Evaluation Framework for Energy-Aware Multiprocessor Scheduling in Real-Time Systems

PEDRO MEJIA-ALVAREZ*, DAVID MONCADA-MADERO†, HAKAN AYDIN‡, and ARNOLDO DIAZ-RAMIREZ§

Multiprocessor and *multicore* architectures are fast becoming the platform of choice for deploying workloads, as they have higher computing capabilities and energy efficiency than traditional architectures. In addition to time constraints, a number of real-time applications are required to operate in systems working with limited power supplies, which also imposes tight energy constraints on their execution. Therefore, it is desirable for the system to minimize its energy consumption while still achieving a satisfactory performance. Several energy-aware scheduling techniques addressing this issue have been proposed over the past few years. Unfortunately, few aspects of implementation are seldom considered in theoretical work, and only a tiny fraction of these techniques have been implemented in an actual hardware platform and evaluated by analytical methods. The work presented in this paper thus attempts to provide a prototyping and evaluation framework in which energy-aware multiprocessor scheduling algorithms can translate into full-fledged practical realizations, where their power consumption profiles can be properly measured.

Keywords: Real-Time Systems, Energy Management, Scheduling,Multicore Architectures

1 INTRODUCTION

Energy management has become a major design and operational constraint for electronic devices working with limited power supplies. This is especially true for embedded real-time systems, whose performance requirements are often very high and which usually rely on limited energy sources such as batteries. Systems that are better at managing available energy have a longer lifetime and are more reliable. They also help to reduce the carbon footprint and lower the power dissipation that results in heat transfer. Even for systems connected to the power grid, reducing energy consumption provides significant savings in business costs and alleviates environmental issues.

Over the past two decades, many approaches for managing energy consumption (e.g., through processor slowdown or shutdown) in a real-time setting have been proposed for the uniprocessor case. As multiprocessor platforms have become more commonplace, advancements have been made in scheduling theory for supporting real-time applications such that they could benefit from improved multiprocessor performance. However, the use of multiple processing units further complicates the management of resources, including energy, and energy-aware real-time multiprocessor scheduling has begun to attract attention within the real-time research community. Early multiprocessor architectures had processors placed on separate chips, allowing them to independently switch their operating frequency. As such, the first works focusing on energy management for real-time systems running on multiprocessor platforms considered hardware models in which the processors featured per-CPU voltage and frequency scaling capabilities [14, 35, 48]. Subsequent

*Institution: Departamento de Computacion, CINVESTAV-IPN,Mexico City.
†Institution: Microsoft, Redmond, Wa..
‡Institution: Department of Computer Science, George Mason University, FaixFax Virginia..
§Institution: Instituto Tecnologico de Mexicali, MEXICO..

Authors' address: Pedro Mejia-Alvarez, pmalvarez@cs.cinvestav.mx; David Moncada-Madero; Hakan Aydin, aydin@cs.gmu.edu; Arnoldo Diaz-Ramirez, adiaz@itmexicali.edu.mx.

2018. XXXX-XXXX/2018/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

generations of multiprocessor systems began packing multiple processing cores onto a single chip, forcing these to share a common voltage level and run at the same frequency, and research efforts were devoted to tackling the problem of supporting real-time workloads on platforms constrained to global frequency scaling [37, 46].

In most cases, the proposed energy-efficient real-time scheduling algorithms (for both the uniprocessor and multiprocessor cases) are studied using simulations, where the primary objective is to assess their performance in terms of schedulability and energy consumption. Simulation based studies are attractive for a number of reasons: insight into the working of a fairly complex system can be gained relatively quickly and cheaply, and it is easy to identify the most influential factors in the simulation outcome. In the case of energy-aware real-time multiprocessor scheduling algorithms, simulations provide a means for evaluating their energy consumption profile without the need to procure specialized hardware or measurement instrumentation. Additionally, simulations allow focusing on the essential aspects of the algorithms by abstracting away certain details that would complicate the approach, and running a multitude of workloads in a fraction of the time required to run them on real hardware.

Despite extensive study into energy-efficient techniques in the real-time literature [4, 5], there seems to be little interest in evaluating them in an actual Real-Time Operating System (RTOS) running on a real experimental testbed. Studies based *solely* on simulations have limitations. The lack of a standardized simulation environment within the real-time community has led researchers to develop their own simulators, embedding their own set of simplifying assumptions. This makes it difficult to validate the presented results or to determine the relative performance of the proposed techniques. In some works, techniques have been derived that presume hardware models with processors capable of switching to any frequency within a range. Other works assume power dissipation models, only considering the effect of voltage/frequency on the platform's overall energy consumption, and sometimes disregarding the role of static dissipation entirely. Lastly, little advice is provided regarding problems that arise when the techniques are implemented on real hardware. Without proper experimental evaluation of the proposed techniques, real performance risks remain largely unknown.
Specifically, the main contributions of this article are:

- The development of a loosely coupled, generic, reusable component for conferring real-time scheduling plugins with processor voltage and frequency scaling capabilities and managing the synchronization of processors with respect to the global voltage and frequency level [4, 5].
- The implementation and empirical comparison of two representative energy-aware real-time multiprocessor scheduling algorithms designed for platforms where processors are constrained to a single clock domain. Implementation was carried out in LITMUS^{RT} [9, 12], an actively developed and supported real-time extension for the Linux kernel.
- A power dissipation measurement methodology for assessing the practical merits of the implemented algorithms in terms of energy consumption.

The remaining of this article is organized as follows: section 2 provides an overview of the power management subsystems available in Linux and briefly describes LITMUS^{RT}, as well as the extension developed for enabling predictable processor frequency adjustments; section 3 describes LITMUS^{RT}and its extension with a CPUFreq module to perform frequency adjustments at runtime; section 4 describes the key properties of the energy-aware real-time multiprocessor scheduling algorithms evaluated within the framework of the case study presented in this article; section 5 presents the methodology followed for conducting the experimental evaluation of the considered algorithms and discusses the results. Finally, section 6 concludes the article with a few remarks regarding the goals reached.

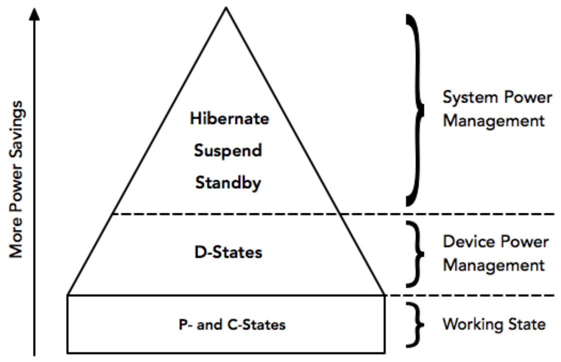


Fig. 1. Overview of the Linux power management infrastructure.

2 ENERGY MANAGEMENT IN OPERATING SYSTEMS

This section provides a comprehensive overview of the power management infrastructure available within the Linux kernel, which allows Linux to fully exploit the power saving capabilities of the various platforms upon which it runs. Also, the architecture of the real-time extension for the Linux kernel behind the evaluations presented in this work, LITMUS^{RT}is described, as well as the design and implementation of the component allowing LITMUS^{RT} scheduling plugins to mesh with part of the Linux power management infrastructure.

2.1 Energy Management in Linux

Within the Linux kernel, power management can be divided into two broad categories: *system* power management and *device* power management [34]. System power management is in charge of transitioning the entire system (or parts of it - the processor, for example) into performance states or low-power modes, using the CPUFreq and CPUIdle subsystems, among other measures. Device power management is concerned with placing unused system devices into low-power states. Figure 1 shows an overview of the performance and low-power states provided by the Linux power management infrastructure.

2.2 System Power Management

System power management involves moving the entire system into a low-power state, where it consumes a small amount of power while simultaneously conserving a relatively low response latency to the user. The system's power consumption, as well as the overhead incurred in placing the system back in the active state, depends on the state of the system. As a general rule, in Dynamic Power Management (DPM) techniques, the deeper the low-power state, the lower the power consumed by the system, but also, the higher the latency involved [32, 42, 47]. Regardless of the low-power state the system moves into, the running context is saved to volatile or non-volatile storage before the system is powered down and subsequently restored when the system is powered back up, preventing unnecessary shutdown and startup sequences.

The low-power states a system can enter into depend largely on the underlying platform and hardware architecture. Despite this, at least three states are commonly available: *standby, suspend, and hibernate. Standby* halts the processor and moves the devices to a low-power state, saving moderate power while still retaining a very low response latency – typically less than one second. *Suspend* (also known as suspend-to-RAM) places all devices in a deeper sleep state, except for main memory, which is placed in self-refresh mode so that its contents are not flushed. More time is

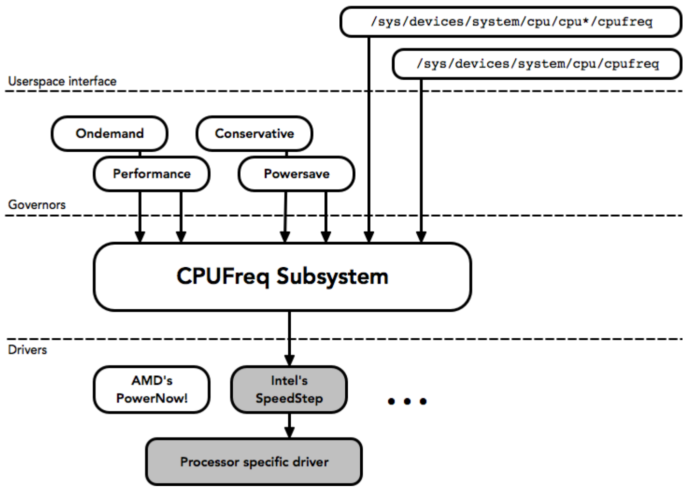


Fig. 2. The CPUFreq subsystem architecture.

required to return from suspend than from standby, but the latency still is mildly low at typically few seconds. *Hibernate* saves the most power by turning off the entire system after saving its running context to non-volatile storage (usually to disk). It also incurs the worst latency – around thirty seconds.

The Linux kernel provides two mechanisms as part of its power management infrastructure which are voltage and frequency scaling (DVFS) and processor shutdown (DPM). The use of these two techniques is pivotal in decreasing the platform's power and energy consumption and limiting its thermal output, as power dissipation produces heat as a byproduct. DVFS actions are carried out by the CPUFreq module, whereas DPM transitions are provided by the CPUIdle module.

