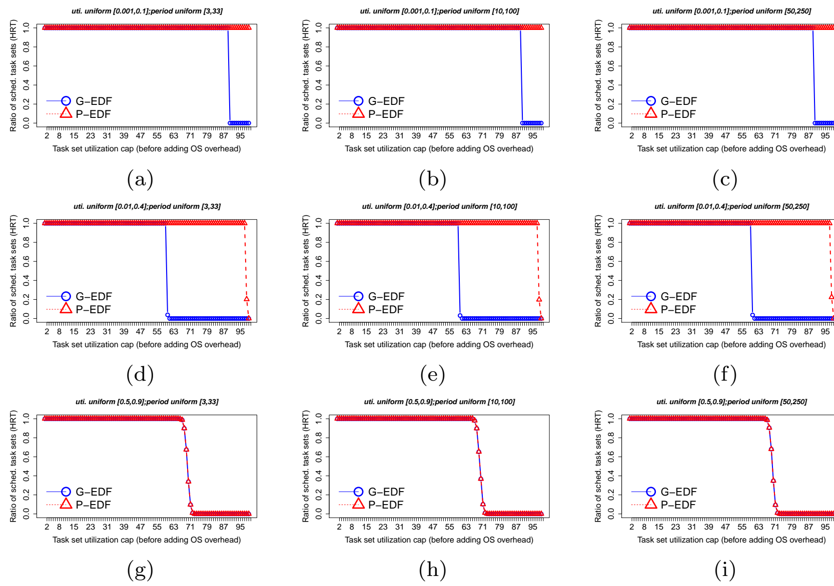

Fig. 17 Comparison between G-EDF and P-EDF using uniform utilizations: first row light uniform, second row medium uniform, and third row heavy uniform.

Figure 18 shows the obtained task set schedulability ratio for the bimodal distributions. The x-axis defines the utilization cap and the y-axis the ratio of schedulable task sets. P-EDF is again always better than G-EDF for HRT tasks. Moreover, for light bimodal utilization (Figures 18(a), (b), and (c)), the schedulability ratio of G-EDF reaches 0 when the utilization cap is 45. This clearly states the need for less pessimistic G-EDF sufficient schedulability tests, since 55% of the available processors are "wasted" due to HRT guarantees.

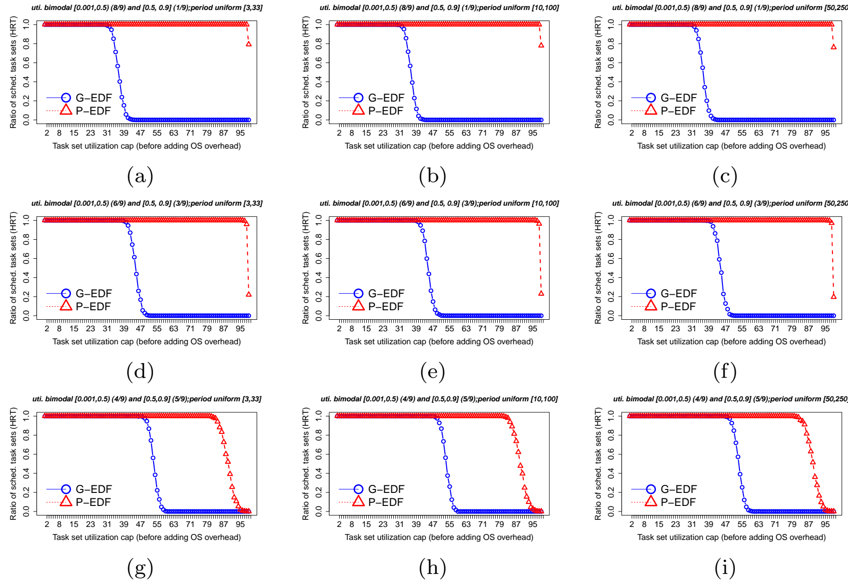

Fig. 18 Comparison between G-EDF and P-EDF using bimodal utilizations: first row light bimodal, second row medium bimodal, and third row heavy bimodal.