2.2.1 The CPUFreq Subsystem. The CPUFreq subsystem [21] enables scaling processor frequencies at runtime within the Linux kernel. CPUFreq has been available since the 2.6.0 Linux kernel version. CPUFreq is composed mainly of a set of defined *governors* and low-level drivers, as shown in Figure 2. The subsystem complies with the ACPI standard [25], offering an interface to control processor performance states, or P-States (sometimes also called Operating Performance Points, or OPP). These power management states are specified by ACPI and translate into voltage and operating frequency pairs. P-States are numbered according to their associated speed and power levels; higher P-States represent slower processor speeds and lower power consumption. For example, a P3 state demands less power and runs more slowly than a P1 state. The number of available P-States is processor-dependent.

In addition to P-States, the ACPI specification also defines C-States for processor power management when the system idles. C-States offer an idle processor the ability to turn off unused components to save power, such as stopping the processor clock and bypassing hardware interrupts. As with P-States, the number and attributes of C-States are specific to the processor. Fully operational processors run in the C0 state. At higher C-States, the processor moves to deeper sleep states, and more components are shut down.

CPUFreq *governors* implement a particular policy for controlling how the processor frequency and voltage levels are scaled. These are designed with different goals in mind: some governors deliberately switch to low frequencies in systems with low energy budgets (such as laptops running on battery power). Others governors focus on short bursts of high processor utilization. It follows that selecting the most adequate governor is highly dependent on how the system is optimized, whether it be for power, for performance, or some other type of hybrid or predictive approach. The CPUFreq subsystem has four default governors[1]:

- **The performance governor.** This governor statically sets the processor speed to the highest available frequency. As such, it is most useful when the platform is required to operate at peak performance. Also, transition latency times are nonexistent, since the switching occurs only once. As a disadvantage, running at top speed for long periods quickly leads to overheating.
- **The powersave governor.** Similar to the performance governor, with the difference that this governor sets the processor speed to the lowest available frequency instead. This results in a great deal of power savings at the expense of a increased execution times. Nevertheless, using this governor might extend the application completion times beyond energy efficiency, as running the application for a longer time while consuming less power might lead to more energy consumption than doing so at faster rates. For example, when the *energy-efficient frequency* [3] is higher than the system's lowest frequency level."
- **The ondemand governor.** Introduced in the 2.6.10 Linux kernel version, this governor is one of the first attempts to integrate dynamic frequency scaling with processor utilization tracking. The ondemand governor automatically selects the highest available frequency when the average processor load rises above a predefined threshold. Average load estimation is triggered by the task scheduler. The governor tracks changes in load when a certain period has elapsed and sets the frequency accordingly.
  When the average processor utilization rises above the threshold specified by the user, the ondemand governor increases the frequency to the maximum allowable value. If the average processor utilization falls below a second lower bound, the governor decreases the frequency in steps, setting the processor to run at the next lowest frequency. The governor stops decreasing the frequency when it hits the minimum allowable value. At predefined intervals, the current processor utilization is queried and the same procedure is applied to dynamically adjust the frequency. The governor can optionally reject the utilization contributed by *niced* processes[2]; any processes that run with a positive nice value will not be accounted for when determining the current processor utilization. In addition, the ondemand governor considers a tunable parameter that modifies its behavior to save more power by reducing the target frequency by a specified percentage.
- **The conservative governor.** Similar to the ondemand governor, the conservative governor also works by dynamically adjusting frequencies based on average processor utilization. The difference between the two is that the latter increases and decreases processor speed more gradually, resulting in a more finely tuned frequency adjustment. The governor was introduced in the 2.6.12 Linux kernel version.
  Unlike the ondemand governor, the conservative governor always adjusts the frequency up or down in steps. If the average processor utilization rises above a user-defined threshold, the governor steps up the processor speed to the next highest frequency below or equal to the maximum value. Correspondingly, the governor steps down the frequency when the average utilization falls below a second, also user-defined threshold. After each predefined interval, the current processor utilization is checked and the procedure is repeated. The utilization contribution of processes with a positive nice value can optionally be ignored

[1] CPUFreq actually has five default governors, the fifth of these being the *userspace* governor. Here the authors focus only on governors that perform frequency adjustments.
[2] Within the Linux kernel task scheduler, processes with large *nice* values have lower priorities.

when computing overall processor utilization (as in the ondemand governor). Therefore, niced processes will not cause the processor frequency to increase.

Although some CPUFreq governors rely on information provided by the task scheduler, the modular and autonomous design of governors makes it challenging to merge the power management policies they enforce into the task scheduler. The desire to infuse the task scheduler with processor power management capabilities, particularly processor frequency scaling, has been present for some time within the Linux community [11]. Indeed, some initial efforts have been made to more closely link the scheduler to the processors power management infrastructure, paving the way for a more complete solution. For example, a patch introducing a new CPUFreq governor, labeled *schedutil*[3] by Wysocki, has recently been merged into the mainline 4.7 Linux kernel release. The schedutil governor benefits from a callback function that the scheduler uses whenever it computes its new load average based on the currently scheduled tasks. This simple scheme makes for a much more precise frequency adjustment.

2.2.2 The CPUIdle Subsystem. When older systems became idle, a low-priority "idle" task running a busy-wait loop was scheduled until some other task requested service [15]. If the system ran a light workload with many idle periods, busy-waiting resulted in CPU cycles being burned and power being drained without any useful work being performed. This situation motivated hardware architects to equip next-generation platforms with a variety of low-power and idle states to switch to when there are no jobs. These low-power states (previously mentioned in this section as C-States) vary in their associated power consumption and entry-exit latency.

Within the Linux kernel, the CPUIdle subsystem [38] was developed for handling low-power states. Much like the CPUFreq module, CPUIdle offers a generic, hardware independent interface to handle the different processor idle states, and installs a layer of abstraction between policies and hardware drivers. Policies, as with CPUFreq, are enforced by governors. Governors implement the algorithm in charge of selecting the most appropriate idle state for a particular situation. For instance, a system running a performance-sensitive application (such as video playback) might not be able to afford staying idle for too long; in this case, the CPUIdle governor should select a low-latency idle state.

2.3 Device Power Management

Device power management [34] is the other component of the Linux kernel power management infrastructure. It is concerned with putting peripheral devices into low-power modes at runtime or when the system itself engages in low-power mode. Borrowing from the ACPI specification, device power management defines low-power states for peripherals as D-States (ranging from D0 to D3) as well as a mechanism for controlling those states. Each D-State involves a tradeoff between the amount of power consumed by the device and how functional it is. As with P- and C-States, higher D-States represent lower power consumption but more device context lost. All devices implicitly support D0 (when the device is fully powered) and D3 (when the device is off) states; D1 and D2 are optional.

Device power management is made possible by a new driver model introduced in the 2.5 Linux kernel version. This new model allows the systems power management infrastructure to interface with all available device drivers, regardless of which bus or physical device the driver controls. In addition, the model establishes parent-child relationships between system devices to help sort out power transition sequencing issues that arise when one driver depends on another. That is,

[3] See `https://patchwork.kernel.org/patch/8477261`.

before powering down a certain device (parent), the system must first power down its dependents (children).

3 INCORPORATING ENERGY MANAGEMENT FEATURES IN LITMUS^{RT}

This section provides a brief description of LITMUS^{RT}, a real-time extension for the Linux kernel, which significantly simplifies the implementation of real-time multiprocessor scheduling policies in the form of plugins, offers a programming library for developing custom real-time applications, and provides a kernel overhead measurement toolkit. Then, the development carried out to facilitate the integration of LITMUS^{RT} and the Linux CPUFreq module is described, which permits the implemented multiprocessor scheduling plugins to perform frequency adjustments at runtime.

3.1 LITMUS^{RT}

LITMUS^{RT} [9, 12] is a real-time extension for the Linux kernel that aims to provide a useful experimental platform for applied research into real-time systems assuming realistic conditions. LITMUS^{RT} was conceived within the real-time systems group of the UNC at Chapel Hill in 2006, and has been actively maintained and developed ever since (as of 2018). Its main focus is on real-time multiprocessor scheduling and synchronization algorithms. The current version of LITMUS^{RT} (2017.1) is built on top of the 4.9.30 Linux kernel release and supports Intel's 32-bit and 64-bit x86 architectures, as well as the ARMv6 architecture. LITMUS^{RT} has been embraced in a vast number of works [7, 8, 10, 22, 23, 27] researching numerous topics including real-time scheduling on GPUs, mixed-criticality real-time systems, adaptive real-time tasks, real-time scheduling with semi-partitioned reservations, hierarchical processor affinities, and many others[4].

3.1.1 Core Infrastructure. The LITMUS^{RT} architecture follows a modular pattern that decouples the development of scheduling policies (i.e., plugins) from changes introduced into the Linux kernel code. This additional code, as well as some reusable components and shared functionality, constitute the LITMUS^{RT} core infrastructure.

The LITMUS^{RT} core infrastructure installs an additional scheduling class in the Linux scheduling hierarchy on top of every other scheduling class (i.e., on top of Linux's `SCHED_FIFO` and `SCHED_RR` real-time scheduling classes, and the `SCHED_NORMAL` timesharing scheduling class), allowing LITMUS^{RT} to override Linux's normal scheduling decisions. However, unlike regular Linux scheduling classes, the LITMUS^{RT} scheduling class does not actually implement any particular scheduling policy, delegating all scheduling decisions to the currently active scheduling plugin instead.

A LITMUS^{RT} task (i.e., a task conforming to the sporadic real-time task model enforced by LITMUS^{RT}) that is eligible for execution is always scheduled over any regular Linux task. When a LITMUS^{RT} task is released, it continues to run until it is blocked or preempted by another LITMUS^{RT} task of higher priority. Regular Linux tasks are treated as best-effort tasks with statically low priority. If a LITMUS^{RT} task is runnable, no regular Linux task can run until the former becomes unrunnable. When there are no LITMUS^{RT} tasks, the OS continues scheduling every other task as usual, which allows the system to act as a normal Linux distribution in the absence of real-time tasks.

The LITMUS^{RT} core infrastructure also provides reusable *ready* and *release queues* for managing the admitted real-time tasks. The ready queue implements the mechanism to order ready jobs, whereas the release queue implements the mechanism to queue jobs for future time-based releases. The ready queue is implemented as a binomial heap [45], a tree-like data structure that supports the merging of two heaps in $O(\log n)$ time. The release queue is implemented such that the worst-case

[4] Please refer to `https://wiki.litmus-rt.org/litmus/Publications` for a comprehensive list of publications that use LITMUS^{RT} as base RTOS.

overhead for releasing multiple jobs simultaneously (e.g., on a hyperperiod boundary) is minimized. The release queue abstraction maps time instants to heaps, and queues release jobs into the heap that corresponds to their release time. When it is time for the jobs to be released, the release queue simply merges the release heap with the ready heap. Both queues are abstracted in the form of a reusable component known as *real-time domain*. Depending on the scheduling policy, real-time domains are private or shared. For example, a partitioned policy (where each processor has its own ready and release queues) compels each processor to hold its own exclusive real-time domain, whereas in a global policy, a single real-time domain instance is shared between all processors.

3.1.2 Scheduling Plugins. In LITMUS^{RT}, the actual scheduling decisions are taken by *scheduling plugins*, entities living in kernelspace with access to the LITMUS^{RT} core infrastructure. The modular design of LITMUS^{RT} allows rapid prototyping of new scheduling policies without exposure to the full complexity of the Linux kernel. LITMUS^{RT} is equipped with a few stock scheduling plugins to implement particular multiprocessor scheduling policies, including *partitioned* EDF (P-EDF), partitioned RM (known within LITMUS^{RT} as *partitioned fixed-priority*, or P-FP), *global* EDF (G-EDF), and *clustered* EDF (C-EDF).

The LITMUS^{RT} scheduling class invokes the *active* plugin when a scheduling decision must be made, e.g., when a job is released or a task must be rescheduled. Before launching any real-time task, however, the user must select one of the included plugins by means of the `setsched` utility (the default scheduling plugin simply defers all scheduling decisions to Linux's `SCHED_NORMAL` scheduling class). Once a LITMUS^{RT} scheduling plugin has been selected, the user can launch individual sporadic real-time tasks using the `rtspin` or `rt_launch` utilities or prepare a sporadic real-time task to be released at the firing of a signal (with the `release_ts` utility). When released, the task is scheduled following the selected multiprocessor scheduling policy until the experiment is completed. The active scheduling plugin can be switched at runtime by activating another of the included plugins. However, plugin switching can only occur in the absence of real-time tasks (i.e., before a task is configured, or after its execution is complete).

Developing scheduling plugins in LITMUS^{RT} is akin to a certain extent. LITMUS^{RT} offers a plugin interface consisting of several object-oriented methods that can be grouped by functionality. Some of these methods handle events that correspond to scheduling points in the scheduling theory, such as job release, preemption, completion, *etc.* Other methods relate to initialization, bookkeeping, and cleanup activities. To modify the behavior of a scheduling plugin in the occurrence of certain event, the developer must insert the desired behavior within the method responsible for handling such event. To include a new, custom-made scheduling plugin in a particular LITMUS^{RT} kernel release, the plugin must be compiled and linked along with the kernel[5].

3.2 Extending LITMUS^{RT} with Support for DVFS

Despite being feature-rich, the main LITMUS^{RT} kernel distribution does not yet include (to the best of the author's knowledge) any means to control the performance states offered by the Linux power management infrastructure. This opens up an interesting line of research from which real-time application developers and researchers alike can benefit. In particular, an experimental platform based on an open source OS such as Linux, with the capacity to regulate its own energy consumption while supporting workloads with stringent timing constraints, would allow researchers to validate their proposals and evaluate their performance on real hardware quickly and cheaply.

[5] Providing detailed instructions for developing new plugins is beyond the scope of this work. Please refer to `http://www.litmus-rt.org/create_plugin/create_plugin.html` for a step-by-step guide to implementing a new LITMUS^{RT} scheduling plugin from scratch.

Working with LITMUS$^{\text{RT}}$ as base RTOS allows reusing several components from its existing code base, which already implements many of the features required for this research. In addition, contrary to other patch-based real-time extensions for the Linux kernel, such as [2, 18] - whose rare adoption by real-time research groups looking for a platform to evaluate their developments has led maintainers to withdraw their support - LITMUS$^{\text{RT}}$ is actively maintained and updated, offering the ability to continue extending this line of research with further developments while knowing that support is readily available.

The focus in this research is on the implementation and evaluation of energy-efficient multiprocessor real-time schedulers with global DVFS capabilities. Indeed, schedulers with these characteristics are known to be effective at lowering the platforms energy demand [5]. A recurrent pattern in real-time scheduling algorithms exploiting processor slowdown is to compute the lowest speed that still guarantees the fulfillment of timing constraints and then adjust the frequency to the computed value at specific points in the schedule, at job release or completion, for example.

3.2.1 Design. As discussed above, the CPUFreq infrastructure requires the client to invoke their interface to switch a frequency from a context where both blocking and hardware interrupts are enabled. Unfortunately, when LITMUS$^{\text{RT}}$ scheduling plugins carry out their scheduling decisions (for job release, preemption, and completion), the kernel is executing code paths that are particularly sensitive to performance with hardware interrupts disabled. This is contrary to the goal of allowing the implemented real-time multiprocessor scheduling plugins to perform frequency scaling actions directly from the scheduling decision handler. However, the problem can be avoided altogether through a simple scheme.

The approach here presented is similar in nature to the *dedicated interrupt handling* [9] (sometimes referred to as *interrupt shielding*) scheme already implemented in LITMUS$^{\text{RT}}$ in which a single processor is reserved as the *system* processor and becomes solely responsible for system management tasks, including device and timer interrupt handling. For the purposes of energy management, the system processor is designated to receive requests and subsequently perform frequency adjustments at runtime. The remaining *application* processors (which also encompasses the systems processor) on the platform perform scheduling decisions and submit frequency adjustments to the processor designated to this activity.

Since application processors carry out job scheduling, they have direct access to the effective processor requirement (utilization) being requested by the workload. This information is important to accurately compute the frequency level needed to preserve the feasibility of all processors (processor feasibility is guaranteed as long as its operating frequency is no smaller than its total utilization). Application processors convey their load value to the systems processor by means of a message passing mechanism, where messages are attended to on a *first-in first-out* basis. Figure 3 shows a conceptual overview of the framework, considering a quad-core multiprocessor platform.

The systems processor itself delegates the actual frequency switching to a kernel-level event handler specifically maintained for this purpose. The handler is implemented as a *kthread*[6] scheduled in the systems processor. The handler operates by constantly switching between the active and inactive states. When a processor requests a frequency adjustment, the handler is activated and forwards the application processor's request to switch the frequency to the CPUFreq driver in use. When the frequency switching is complete, the handler checks if any other frequency adjustment is in place. If so, it begins a new transaction with the CPUFreq driver; if not, it returns to the inactive state. This workflow enables the interface between scheduling plugins and the CPUFreq subsystem,

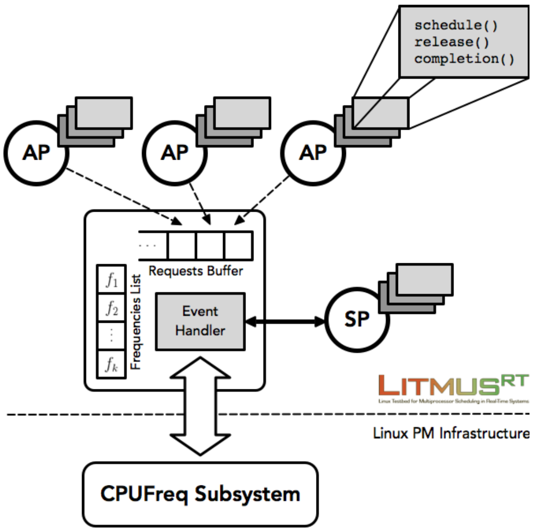


Fig. 3. High-level overview of frequency scaling in LITMUS^{RT}.

as the delegate (the dedicated kernel thread) carries out the requested frequency switching on behalf of the requesting application processors from process context.

The event handler arranges frequency scaling requests when they occur concurrently on different application processors. Specifically, when any two processors attempt to readjust the frequency to different levels at *almost* the same time (since it is impossible for the two events to occur at *exactly* the same time, given the limited resolution of OSes time tracking mechanisms), the handler serializes the requests by enacting the first to arrive and queueing the second to arrive. To allow voltage and frequency levels on the platform to stabilize following an adjustment, a minimum period of 200 microseconds (or the minimum allowed by the CPUFreq driver in use) is enforced between any two adjustments. When this period elapses, the handler evaluates if another frequency adjustment is in place, for instance, if the scheduled task set requires the platform to speed up to meet timing constraints.

The solution presented draws inspiration from the scheduler design pioneered in [13], in which coordination between processors is managed by a designated moderator, and from the CPUFreq governor described in [11], which manages to integrate more closely with the Linux task scheduler by offering a set of callbacks that the scheduler can invoke. While conceptually simple, this new element installed between the systems processor and the application processors allows arming LITMUS^{RT} scheduling plugins with CPUFreq governor-like capabilities.

3.2.2 Implementation. Implementation for the systems processor side of the workflow described above is abstracted in a single reusable component attached to the LITMUS^{RT} core infrastructure. The component contains a simple API to LITMUS^{RT} scheduling plugin running on application processors, which is used to perform various operations. The component implements four main operations, which are intended to be used in specific contexts:

[6] *Kernel threads* [31] (often referred to as *kernel daemons*, or simply *kthreads*) are standard processes provided by the Linux kernel useful for carrying out some operations in the background. They are standard processes living in kernel space, with the main difference between these and regular processes being that kernel threads do not have an address space. The Linux kernel employs kernel threads for various purposes, ranging from data synchronization of RAM to helping the scheduler to distribute processes among CPUs to managing deferred actions.

- `rt_dvfs_init()`, which instantiates and initializes the kernel-level event handler at system boot time;
- `rt_dvfs_update_cpu_load()`, through which application processors communicate changes in their processor load (arising, for instance, from workload variability) to the systems processor;
- `rt_dvfs_issue_freq_update()`, through which application processors request frequency adjustments to the systems processor; and
- `rt_dvfs_exit()`, which disposes of the kernel thread and carries out cleanup actions when the system is shut down.

The event handler implements the main logic behind frequency switching by means of the `rt_dvfs_task_func()` function, although this procedure is not part of the API. Instead of sending a specific frequency level when requesting a frequency update, application processors pass their actual processor load to the event handler, which in return computes the most appropriate frequency level for the platform (e.g., the frequency level, which preserves the feasibility of the processor with the highest load). The event handler has exclusive access to a few pieces of code that define the state of the system with respect to operational frequency. For instance, the event handler retrieves frequency values from a lookup table, mapping operating frequency to processor load. It also keeps track of the platform's current operational frequency to avoid unnecessary frequency transitions when the current and the target frequency levels coincide, as well as the time of the last frequency transition in order to determine if enough time has elapsed. Communication between application processors and the event handler implementing frequency scaling is established through a *first-in first-out* (FIFO) buffer, which is protected from concurrent writes by a set of per-CPU locks. Each message passed to the systems processor (i.e., written to the FIFO buffer) includes the requester's ID and the time that the message was sent.

4 CASE STUDY: ENERGY-AWARE MULTIPROCESSOR SCHEDULING ALGORITHMS

This and subsequent sections report on a case study that demonstrates how the energy management framework presented in Sections 2 and 3 can be applied to the evaluation of energy-aware multiprocessor scheduling algorithms. This section is devoted to describing the energy-saving mechanisms employed by algorithms EDF$^{(k)}$ and CVFS, which serve as the objects under study.