6 Discussion

During our evaluation, we observed a set of interesting facts regarding aspects of the OS, processor architecture, and real-time scheduling:

- **OS design versus data structures:** we noticed that the OS design is as important as the internal data structures of the scheduling and task release functions. In comparison to LITMUS^{RT}, EPOS performs less operations before and after a scheduling decision and a task release, resulting in less overhead. Furthermore, we believe that changing the current EPOS lists to a data structure with better performance, such as heaps, can reduce the run-time overhead and consequently, reduce the impact on the schedulability ratio.
- **RTOS versus general-purpose OS:** our results show that an RTOS designed from scratch considerably reduces the run-time overhead in comparison to general-purpose OSes with real-time patches. In scenarios composed of several light utilization tasks, we could note an improvement of about 25% in the task set schedulability ratio of G-EDF considering the overhead in EPOS in contrast to the G-EDF inflated by the overhead in LITMUS^{RT} (Figure 12(a)). Considering P-EDF, each ready scheduling list has less tasks than the global scheduling list of the G-EDF, which reduces the scheduling run-time overhead. Nevertheless, for the same light uniform utilization scenario, P-EDF considering the overhead in EPOS was about 20% better than P-EDF considering the overhead in LITMUS^{RT}. For ex-

ample, while P-EDF inflated by the overhead in LITMUS^{RT} reaches 0% of schedulability ratio at the utilization cap of 6.2, P-EDF considering the overhead in EPOS reaches 0% at the utilization cap of 8. For task sets composed of only heavy tasks, the influence of the run-time overhead on the schedulability ratio is less significant, because there are few tasks in the system. Thus, the scheduling and alarm lists (see Section 4) manage less elements, reducing the overhead.

- **P-EDF is always equal or better than G-EDF for HRT:** for all distributions, except the heavy utilization, P-EDF was superior to G-EDF. For task sets consisting of only heavy utilization tasks, P-EDF and G-EDF had the same schedulability ratio. This is due to the bin packing problem limitation, in which the partitioning heuristics can only partition task sets with a task number equal to the number of processors, and due to the G-EDF schedulability bounds, which usually have a relation between the number of processor and the largest utilization or density (Bertogna and Baruah, 2011). For task sets with few heavy tasks, as in the case of light bimodal utilization distribution, G-EDF presented the biggest difference in terms of task set schedulability ratio in comparison to P-EDF. This reinforces the need for better G-EDF schedulability tests for HRT systems with heavy utilization tasks (Brandenburg and Anderson, 2009).
- **Differences in task period length:** varying period lengths (short, moderate, and long) in our empirical evaluations did not affect the schedulability tests for the theoretical (i.e., without overhead) G-EDF and P-EDF schedulers. On the other hand, it has a significant impact on the run-time overhead. In short period distributions, the proportion between the period length and the overhead is higher than in long period distributions (see Figure 12(a), Figure 13(a), and Figure 14(a)). For example, for short periods and light uniform utilization, the schedulability ratio for G-EDF inflated by the overhead in LITMUS^{RT} starts to drop at the utilization cap of 3.8, while for long periods and the same utilization distribution, the schedulability ratio starts to drop at the utilization cap of 6.9.
- **Hardware performance counters:** we believe that hardware performance counters are useful for helping scheduling and memory management in RTOSs. Although each processor architecture supports different hardware events, different names for the same events, and presents different hardware limitations (e.g., number of registers and features), well-designed OS APIs can abstract these differences for the rest of the system. Usually, hardware event names change but their meaning remain.
  Moreover, we believe that performance monitoring units (PMUs) will support even more events and features in the near future. Examples of features that could be added by hardware designers into future PMUs are (Gracioli and Fröhlich, 2011): (i) data address registers to store addresses that generated an event; (ii) monitoring address space intervals to provide more precise view of specific application address ranges; (iii) processing cycles spent in specific events, such as bus activities and memory coherency protocols; and (iv) OS trap generation according to pre-defined event numbers.