The strategy followed by these algorithms is to select the lowest operating frequency at which tasks can be executed without risking missing their deadlines. As mentioned previously, reducing the platform's operating frequency (and its corresponding supply voltage) yields a significant reduction in its energy consumption, while simultaneously increasing the execution of tasks. Thus, the operating frequency must be carefully selected in order to avoid prolonging tasks beyond their deadlines.

We emphasize that our objective in this paper is *not* undertaking a comprehensive experimental valuation of existing energy-aware multiprocessor scheduling algorithms – there is a large number of representative algorithms that have been surveyed in recent research articles (e.g., [5]). We put our efforts in implementing at the Litmus kernel two well-known algorithms EDF$^{(k)}$ and CVFS as a proof of concept for the testbed that we developed. We hope that our testbed will form a basis for the kernel-level evaluation and implementation of many other energy-aware multiprocessing scheduling algorithms.

4.1 EDF$^{(k)}$

EDF$^{(k)}$ [26] is a priority-driven multiprocessor scheduling algorithm designed to overcome inherent limitations of G-EDF scheduling. EDF$^{(k)}$ performs better than EDF in that it can schedule all task sets schedulable by EDF, in addition to some other task sets that EDF may fail to schedule. It was first proposed by Goossens *et al.* [26] and later revisited by Nlis [36], the latter having developed an offline technique for determining the lowest processor frequency at which the workload can be executed across all processors without compromising feasibility.

The idea behind EDF$^{(k)}$ is to isolate high- and low-demand tasks from each other, as their interaction leads to losses in performance for EDF in a multiprocessor setting, a situation informally known as the *Dhall effect* [20]. EDF$^{(k)}$ splits the set of tasks into k subsets, $1 \leq k \leq m$, of "privileged" and non-privileged tasks. Priorities (privileges) are assigned according to task utilization; the $(k-1)$ highest-utilization tasks are assigned the highest priorities (and are, therefore, privileged), whereas the remaining $(n-k+1)$ tasks are assigned regular EDF priorities. Privileged tasks are then dispatched to their own dedicated processor, devoting $(m-k+1)$ processors to non-privileged tasks.

Task sets composed of a mix of relatively few high-utilization tasks and many low-utilization tasks are somewhat more easily supported by EDF$^{(k)}$. In the worst case, however, EDF$^{(k)}$ either degenerates into G-EDF (when $k = 1$) or acts as a rather unbalanced instance of P-EDF (when $k = m$, which implies that $(m-1)$ processors are occupied by a single privileged task and one processor is left for the remaining tasks).

Suppose that tasks in a sporadic real-time task set τ are indexed non-increasingly by utilization (i.e., for all $i, 1 \leq i < n, u_i \geq u_{i+1}$). Let $\tau^{(k)} = \{\tau_k, \tau_{k+1}, \ldots, \tau_n\}$ denote τ without the $(k-1)$ highest-utilization tasks. According to the procedure devised in [36], τ is schedulable by EDF$^{(k)}$ on an m-processor platform running at normalized speed s[7] if

$$s \geq \max\left\{u_1, u_k + \frac{U(\tau^{(k+1)})}{m-k+1}\right\} \tag{1}$$

The lowest value for s can be identified iteratively by finding the smallest value from all those computed by Expression 1 for k between 1 and m. Hence, the procedure's time complexity is $O(m)$.

4.2 CVFS

The *Coordinated Voltage and Frequency Scaling* (CVFS) [19] energy-aware multiprocessor scheduling algorithm is optimized for platforms on which processors share the same supply voltage and operating frequency. It explicitly addresses the single clock domain restriction to which processing cores within a single chip are constrained by setting the operating frequency of the entire platform to the highest level requested from among all cores. Furthermore, the algorithm benefits from under-utilization of the platform, in the form of both unused processor capacity and idle periods resulting from early task completions.

For a multiprocessor platform with global DVFS capabilities, CVFS consistently sets the shared operating speed s to the maximum processor share requested from all active processing units. Specifically, for a given sporadic real-time task set $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ partitioned across a computing platform $\pi = \{\pi_1, \pi_2, \ldots, \pi_m\}$, let ψ_i denote the subset of tasks from τ allocated to processor π_i. Let the processor utilization (load) requested for processor π_i by ψ_i be given by $U_i = \sum_{\tau_j \in \psi_i} u_j$. Assuming the EDF scheduling scheme, which is optimal for the uniprocessor case, CVFS preserves the feasibility of all active processing units by setting the operating frequency of the platform to $f = \max\{U_i\} \cdot f_{\max}$, which can be regarded as a *static* frequency selection, in the sense that the frequency is adjusted according to the worst-case computation requirement of the supported task set.

[7]The normalized speed s is defined as the ratio of the current frequency to the maximum frequency

CVFS further benefits from the observation that jobs of tasks in a sporadic real-time task set usually use much less than their worst-case time allotment at runtime. To exploit this situation, CVFS builds upon the well-known *cycle-conserving* EDF (cc-EDF) [40] algorithm, adapting it to multiprocessor environments with global DVFS capabilities. Like cc-EDF, CVFS enforces several runtime energy management rules for maintaining an accurate utilization estimation and reducing the frequency even further:

- In order to avoid compromising workload feasibility when a job τ_j^k is released, a conservative assumption is made by resetting the utilization due to τ_j^k to its worst-case, that is $\frac{C_j}{T_j}$.
- When a job τ_j^k is complete, the actual amount of processor cycles cc_j consumed by the job is compared to its worst-case specification. Any unused cycles allocated to τ_j^k are reclaimed by setting their utilizations to $\frac{cc_j}{T_j}$.

At any of these two scheduling events, the global frequency of the platform is readjusted according to the updated processor utilization requirements.

CVFS is further driven by the fact that some processing cores will operate at a higher frequency than that required to guarantee workload feasibility, since all cores are constrained to the same global operating frequency level. CVFS exploits this fact through a runtime optimization to refine the load estimation. The working principle is as follows: *a job completing a certain amount of work at a high frequency during a predefined interval can be seen as equivalent to completing a smaller amount of work at a lower frequency in the same amount of time.* In other words, CVFS regards the completion of a job on a core whose utilization is smaller than the largest job on the platform as an *early* completion, which allows lowering the core's effective utilization and (conceivably) the platform's frequency.

Specifically, suppose that a job τ_j^k allocated to core π_i executes in p contiguous chunks $\{e_1, e_2, \ldots, e_p\}$. Let at_x be the amount of time consumed by τ_j^k during chunk e_x (see Figure 4). The total workload executed c_j by τ_j^k is computed as $c_j = \sum_{k=1}^{p}(at_x \cdot U_i)$. This simple optimization allows for a more precise load estimation at job completions.

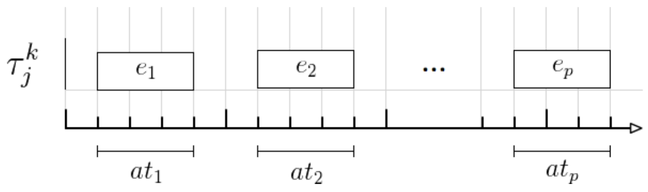


Fig. 4. A job executing in p contiguous chunks of execution.

4.3 Implementation Issues

Implementing the EDF$^{(k)}$ energy-aware scheduling policy is not as simple as it seems. EDF$^{(k)}$ might group processors into m different patterns, one for each value that k can take. However, it is not obvious at first how to do this in practice. The clustering approach implemented by the LITMUS$^{\text{RT}}$ stock *clustered* EDF (C-EDF) scheduling plugin was chosen here. However, as opposed to C-EDF, which groups processors into clusters of the same size around the different cache levels offered by the hardware[8], the EDF$^{(k)}$ scheduling plugin assembles processors into clusters of different sizes, as required by the algorithm (subsection 4.1). To accomplish this, the clustering functionality for EDF$^{(k)}$ was linked to the procedure for clustering processors already implemented in LITMUS$^{\text{RT}}$.

The user writes the desired clustering option (e.g., $k = 1$, $k = 2$, *etc.*) to a virtual file in the `/proc` file system. When the EDF$^{(k)}$ scheduling plugin is activated, it dynamically determines which processors to group based on the selected configuration. The frequency of the platform is then set to that computed by Expression 1.

The CVFS algorithm was implemented in LITMUS$^{\text{RT}}$ by building a scheduling plugin similar to the LITMUS$^{\text{RT}}$ stock *partitioned* EDF (P-EDF) plugin. Notably, the P-EDF and CVFS plugins differ in how they manage scheduling decisions at runtime. The CVFS plugin can accomodate the necessary operating frequency adjustments at job release and completion using the energy management infrastructure described in section 2.

Concurrent frequency adjustment requests are particularly troublesome. Since CVFS is a partitioned scheduler, different processors might attempt to adjust the frequency to different levels at the same time. For instance, two processors might attempt to both increase and decrease the frequency. In order to coordinate the frequency adjustment actions occurring concurrently on different processors, the CVFS plugin relies on the synchronization procedure enforced by the infrastructure, where updates are attended to in first-in first-out order (section 3). When two frequency adjustment requests arrive, the first is enacted immediately and the second is queued. Once the minimum period between any two adjustments has elapsed, the second request is enacted if it is in accordance to the CVFS energy management policy (i.e., if the requested frequency level is enough to guarantee the feasibility of the workload), and discarded otherwise.

5 EXPERIMENTAL EVALUATION OF ENERGY-AWARE MULTIPROCESSOR ALGORITHMS

This section presents and discusses the results of the measurements taken of the implemented energy-aware multiprocessor scheduling algorithms (EDF$^{(k)}$ and CVFS), with the goal of unveiling their performance trend in terms of energy consumption when running on real hardware. To accomplish this, the implementation makes explicit use of the infrastructure presented in section 2. The methodology followed for carrying out the evaluation is described, which includes a description of the hardware platform underlying the experiments, the benchmark used for stressing the schedulers, and the instrumentation equipment employed.

5.1 Platform

The prototypes for the schedulers considered for the case study were implemented in LITMUS$^{\text{RT}}$ version 2017.1, which is based on the 4.9 Linux kernel release. The system ran the Ubuntu 16.04 LTS “Xenial Xerus” Linux distribution.

Power dissipation measurements described in the following sections were taken on an Intel Core i7-2600[9] `SandyBridge` system. The i7-2600 is a quad-core 64-bit Chip Multiprocessor (CMP). The four processing cores in the chip run at a nominal frequency of 3.4 GHz, and always operate at the same speed (i.e., the four cores belong to the same *voltage island*).

The i7-2600 supports Enhanced Intel SpeedStep Technology, Intel’s implementation of dynamic processor frequency scaling that defines multiple voltage and frequency operating points (referred to as P-States in section 2). The i7-2600 clock can oscillate at fourteen different frequencies, from 2.1 to 3.4 GHz in steps of 100 MHz. At lower speeds, the processor consumes less power, but the workload execution time is longer.

[8] Under the C-EDF scheduling plugin, the user clusters processors based on the cache topology of the platform, for instance around the L1, L2, *etc.* cache levels. This makes the grouping of cores dependent on the architecture of the underlying hardware.
[9]See `https://ark.intel.com/products/52213/`.

5.2 Workload

In order to assess the performance of the implemented real-time energy-aware multiprocessor scheduling algorithms, an appropriate benchmark for exercising the aspects of the system of greatest interest was needed. Previous research efforts [4, 40, 43] focusing on DVFS techniques for managing the processor's energy consumption in a real-time setting have mostly studied the effect of their proposals on the execution of a CPU-bound workload, and therefore so does this work. The effectiveness of the implemented real-time multiprocessor scheduling algorithms on a memory bound[10] workload is deferred for future work.

The userspace interface available within LITMUS^{RT}was employed to build a suitable benchmark. The interface comprises the *liblitmus* library and accompanying tools to facilitate the writing of custom real-time tasks. The liblitmus library contains all the required system calls and definitions to interact with the kernel services that LITMUS^{RT} provides to real-time tasks (recall from section 2 that in LITMUS^{RT} a real-time task is one that has been admitted to the LITMUS^{RT} scheduling class).

A few steps are required to convert a regular task into a LITMUS^{RT} real-time task. All modifications to the target task must be done in its source code. Before declaring the actual activity that will be carried out, a few lines of code must be devoted to establishing the interaction between the task and the LITMUS^{RT} kernel by means of the userspace real-time interface. Most of the functions used for this purpose are simply system calls provided by LITMUS^{RT}. In particular, converting a regular task into a real-time task requires the use of the following set of system calls:

(i) `init_rt_task_param()`, which initializes the interface;
(ii) `set_rt_task_param()`, which sets the real-time parameters for each job that the target task will produce, including period, deadline, and execution budget;
(iii) `task_mode()`, which "transitions" the target task to real-time mode; and
(iv) `wait_for_ts_release()`, which blocks the task (whose status by now is real-time) until signaled from userspace to begin execution.

After performing the series of steps outlined above, the task is now ready to be launched when signaled by the `rt_launch` utility. When this happens, the real-time task begins execution. Within LITMUS^{RT}, real-time jobs are more of an accounting abstraction to keep track of the number of times that the real-time task has executed its main job loop, which are all the relevant instructions accomplished by the task; a "job" is therefore a single round of execution of all instructions within the boundaries of the task's main loop. Periodic execution of jobs is achieved by calling the `sleep_next_period()` function at the end of each job. This function triggers the kernelspace mechanism for moving the task from the ready to the release queue. Subsequently, a high-resolution timer is set to fire at the release time for the next job of the task, at which time LITMUS^{RT} will move it back to the ready queue and the task will again be eligible for execution. Figure 5 shows a code template for developing periodic real-time tasks, which demonstrates the use of all the functions mentioned above.

The benchmark prepared for exercising the platform attempts to replicate the use of a real-world CPU-intensive workload. The reasons for this are twofold: first, CPU-bound workloads are more sensitive to changes on the platform's operating frequency (the aspect to be analyzed), as these have a direct dependency on the processor's clock rate [1, 6, 28], and are more likely to bring about the processor's power consumption trend. Second, embedded real-time multiprocessor systems are typically host to compute-intensive tasks such as high-definition multimedia playback, digital

[10] The authors are aware of previous studies exploring the problem of ensuring performance while decreasing power consumption (e.g., [33]) considering both *cache sensitive* and *memory bound* workloads. These studies focus mostly on non-real-time computing contexts. However, their methodologies could serve as base for future evaluations in a real-time setting.

signal processing, and object and pattern recognition [29]. As such, compute-intensive workloads are a natural starting point for testing resource optimization approaches targeting the real-time domain.

With this in mind, the benchmark designed for the purposes of this work consists of the following two components:

- A subroutine for computing an LUP decomposition for a reasonably large system of linear equations [16], whit the aim of representing a subset of the operations commonly performed in machine learning, data analysis, and computer vision applications.
- A subroutine for producing the discrete Fourier transform of a sequence of values by means of an iterative Fast Fourier Transform (FFT) algorithm [16, 41], whit the aim of representing data filtering and signal processing applications.

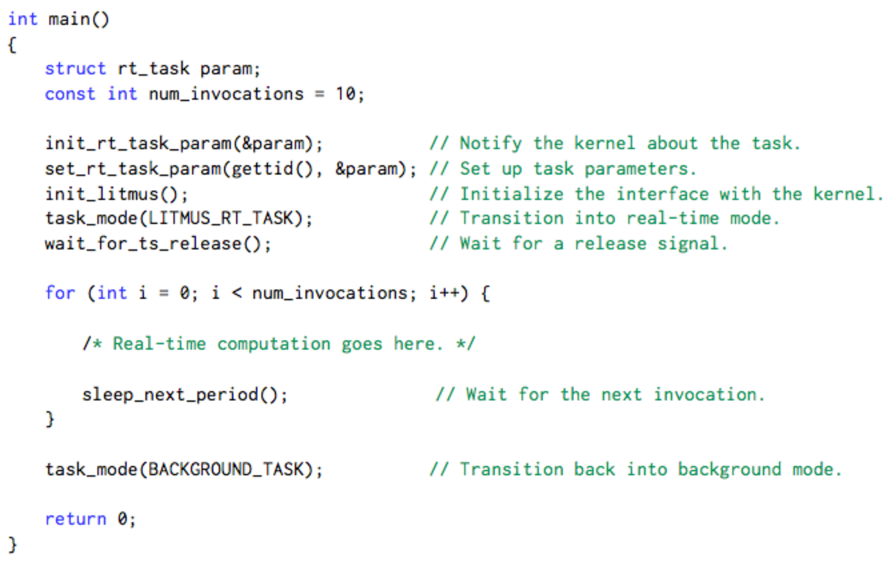
5.3 Methodology

In what follows, the steps taken to build the experiments (based on the workload presented in the previous section) and measuring the power consumption of our experimental platform are described. Figure 6 presents an illustration of the different stages involved in the evaluation.

5.4 Experiments

The experimental evaluation targeted four different scheduling policies: LITMUS$^{\mathrm{RT}}$'s stock *partitioned* EDF (P-EDF), EDF$^{(k)}$ (without frequency scaling), CVFS, and EDF$^{(k)}$. The first two carried out the scheduling of tasks while running at *full* speed, whereas the other two did so while applying their respective energy management actions. The two schedulers not implementing any particular energy management policy in our evaluation are included mainly for comparison purposes.

In the context of energy-efficient scheduling algorithms for real-time systems, power consumption and processor utilization are easily recognized as being correlated to one another, as a lower processor utilization naturally leads to long slack periods arising from the spare processor capacity, which the algorithm is likely to exploit in some way to decrease the platform's power consumption



```
int main()
{
    struct rt_task param;
    const int num_invocations = 10;

    init_rt_task_param(&param);           // Notify the kernel about the task.
    set_rt_task_param(gettid(), &param);  // Set up task parameters.
    init_litmus();                        // Initialize the interface with the kernel.
    task_mode(LITMUS_RT_TASK);            // Transition into real-time mode.
    wait_for_ts_release();                // Wait for a release signal.

    for (int i = 0; i < num_invocations; i++) {

        /* Real-time computation goes here. */

        sleep_next_period();              // Wait for the next invocation.
    }

    task_mode(BACKGROUND_TASK);           // Transition back into background mode.

    return 0;
}
```

Fig. 5. Real-time task boilerplate code.

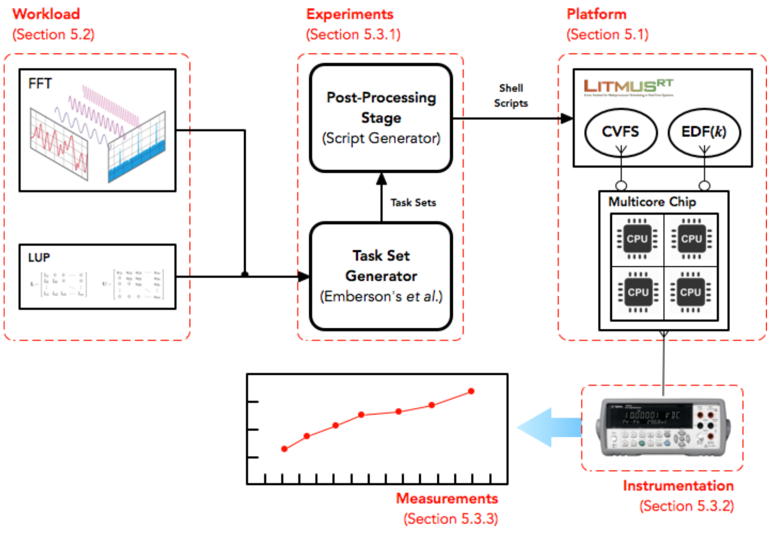


Fig. 6. Flowchart illustrating the followed steps in the evaluation.

[29]. To align with previous studies, the experiments carried out in this study likewise considered processor utilization as the primary parameter for revealing the performance of the tested scheduling algorithms in terms of energy consumption. The role of the size of the sporadic real-time task sets used for exercising the schedulers is yet to be determined.

Workloads with total number of tasks ranging from $n \in \{6m, 8m, 10m, 12m\}$, where $m = 4$ were considered. It was decided to execute sets with a large number of tasks so that the implemented algorithms had a better chance of success when partitioning the workload [10]. For each number of tasks n, sporadic real-time task sets with total *normalized utilization* U across $U \in \{20\%, 30\%, \ldots, 80\%\}$ were considered. Normalized utilization refers to the mean processor share requested to each processor on the platform, i.e, to the quantity $\frac{U_{tot}}{m}$. For each n and U pair, ten random task sets were generated (for a total of 280 task sets) using the task set generator from Emberson *et al.* [24]. The generator produces task sets with a given number of tasks, whose cumulative utilization adds up to a given utilization value. Each generated task is assigned a uniformly distributed utilization u_i and a period T_i chosen at random from the set $\{10, 20, 25, 40, 50, 100, 125, 200, 500, 1000\}$ (in milliseconds), which are comparable to those found in actual real-time workloads [29]. The worst-case execution requirement for each task is computed as $C_i = u_i \cdot T_i$. All tasks were assumed to execute up to their WCET.