These features can improve scheduling decisions at run-time, providing a correct and precise view of the running applications. Also, such features improve shared memory partitioning algorithms, which are useful to avoid the overlap of shared cache spaces by tasks executing on different processors. Thus, cache partitioning reduces the contention for the shared cache and increases the system predictability (Lin et al, 2008; Suhendra and Mitra, 2008; Srikantaiah et al, 2008; Muralidhara et al, 2010).

- **Cache-related preemption and migration delay:** we measured the CPMD using hardware performance counters and used the weighted schedulability to account for CPMD in the task set schedulability ratio. In our evaluations, P-EDF is better than G-EDF for WSSs of 4 KB and 128 KB. As the WSS increases, P-EDF and G-EDF tend to be equal due to higher CPMD. For uniform and bimodal light and uniform moderate utilizations distributions, which have more tasks than the other generated task sets, the difference between EPOS and LITMUS^{RT} is higher. Moreover, P-EDF and G-EDF had the same performance for task sets composed of only heavy tasks.
  A possible way to decrease the CPMD is the use of cache locking mechanisms (Vera et al, 2003; Suhendra and Mitra, 2008; Aparicio et al, 2011). Cache locking prevents cache lines or ways to be evicted by the cache replacement policy during the program execution. The combination of cache partitioning and cache locking improves the system predictability (Suhendra and Mitra, 2008). However, most of the current processors do not support cache locking. Hardware designers should consider this feature for future processors.

7 Conclusion

In this work, we performed an empirical evaluation in terms of the task set schedulability ratio between G-EDF and P-EDF up to 100 processors. We used eight state-of-the-art G-EDF sufficient schedulability tests and three P-EDF partitioning techniques. Our results show that P-EDF is always better than G-EDF for HRT tasks. An exception is for task sets formed by only heavy tasks, in which P-EDF and G-EDF had the same performance due to partitioning limitation and G-EDF test bounds. These results clearly reinforce the need for better G-EDF schedulability tests, both in terms of HRT bounds and processing time. We also provide real-time multicore support on EPOS by extending the scheduling mechanism to support global schedulers. To the best of our knowledge, EPOS is the first open-source RTOS that supports global real-time schedulers. We then compared EPOS to LITMUS^{RT} in terms of the run-time overhead. The use of a RTOS properly designed to fulfill the application requirements can provide HRT guarantees close to theoretical schedulability tests. In addition, we measured CPMD through the use of hardware performance counters and used the weighted schedulability metric to account for CPMD in the schedulability ratio. In our platform, P-EDF is better than

G-EDF for WSSs of 4 KB and 128 KB. As the WSS increases, P-EDF and G-EDF tend to be equal due to higher CPMD.

Future work. There are several directions for future work. First, we want to improve the real-time support on EPOS by implementing different data structures, such as heaps, and real-time scheduling algorithms, such as PFair, cache-aware, and semi-partitioned schedulers. Second, we want to evaluate how different cache partitioning strategies affect real-time scheduling algorithms. Third, we would like to port EPOS to an embedded multicore processor, such as Arm cortex-A9. Finally, we would like to investigate how shared data between tasks affects HRT schedulers.

Acknowledgements This work was supported by the Coordination for Improvement of Higher Level Personnel (CAPES) and the Foreign Affairs and International Trade Canada / Affaires étrangères et Commerce international Canada (DFAIT) grants, projects RH-TVD 006/2008 and CAPES-DFAIT 004/11.

References

Abeni L, Buttazzo G (2004) Resource reservation in dynamic real-time systems. Real-Time Systems 27(2):123–167

Altmeyer S, Davis RI, Maiza C (2012) Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. Real-Time Systems pp 1–28

Anderson JH, Block A (2000) Early-release fair scheduling. In: In Proceedings of the 12th Euromicro Conference on Real-Time Systems, pp 35–43

Anderson JH, Block A, Srinivasan A (2003) Quick-release fair scheduling. In: Proceedings of the 24th IEEE International Real-Time Systems Symposium, IEEE Computer Society, Washington, DC, USA, RTSS '03, pp 130–

Anderson JH, Bud V, Devi UC (2005) An edf-based scheduling algorithm for multiprocessor soft real-time systems. In: Proceedings of the 17th Euromicro Conference on Real-Time Systems, IEEE Computer Society, Washington, DC, USA, ECRTS '05, pp 199–208

Anderson JH, Calandrino JM, Devi UC (2006) Real-time scheduling on multicore platforms. In: RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE Computer Society, Washington, DC, USA, pp 179–190

Aparicio L, Segarra J, Rodríguez C, Viñals V (2011) Improving the wcet computation in the presence of a lockable instruction cache in multitasking real-time systems. Journal of Systems Architecture 57(7):695 – 706, special Issue on Worst-Case Execution-Time Analysis

Åsberg M, Nolte T, Kato S, Rajkumar R (2012) Exsched: An external cpu scheduler framework for real-time systems. In: 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'12)

Azimi R, Stumm M, Wisniewski R (2005) Online performance analysis by statistical sampling of microprocessor performance counters. In: Proceedings

of the 19th annual international conference on Supercomputing, ACM, New York, NY, USA, ICS '05, pp 101–110

Azimi R, Tam D, Soares L, Stumm M (2009) Enhancing operating system support for multicore processors by using hardware performance monitoring. SIGOPS Operating System Review 43:56–65

Baker TP (2003) Multiprocessor edf and deadline monotonic schedulability analysis. In: Proceedings of the 24th IEEE International Real-Time Systems Symposium, IEEE Computer Society, Washington, DC, USA, RTSS '03, pp 120–

Baker TP (2005) An analysis of edf schedulability on a multiprocessor. IEEE Trans Parallel Distrib Syst 16(8):760–768

Baker TP (2005) A comparison of global and partitioned edf schedulability tests for multiprocessors. Tech. rep., In International Conference on Real-Time and Network Systems

Baker TP, Baruah S (2009) An analysis of global edf schedulability for arbitrary-deadline sporadic task systems. Real-Time Systems 43(1):3–24

Baruah S (2007) Techniques for multiprocessor global schedulability analysis. In: Proceedings of the 28th IEEE International Real-Time Systems Symposium, IEEE Computer Society, Washington, DC, USA, RTSS '07, pp 119–128

Baruah S, Bonifaci V, Spaccamela AM, Stiller S (2009) Implementation of a speedup-optimal global edf schedulability test. In: Proceedings of the 2009 21st Euromicro Conference on Real-Time Systems, IEEE Computer Society, Washington, DC, USA, ECRTS '09, pp 259–268

Baruah SK, Cohen NK, Plaxton CG, Varvel DA (1996) Proportionate progress: A notion of fairness in resource allocation. Algorithmica 15:600–625

Bastoni A, Brandenburg BB, Anderson JH (2010a) Cache-Related Preemption and Migration Delays: Empirical Approximation and Impact on Schedulability. In: Proc. Sixth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications, Brussels, Belgium

Bastoni A, Brandenburg BB, Anderson JH (2010b) An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers. In: Proceedings of the 2010 31st IEEE Real-Time Systems Symposium, IEEE Computer Society, Washington, DC, USA, RTSS '10, pp 14–24

Bastoni A, Brandenburg BB, Anderson JH (2011) Is semi-partitioned scheduling practical? In: Proceedings of the 2011 23rd Euromicro Conference on Real-Time Systems, IEEE Computer Society, Washington, DC, USA, ECRTS '11, pp 125–135

Bertogna M, Baruah S (2011) Tests for global edf schedulability analysis. J Syst Archit 57(5):487–497

Bertogna M, Cirinei M (2007) Response-time analysis for globally scheduled symmetric multiprocessor platforms. In: Proceedings of the 28th IEEE International Real-Time Systems Symposium, IEEE Computer Society, Washington, DC, USA, RTSS '07, pp 149–160