Energy-aware multiprocessor approaches based on partitioned scheduling generally perform better when the workload is split evenly. For partitioned schedulers, the generated task sets were divided using the *worst-fit decreasing* (WFD) [30] heuristic, which is known to generate better balanced partitions. A task set was deemed *valid* if WFD was able to partition the set successfully. Invalid task sets were simply discarded and new ones were generated until the target 280 task sets were produced. For EDF$^{(k)}$ schedulers, a task set was considered valid if EDF$^{(k)}$ managed to cluster the tasks in the set following the procedure outlined in section 4. Figure 7 summarizes the experiment generation procedure.

As opposed to the other scheduling policies, CVFS operates by dynamically reclaiming unused processor capacity in the form of *dynamic slack*, which stems from tasks finishing their execution earlier than expected. In order to capture the impact of dynamic workload variability on the

```
generate_experiments()
    set $\Gamma \leftarrow \emptyset$:
    for each $n \in \{6m, 8m, 10m, 12m\}$
        for each $U \in \{20\%, 30\%, \ldots, 80\%\}$:
            for $count \leftarrow 1, 2, \ldots, 10$:
                set $valid \leftarrow \perp$
                While **not** $valid$:
                    set $\tau \leftarrow$ generate_taskset($n, U$)
                    set $valid \leftarrow$ determine if the task set is valid
    return $\Gamma$
```

Fig. 7. Pseudocode for generating experiments.

performance of CVFS, 3 more sets of task systems were generated (with 280 task systems each) specifically for CVFS, in which tasks randomly underrun their WCET by a factor of 10%, 20%, and 30%. The partitioning of these sets was carried out in the same manner.

Each generated experiment was post-processed and translated into an executable shell script containing the specification of the task set, the scheduling plugin that handled the task set at runtime, and the duration of the experiment. Each task was mapped to one of the two CPU intensive workloads described in subsection 5.2, which were developed specifically for the purposes of experimentation. Each task set was traced for 20 seconds. In total, 1,960 sporadic real-time task sets were executed and measured over more than 20 hours of continuous real-time execution and power measurement.

5.5 Instrumentation

Measuring the exact power usage of a processor is notoriously difficult. Modern processors use hundreds of pins and multiple interconnect layers for power and ground lines [39]. These lines are further distributed within the chip between the many processor components, some of which have uneven power requirements. Therefore, accurately determining the processor's power consumption under full consideration of its internal architectural traits would require the use of highly specialized and expensive equipment. Instead, a much simpler approach is followed that allows approximating the actual numbers for the power being drawn at a reasonable cost.

To characterize the processor's power consumption, the electrical current passing through it must first be determined. One way to accomplish this is to intercept the line delivering power to the chip and insert a measurement device to monitor the current flowing through this line. However, the current can rise to levels that are unbearable for the measurement instrument during instances when the processor is running at peak performance, resulting in incorrect readings or even damage to the instrument. An alternative, safer solution involves inserting a *shunt* (a manganin resistor of accurately known resistance) between the processor and its power supply. The voltage drop across a shunt is proportional to the current flowing through it. Given a shunt's resistance, it is easy to compute the value of the current passing through the circuit where the shunt is installed on using Ohm's law. The placement of a shunt resistor has an almost negligible effect on the circuit, as the shunt normally offers very small electrical resistance. Once the current passing through the processor has been determined, its power draw is also easily computed as the product of the current and the voltage being fed to the processor.

The above described method was chosen because it has the pragmatic benefit of being clean, simple, and accurate enough for most purposes. The voltage drop was measured across a low-resistance shunt resistor installed in series between the processor and the power delivery line

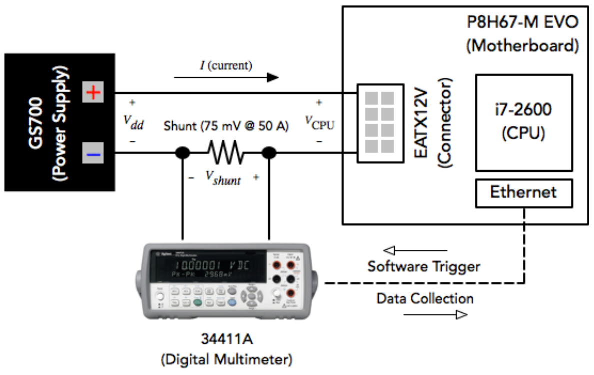


Fig. 8. Physical setup for measuring power consumption.

coming from the power supply. Shunts were rated by maximum current and voltage drop at that current. The voltage drop of the shunt employed was 75 mV at a maximum current of 50 A. The resistance offered by the shunt was therefore 1.5 mΩ. The motherboard holding the i7-2600 chip, an Asus P8H67-M EVO[11], used an eight-pin EATX12V connection to deliver power to the processor. The connection consisted of four DC +12 V wires and four ground wires coming from a Corsair GS700 power supply[12]. To improve the accuracy of the voltage readings, the shunt was inserted as close to the ground leg of the circuit as possible [17]. Figure 8 shows the physical interconnection of the power supply, processor, and shunt device.

Let V_{dd}, V_{CPU}, and V_{shunt} denote the voltage delivered by the power supply, the voltage drop at the CPU, and the voltage drop at the shunt device, respectively (Figure 8). R denotes the resistance offered by the shunt, and I is the current flowing through the circuit. The power dissipation of the CPU can be equated to the product of its voltage and current consumption, $P_{\mathrm{CPU}} = V_{\mathrm{CPU}} \cdot I$. Based on basic circuit theory, $V_{\mathrm{CPU}} = V_{dd} - V_{shunt}$. The value of the current being drawn can be approximated to $I = \frac{V_{shunt}}{R}$. These two equations allow expressing the power consumption for the CPU as a function of the shunt's voltage drop:

$$P_{\mathrm{CPU}} = V_{\mathrm{CPU}} \cdot I = (V_{dd} - V_{shunt}) \left(\frac{V_{shunt}}{R} \right) \tag{2}$$

A Keysight 34411A 6 1/2-digit multimeter [44] was used to measure the voltage drop across the installed shunt resistor (Figure 8). The 34411A is a high-performance Digital Multimeter (DMM) capable of taking measurements over a fixed period without user intervention. The instrument can be connected to a computer by means of the USB, LAN, or GPIB interface. The multimeter can be triggered by software and has the capacity to store the acquired data in its internal non-volatile memory for later retrieval. Readings were automated by means of a dedicated script consisting mostly of commands that conform to the SCPI[13] standard. The script was deployed prior to the execution of each experiment, and was responsible for configuring the instrument, triggering the acquisition of data, and retrieving the collected measurement samples.

[11]See `https://www.asus.com/Motherboards/P8H67M-EVO`.
[12]See `http://www.corsair.com/en-us/gs700w`.
[13] The *Standard Commands for Programmable Instruments* (SCPI) specification defines the syntax rules and conventions used in controlling programmable test and measurement devices. The 34411A complies with SCPI, which allows the instrument to be programmed using simple, generic commands.

The experiments were performed while measuring the actual power consumption of the test bench, following the measurement procedure described above. Each experiment lasted for 20 seconds, during which voltage readings were collected at a rate of $1,000$ readings per second. The energy consumption of a running sporadic real-time task set was computed by approximating the integral of the power consumed over time using the Riemann sum. Specifically, let E_{CPU} denote the energy consumed by the processor over a period of time T, and x_i be the set of voltage readings recorded by the multimeter. Any two consecutive voltage readings are separated from each other by a $\left(\frac{1}{1000}\right)^{\text{th}}$ of a second. Expression 2 allows determining the power consumed by the processor across $T = 20$, by plugging to it each collected voltage reading. Hence, by the Riemann sum,

$$E_{\text{CPU}} = \int_0^T P_{\text{CPU}} \cdot dt \approx \frac{1}{1000} \cdot \sum_{i=1}^{n} (V_{dd} - x_i) \left(\frac{x_i}{R}\right) \tag{3}$$

The energy consumption numbers reported in Figure 10 through Figure 12 reflect the average of ten measurements (see subsection 5.4). For all experiments, more than 450 MB of trace data were recorded containing more than $50,000,000$ measurement samples.

5.6 Measurements and Observations

Once the entire experimental flow was set up and deployed, it was possible to acquire meaningful measurement data to evaluate the overall energy consumption resulting from the execution of sporadic real-time task sets under different energy-aware policies.

Given the short separation allowed by the measurement device between any two consecutive readings, it was possible to capture very slight variations in power consumption, which aids in gaining a deeper understanding of the implemented scheduler's runtime behavior. An interesting pattern was noted when the computing platform was underutilized. The power measurements for sporadic task sets whose normalized utilization is markedly low exhibit a bimodally distributed arrangement. This situation might be explained by the likelihood of sudden short bursts of activity being interspersed with long periods of idleness. The distribution of the data might reflect the platform's power consumption when alternating between the active and idle states. As the processor utilization increases, the power dissipation begins to display a more familiar normal distribution pattern. This phenomenon might represent the scheduler's tendency to consistently select a specific operating frequency level to execute the workload. When the processing requirement increases, the schedulers opt for the lowest operating frequency that still guarantees meeting all timing requirements. Consequently, the execution of the workload is extended and slack times are shortened, resulting in a more regular power consumption at a constant operating frequency level.

Figure 9 illustrates the effect described above, showing the power measurement distribution for task sets of varying normalized utilization scheduled under CVFS. In the figure, the blue histogram represents the power measurements distribution for a task set with a 20% total normalized utilization, whereas the red histogram represents that of an 80% utilization task set.

The power consumption trends of both CVFS and EDF$^{(k)}$ (described in section 4) were profiled while scheduling sporadic real-time task sets of increasing worst-case processor utilization. Both algorithms were designed for multiprocessor platforms where all processing units are constrained to operate at the same voltage and frequency level. The algorithms differ, however, in the power saving measures they take to lower the platform's overall energy consumption. EDF$^{(k)}$ relies on a static frequency selection, while CVFS benefits from tasks finishing their execution earlier than expected. The actual processor power dissipation for P-EDF and EDF$^{(k)}$ (running at full speed) were also measured to provide a baseline for their energy-efficient counterparts. Figure 10 through Figure 12 present the performance of the schedulers with respect to energy consumption. In the

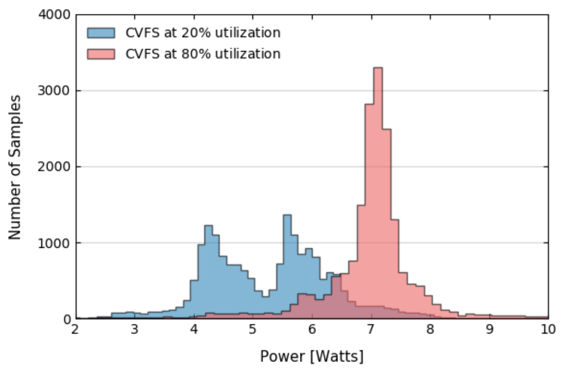


Fig. 9. Power measurements for different utilization levels.

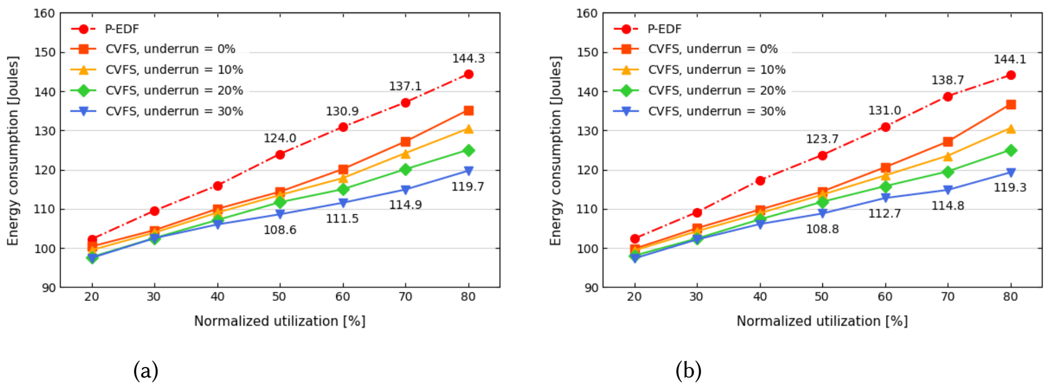


Fig. 10. Energy measurements for CVFS while scheduling sporadic real-time task sets with a number of tasks (a) $n = 6m = 24$ (b) $n = 8m = 32$.

figures for CVFS, "underrun" relates to the deviation of the actual execution time of tasks from its worst-case. For instance, a task with a underrun value of 10% is likely to run for 90% of its WCET.

Observation 1. CVFS *dominates* P-EDF *in terms of energy consumption in all tested scenarios, and even more so when the actual case execution time of tasks declines.* When the overall processing requirement is low, both schemes display similar energy consumption trends. As processor utilization increases, the effectiveness of CVFS's energy saving features becomes apparent. The gap between CVFS and P-EDF gradually widens as the platform becomes more occupied, achieving an $\approx$ 18% reduction in power consumption at a normalized utilization value of 80%. Interestingly, CVFS performance remains the same even with an increasing number of tasks. In fact, the energy consumption of a task set comprising $n = 6m = 24$ real-time tasks scheduled under CVFS is nearly identical to that of a task set comprising twice as many tasks (Figure 10 (a) and Figure 11 (b)), which indicates that energy is much more dependent on processor utilization than on the size[14] of the task set.

[14] It would be interesting to determine if this claim holds for extremely large task counts.

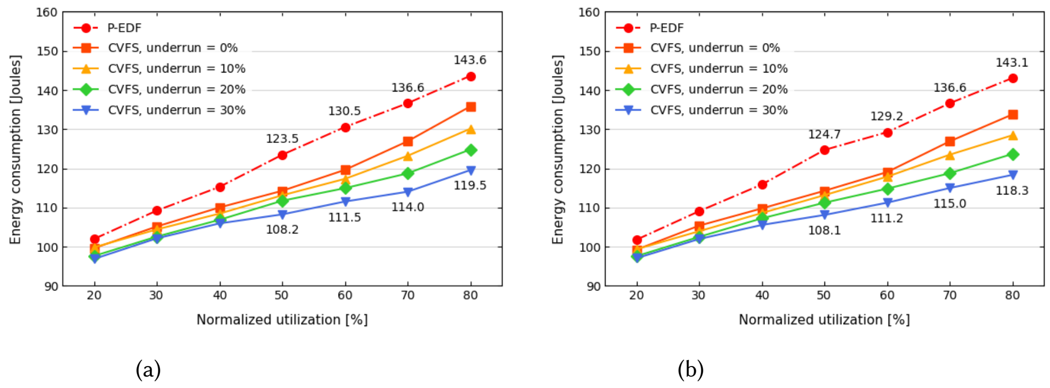


Fig. 11. Energy measurements for CVFS while scheduling sporadic real-time task sets with a number of tasks (a) $n = 10m = 40$ (b) $n = 12m = 48$.

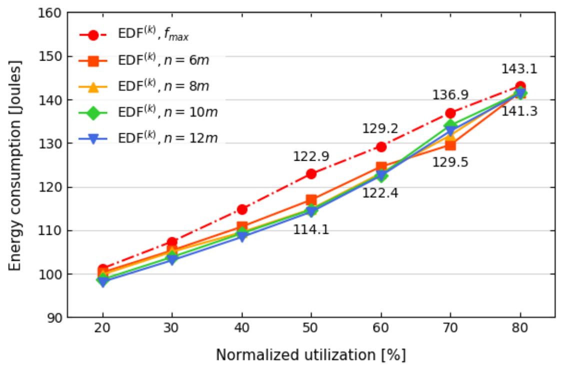


Fig. 12. Energy measurements for EDF$^{(k)}$ while scheduling task sets with varying numbers of tasks $n \in \{6m, 8m, 10m, 12m\}$, with $m = 4$.

Observation 2. EDF$^{(k)}$ *is unattractive for supporting high-utilization task sets from the energy consumption perspective.* Figure 12 shows the performance of EDF$^{(k)}$, both when performing a static frequency selection and running at the maximum allowable frequency. The first notable effect is the decreasing gains in terms of energy consumption at high normalized processor utilization values. When the processor share requested by the task set is low, EDF$^{(k)}$ performance is comparable to that of CVFS, mainly due to long periods of idleness taking place (Figure 10 (a) and Figure 12). Tasks with a low processor utilization demand require either a very short computation time or have a very long activation rate, which provides the platform with ample space to remain idle. Thus, independently of the operating frequency, the processors remain idle for longer periods, consuming much less energy[15]. As the processor utilization rises, however, the effectiveness of EDF$^{(k)}$ in terms of energy consumption decreases, approaching that of its equivalent executing the workload at full speed. This calls into question the viability of a static frequency selection technique

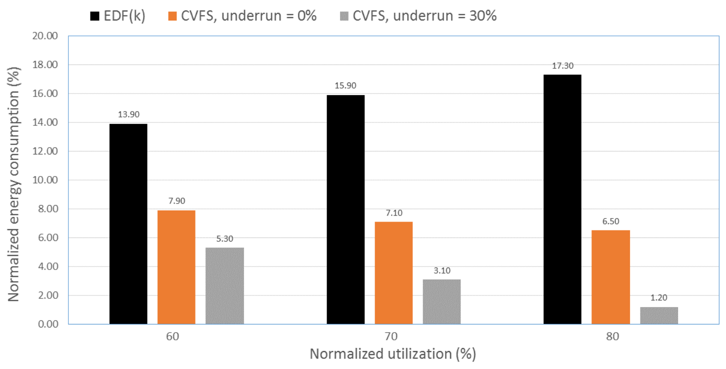


Fig. 13. Energy efficiency of CVFS and EDF$^{(k)}$ relative to their baselines.

when supporting sporadic real-time task set with a high processor utilization requirement on a multiprocessor platform constrained to global DVFS.

Figure 13 summarizes the relative energy efficiency for CVFS and EDF$^{(k)}$ when scheduling compute-intensive benchmarks with varying (high) normalized processor utilization. The numbers shown in the figure were computed by normalizing the performance of each scheduler with respect to their baseline schemes that execute tasks at $f_{\max}$ at all times. It can be seen from the figure that the relative performance of EDF$^{(k)}$ declines as normalized processor utilization approaches 80%, whereas CVFS (assuming no underrun) manages to reduce the platform's energy consumption by more than 6% at 80% of utilization.

Measuring overheads. When the infrastructure described in section 2 is employed to enact frequency adjustments at runtime, some extra overhead is to be expected. Compared to a scheduling policy unaware of the system's power and energy consumption, an energy-efficient policy will most assuredly incur additional overheads (for updating the platform's current load, requesting a frequency adjustment, *etc.*) when making energy saving decisions at runtime. To quantify the impact of such system overheads in a practical setting, additional workloads were run under CVFS and its baseline (P-EDF) on the quad-core platform while recording overhead samples. The low-level latency following a task release was selected as representative of the event-scheduling category, where dynamic frequency adjustment actions take place. A task set generation methodology similar to that described in subsection 5.4 was followed. 150 task sets were generated using the task set generator from Emberson *et al.*, each with a total utilization of either 75%, 80%, or 85%, and a number of tasks ranging from $n = 4m = 16$ to $10m = 40$ in steps of $2m$. Each task set was guaranteed to be feasible under partitioned scheduling and executed under both schedulers for 60 seconds. Overhead samples were collected using LITMUS$^{\mathrm{RT}}$'s *Feather-Trace* low-overhead tracing toolkit (section 2).

Observation 3. Runtime overheads incurred when making frequency adjustment decisions are relatively small in absolute terms. Figure 14 illustrates the task release overhead data observed by Feather-Trace, which measures overheads in terms of CPU cycles. On the experimental platform (where the CPU cycles counter runs at the platform's nominal frequency), 1μs corresponds roughly

[15] Notice, however, that this situation is not exclusive to the energy-efficient schedulers. The same holds for the schedulers not implementing any particular energy saving measure (Figure 10 through Figure 12).

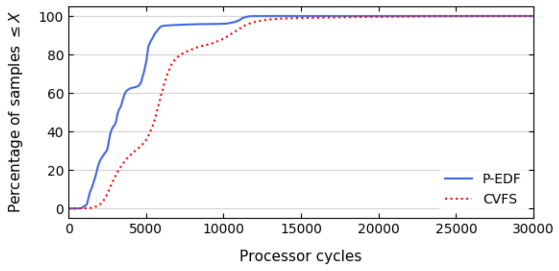


Fig. 14. Overhead incurred at task release under P-EDF and CVFS.

to 3, 400 cycles. In the figure, the y-axis denotes the fraction of all overhead data that measured at most the quantity of processor cycles marked on the x-axis. For instance, it can be seen from the figure that 90% of the overhead samples measured for P-EDF were fewer than 5, 000 cycles $\approx 1.47\mu$s. As expected, CVFS does incur higher overheads because of the latency involved in selecting the next frequency level and synchronizing the decision with respect to every other processing core on the platform (recall that the experimental testbed is based on a CMP featuring a global voltage and frequency level). However, the difference between both policies in terms of runtime overheads is relatively small, within the range of a few microseconds (90% of the overheads for CVFS measured at most 10, 000 cycles $\approx 2.94\mu$s). Even in the presence of additional overheads, these experiments with CVFS resulted in no observable instability with respect to timing requirements. Still, this exchange between performance and energy consumption must be validated by taking the characteristics of the supported application into account.