Bertogna M, Cirinei M, Lipari G (2005) Improved schedulability analysis of edf on multiprocessor platforms. In: Proceedings of the 17th Euromicro Conference on Real-Time Systems, IEEE Computer Society, Washington, DC, USA, ECRTS '05, pp 209–218

Bletsas K, Andersson B (2009) Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. In: Proceedings of the 2009 30th IEEE Real-Time Systems Symposium, IEEE Computer Society, Washington, DC, USA, RTSS '09, pp 447–456

Brandenburg BB, Anderson JH (2007) Feather-trace: A light-weight event tracing toolkit. In: In Proceedings of the Third International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'07, pp 61–70

Brandenburg BB, Anderson JH (2009) On the implementation of global real-time schedulers. In: RTSS '09: Proceedings of the 2009 30th IEEE Real-Time Systems Symposium, IEEE Computer Society, Washington, DC, USA, pp 214–224

Brandenburg BB, Calandrino JM, Anderson JH (2008) On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In: Proceedings of the 2008 Real-Time Systems Symposium, IEEE Computer Society, Washington, DC, USA, RTSS '08, pp 157–169

Burns A, Davis R, Wang P, Zhang F (2012) Partitioned edf scheduling for multiprocessors using a c=d task splitting scheme. Real-Time Systems 48(1):3–33

Calandrino JM, Anderson JH (2008) Cache-aware real-time scheduling on multicore platforms: Heuristics and a case study. In: ECRTS '08: Proceedings of the 2008 Euromicro Conference on Real-Time Systems, IEEE Computer Society, Washington, DC, USA, pp 299–308

Calandrino JM, Leontyev H, Block A, Devi UC, Anderson JH (2006) Litmusrt: A testbed for empirically comparing real-time multiprocessor schedulers. In: Proceedings of the 27th IEEE International Real-Time Systems Symposium, IEEE Computer Society, Washington, DC, USA, RTSS '06, pp 111–126

Carpenter J, Funk S, Holman P, Srinivasan A, Anderson J, Baruah S (2004) A categorization of real-time multiprocessor scheduling problems and algorithms. In: Handbook on Scheduling Algorithms, Methods, and Models, Chapman Hall/CRC, Boca

Cho H, Ravindran B, Jensen ED (2006) An optimal real-time scheduling algorithm for multiprocessors. In: RTSS '06, IEEE, pp 101–110

Cullmann C, Ferdinand C, Gebhard G, Grund D, Maiza C, Reineke J, Triquet B, Wegener S, Wilhelm R (2010) Predictability considerations in the design of multi-core embedded systems. Ingénieurs de l'Automobile 807:36–42

Czarnecki K, Eisenecker UW (2000) Generative programming: methods, tools, and applications. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA

David FM, Carlyle JC, Campbell RH (2007) Context switch overheads for linux on arm platforms. In: Proceedings of the 2007 workshop on Experimental computer science, ACM, New York, NY, USA, ExpCS '07

Dongarra J, London K, Moore S, Mucci P, Terpstra D, You H, Zhou M (2003) Experiences and lessons learned with a portable interface to hardware performance counters. In: Proceedings of the 17th International Symposium on Parallel and Distributed Processing, IEEE Computer Society, Washington, DC, USA, IPDPS '03, pp 289.2–

EPOS (2012) Epos website. URL `http://epos.lisha.ufsc.br`

Faggioli D, Checconi F, Trimarchi M, Scordino C (2009) An EDF scheduling class for the Linux kernel. In: Proceedings of the Eleventh Real-Time Linux Workshop, Dresden, Germany

Fröhlich AA (2001) Application-Oriented Operating Systems. No. 17 in GMD Research Series, GMD - Forschungszentrum Informationstechnik, Sankt Augustin

Fröhlich AA (2011) A Comprehensive Approach to Power Management in Embedded Systems. International Journal of Distributed Sensor Networks 2011(1):19

Fröhlich AA, Gracioli G, Santos JF (2011) Periodic timers revisited: The real-time embedded system perspective. Comput Electr Eng 37(3):365–375

Garey MR, Johnson DS (1990) Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA

Goossens J, Funk S, Baruah S (2003) Priority-driven scheduling of periodic task systems on multiprocessors. Real-Time Systems 25(2-3):187–205

Gracioli G, Fröhlich AA (2011) An embedded operating system API for monitoring hardware events in multicore processors. In: Workshop on Hardware-support for parallel program correctness - IEEE Micro 2011, Porto Alegre, Brazil

Guan N, Stigge M, Yi W, Yu G (2009) Cache-aware scheduling and analysis for multicores. In: EMSOFT '09, ACM, New York, NY, USA, pp 245–254

Hardy D, Puaut I (2009) Estimation of Cache Related Migration Delays for Multi-Core Processors with Shared Instruction Caches. In: George L, and-Mikael Sjodin MC (eds) 17th International Conference on Real-Time and Network Systems, Paris, France, pp 45–54

Intel Corporation (2011) Intel® 64 and IA-32 Architectures Software Developer's Manual. 253668-037US

Intel Corporation (2012) Intel® 64 and IA-32 Architectures Optimization Reference Manual. 248966-026

Johnson D (1973) Near-optimal bin packing algorithms. PhD thesis

Kato S (2012) AIRS website. URL `http://www.ertl.jp/~shinpei/airs/`

Kato S, Yamasaki N (2007) Real-time scheduling with task splitting on multiprocessors. In: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, IEEE Computer Society, Washington, DC, USA, RTCSA '07, pp 441–450

Kato S, Yamasaki N (2008) Portioned edf-based scheduling on multiprocessors. In: Proceedings of the 8th ACM international conference on Embedded software, ACM, New York, NY, USA, EMSOFT '08, pp 139–148

Lelli J, Faggioli D, Cucinotta T, Lipari G (2012) An experimental comparison of different real-time schedulers on multicore systems (to appear). Journal

of Systems and Software

Leontyev H, Anderson JH (2007) Generalized tardiness bounds for global multiprocessor scheduling. In: RTSS '07: Proceedings of the 28th IEEE International Real-Time Systems Symposium, IEEE Computer Society, Washington, DC, USA, pp 413–422

Levin G, Funk S, Sadowski C, Pye I, Brandt S (2010) Dp-fair: A simple model for understanding optimal multiprocessor scheduling. In: Proceedings of the 2010 22nd Euromicro Conference on Real-Time Systems, IEEE Computer Society, Washington, DC, USA, ECRTS '10, pp 3–13

Li C, Ding C, Shen K (2007) Quantifying the cost of context switch. In: Proceedings of the 2007 workshop on Experimental computer science, ACM, New York, NY, USA, ExpCS '07

Lin J, Lu Q, Ding X, Zhang Z, Zhang X, Sadayappan P (2008) Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In: International Symposium on High Performance Computer Architecture, IEEE Computer Society, HPCA '08, pp 367–378

Liu CL, Layland JW (1973) Scheduling algorithms for multiprogramming in a hard-real-time environment. J ACM 20(1):46–61

Liu J (2000) Real-Time Systems, 1st edn. Prentice Hall PTR, Upper Saddle River, NJ, USA

Marcondes H, Cancian R, Stemmer M, Fröhlich AA (2009) On the design of flexible real-time schedulers for embedded systems. In: CSE '09: Proceedings of the 2009 International Conference on Computational Science and Engineering, IEEE Computer Society, Washington, DC, USA, pp 382–387

Masrur A, Chakraborty S, Färber G (2010) Constant-time admission control for partitioned edf. In: Proceedings of the 2010 22nd Euromicro Conference on Real-Time Systems, IEEE Computer Society, Washington, DC, USA, ECRTS '10, pp 34–43

May J (2001) Mpx: Software for multiplexing hardware performance counters in multithreaded programs. In: Proceedings of the 15th International Parallel and Distributed Processing Symposium., p 8 pp.

Mogul JC, Borg A (1991) The effect of context switches on cache performance. SIGOPS Operating System Review 25(Special Issue):75–84

Mohan S, Caccamo M, Sha L, Pellizzoni R, Arundale G, Kegley R, de Niz D (2011) Using multicore architectures in cyber-physical systems. In: Workshop on Developing Dependable and Secure Automotive Cyber-Physical Systems from Components, Michigan, USA

Mollison M, Anderson JH (2012) Utilization-controlled task consolidation for power optimization in multi-core real-time systems (to appear). In: 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), vol 1

Mück TR, Fröhlich AA (2011) HyRA: A Software-defined Radio Architecture for Wireless Embedded Systems. In: 10th International Conference on Networks, St. Maarten, The Netherlands Antilles, pp 246–251

Muralidhara SP, Kandemir M, Raghavan P (2010) Intra-application cache partitioning. In: IPDPS 10': Proceedings of the 25th IEEE International

Symposium on Parallel Distributed Processing, pp 1–12
Negi HS, Mitra T, Roychoudhury A (2003) Accurate estimation of cache-related preemption delay. In: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, ACM, New York, NY, USA, CODES+ISSS '03, pp 201–206
Oikawa S, Rajkumar R (1999) Portable rk: a portable resource kernel for guaranteed and enforced timing behavior. In: Proceedings of the Fifth IEEE Real-Time Technology and Applications Symposium, 1999., pp 111 –120
Palopoli L, Cucinotta T, Marzario L, Lipari G (2009) Aquosaadaptive quality of service architecture. Softw Pract Exper 39(1):1–31
Polpeta FV, Fröhlich AA (2004) Hardware mediators: A portability artifact for component-based systems. In: EUC, pp 271–280
SHARCNET (2012) Sharcnet cluster website. URL https://www.sharcnet.ca
Sprunt B (2002) Pentium 4 performance-monitoring features. IEEE Micro 22(4):72–82
Srikantaiah S, Kandemir M, Irwin MJ (2008) Adaptive set pinning: managing shared caches in chip multiprocessors. In: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems, ACM, New York, NY, USA, ASPLOS XIII, pp 135–144
Srinivasan A, Holman P, Anderson JH, Baruah S (2003) The case for fair multiprocessor scheduling. In: IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing, IEEE Computer Society, Washington, DC, USA, p 114.1
Stärner J, Asplund L (2004) Measuring the cache interference cost in preemptive real-time systems. In: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, ACM, New York, NY, USA, LCTES '04, pp 146–154
Staschulat J, Ernst R (2005) Scalable precision cache analysis for preemptive scheduling. In: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, ACM, New York, NY, USA, LCTES '05, pp 157–165
Suhendra V, Mitra T (2008) Exploring locking & partitioning for predictable shared caches on multi-cores. In: DAC '08: Proceedings of the 45th annual Design Automation Conference, ACM, New York, NY, USA, pp 300–303
Tsafrir D (2007) The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In: Proceedings of the 2007 workshop on Experimental computer science, ACM, New York, NY, USA, ExpCS '07
Vera X, Lisper B, Xue J (2003) Data cache locking for higher program predictability. SIGMETRICS Perform Eval Rev 31(1):272–282
Wanner LF, Fröhlich AA (2008) Operating System Support for Wireless Sensor Networks. Journal of Computer Science 4(4):272–281
Wilhelm R, Engblom J, Ermedahl A, Holsti N, Thesing S, Whalley D, Bernat G, Ferdinand C, Heckmann R, Mitra T, Mueller F, Puaut I, Puschner P, Staschulat J, Stenström P (2008) The worst-case execution-time problem overview of methods and survey of tools. ACM Trans Embed Comput Syst

7(3):36:1–36:53

Yan J, Zhang W (2008) Wcet analysis for multi-core processors with shared l2 instruction caches. In: Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE Computer Society, Washington, DC, USA, RTAS '08, pp 80–89