6 CONCLUSION

The recent availability of such open, feature-complete frameworks for supporting real-time workloads on multiprocessor and multicore platforms such as LITMUS^{RT} has made it possible to implement and evaluate real-time scheduling algorithms found in the literature on real hardware platforms. Moreover, the integration of these frameworks with the hardware and software support for energy management provided by most modern platforms makes it more feasible to implement energy-aware real-time multiprocessor scheduling algorithms with reasonable effort.

In this work, we have provided a means for achieving progress in this regard. Our implementation effort and measurement methodology was crucial in conducting a case study that presented the real performance (for our particular experimental setup) of two energy-aware real-time multiprocessor scheduling algorithms found in the literature. The results obtained in our study indicate that voltage/frequency scaling is indeed an effective means for achieving energy savings in multiprocessor settings where all processing units are constrained to run at a single global speed. In addition, the comparatively low runtime overheads introduced by dynamically performing operating frequency adjustments further confirms the practicality of the technique, even in contexts where stringent timing constraints must be preserved.

ACKNOWLEDGMENTS

The authors would like to thank to Project FOMIX CONACyT-Estado de Jalisco CIIoT (JAL-2015-03), for proving the funds for this research.

REFERENCES

[1] Ishfaq Ahmad and Sanjay Ranka. 2012. *Handbook of Energy-Aware and Green Computing*. Chapman & Hall/CRC.

[2] Mikael Asberg, Thomas Nolte, and Shinpei Kato. 2012. ExSched: An External CPU Scheduler Framework for Real-Time Systems. In *Proceedings of the 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications.*

[3] Hakan Aydin, V. Devadas, and D. Zhu. 2006. System-level Energy Management for Periodic Real-Time Tasks. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS'06), Rio de Janeiro, Brazil.*

[4] Hakan Aydin, Rami Melhem, Daniel Mossé, and Pedro Mejía-Alvarez. 2004. Power-Aware Scheduling for Periodic Real-Time Tasks. 53, 5 (2004), 584–600. https://doi.org/10.1109/TC.2004.1275298

[5] Mario Bambagini, Mauro Marinoni, Hakan Aydin, and Giorgio Buttazzo. 2016. Energy-Aware Scheduling for Real-Time Systems: A Survey. 15, 1 (2016). https://doi.org/10.1145/2808231

[6] Mark Benson. 2014. *The Art of Software Thermal Management for Embedded Systems.* Springer-Verlag New York.

[7] Aaron Block and William Kelley. 2015. Implementing Adaptive Clustered Scheduling in LITMUS-RT. In *Proceedings of the 11th Annual Workshop on Operating Systems Platforms for Embedded Real-Time applications.*

[8] Vincenzo Bonifaci, Björn Brandenburg, Gianlorenzo D'Angelo, and Alberto Marchetti-Spaccamela. 2016. Multiprocessor Real-Time Scheduling with Hierarchical Processor Affinities. In *Proceedings of the 28th Euromicro Conference on Real-Time Systems.*

[9] B. Brandenburg. 2011. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems.* Ph.D. Dissertation.

[10] Björn B. Brandenburg and Mahircan Gül. 2016. Global Scheduling Not Required: Simple, Near-Optimal Multiprocessor Real-Time Scheduling with Semi-Partitioned Reservations. In *Proceedings of the 37th IEEE Real-Time Systems Symposium.*

[11] Neil Brown. [n. d.]. Improvements in CPU frequency management. https://lwn.net/Articles/682391

[12] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. 2006. LITMUS-RT: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium.*

[13] Felipe Cerqueira, Manohar Vanga, and Björn B. Brandenburg. 2014. Scaling Global Scheduling with Message Passing. In *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Applications Symposium.*

[14] Jina-Jia Chen, Heng-Ruey Hsu, and Tei-Wei Kuo. 2006. Leakage-Aware Energy-Efficient Scheduling of Real-Time Tasks in Multiprocessor Systems. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium.*

[15] Jonathan Corbet. [n. d.]. The cpuidle subsystem. https://lwn.net/Articles/384146

[16] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms.* The MIT Press.

[17] National Instruments Corporation. [n. d.]. Current Measurements: How-To Guide. http://www.ni.com/tutorial/7114/en

[18] Matthew Dellinger, Piyush Garyali, and Binoy Ravindran. 2011. ChronOS Linux: A Best-Effort Real-Time Multiprocessor Linux Kernel. In *Proceedings of the 48th ACM/EDAC/IEEE Design and Automation Conference.*

[19] Vinay Devadas and Hakan Aydin. 2010. Coordinated Power Management of Periodic Real-Time Tasks on Chip Multiprocessors. In *Proceedings of the International Green Computing Conference.*

[20] Sudarshan K. Dhall and C. L. Liu. 1978. On a Real-Time Scheduling Problem. 26, 1 (1978), 127–140. https://doi.org/10.1287/opre.26.1.127

[21] Linux Kernel Documentation. [n. d.]. CPU frequency and voltage scaling code in the Linux(TM) kernel. https://www.kernel.org/doc/Documentation/cpu-freq/core.txt

[22] Glenn A. Elliot and James H. Anderson. 2011. Globally Scheduled Real-Time Multiprocessor Systems with GPUs. 48, 1 (2011), 34–74. https://doi.org/10.1007/s11241-011-9140-y

[23] Glenn A. Elliot, Bryan C. Ward, and James H. Anderson. 2013. GPUSync: A Framework for Real-Time GPU Management. In *Proceedings of the 34th IEEE Real-Time Systems Symposium.*

[24] Paul Emberson, Roger Stafford, and Robert I. Davis. 2010. Techniques For The Synthesis Of Multiprocessor Tasksets. In *Proceedings of the 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems.*

[25] Unified Extensible Firmware Interface Forum. [n. d.]. ACPI Specification. http://uefi.org/specifications

[26] Joël Goossens, Shelby Funk, and Sanjoy Baruah. 2003. Priority-Driven Scheduling of Periodic Task Systems on Multiprocessors. 25, 2-3 (2003), 187–205. https://doi.org/10.1023/A:1025120124771

[27] Jonathan L. Herman, Christopher J. Kenna, Malcolm S. Mollison, James H. Anderson, and Daniel M. Johnson. 2012. RTOS Support for Multicore Mixed-Criticality Systems. In *Proceedings of the 18th IEEE Real-Time and Embedded Technology and Applications Symposium.*

[28] Chong-Min Kyung and Sungjoo Yoo. 2011. *Energy-Aware System Design.* Springer Netherlands.

[29] Insup Lee, Joseph Y-T Leung, and Sang H. Son. 2008. *Handbook of Real-Time and Embedded Systems.* Chapman and Hall/CRC.

[30] J. M. Lopez, M. Garcia, J. L. Diaz, and D. F. Garcia. 2000. Worst-Case Utilization Bound for EDF Scheduling on Real-Time Multiprocessor Systems. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems.*

[31] Robert Love. 2010. *Linux Kernel Development.* Pearson Education.

[32] Benini Luca, Alessandro Bogliolo, and Giovanni De Micheli. 2000. A survey of design techniques for system-level dynamic power management. 8, 3 (2000), 299–316.

[33] Abdelhafid Mazouz, Alexandre Laurent, Benoît Pradelle, and Willian Jalby. 2014. Evaluation of CPU Frequency Transition Latency. 29, 3-4 (2014), 187–195. https://doi.org/10.1007/s00450-013-0240-x

[34] Patrick Mochel. 2003. Linux Kernel Power Management. In *Proceedings of the Linux Symposium*.

[35] Gabriel A. Moreno and Dionisio de Niz. 2012. An Optimal Real-Time Voltage and Frequency Scaling for Uniform Multiprocessors. In *Proceedings of the 18th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*.

[36] Vincent Nélis. 2011. *Energy-Aware Real-Time Scheduling in Embedded Multiprocessor Systems*. Ph.D. Dissertation.

[37] Santiago Pagani and Jian-Jia Chen. 2013. Energy Efficiency Analysis for the Single Frequency Approximation (SDA) Scheme. In *Proceedings of the 19th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*.

[38] Venkatesh Pallipadi and Adam Belay. 2007. cpuidle-Do nothing, efficiently. In *Proceedings of the Linux Symposium*.

[39] David A. Patterson and John L. Hennessy. 2012. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann.

[40] Padmanabhan Pillai and Kang G. Shin. 2001. Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems. In *Proceedings of the 8th ACM Symposium on Operating System Principles*.

[41] K. R. Rao, D. N. Kim, and J. J. Hwang. 2010. *Fast Fourier Transform: Algorithms And Applications*. Springer Netherlands.

[42] Irani Sandy, Sandeep Shukla, and Rajesh Gupta. 2003. Online strategies for dynamic power management in systems with multiple power-saving states. 2, 3 (2003), 325–346.

[43] Youngsoo Shin and Kiyoung Choi. 1999. Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*.

[44] Keysight Technologies. [n. d.]. 34411A Digital Multimeter, 6 1/2 Digit Overview and Features. https://www.keysight.com/en/pd-692679-pn-34411A

[45] Jean Vuillemin. 1978. A Data Structure for Manipulating Priority Queues. 21, 4 (1978), 309–315. https://doi.org/10.1145/359460.359478

[46] Chuan-Yue Yang, Jian-Jia Chen, and Tei-Wei Kuo. 2005. An Approximation Algorithm for Energy-Efficient Scheduling on A Chip Multiprocessor. In *Proceedings of the Conference on Design and Automation and Test in Europe*.

[47] Ren. Zhiyuan, Bruce H. Krogh, and Radu Marculescu. 2005. Hierarchical adaptive dynamic power management. 54, 4 (2005), 409–420.

[48] Dakai Zhu, Rami Melhem, and Bruce Childers. 2003. Scheduling with Dynamic Voltage/Speed Adjustment Using Slack Reclamation in Multi-Processor Real-Time Systems. 14, 7 (2003), 686–700. https://doi.org/10.1109/TPDS.2003.1214320

Pedro Mejia-Alvarez received the B.S. degree in computer systems from ITESM, Queretaro, Mexico, in 1985, and the Ph.D. degree in informatics from the Polytechnic University of Madrid, Spain, in 1995. He has been Professor for the computer science department at Cinvestav-IPN, since 1997. His main research interests are mobile computing, real-time systems scheduling, adaptive fault tolerance, and software engineering.

David Moncada-Madero received the MsC. degree in computer science from CINVESTAV-Guadalajara. He is currently a Software Engineer at Microsoft in Redmond Wa, USA. His research interests include real-time systems, mobile and wearable computing.

Hakan Aydin received the Ph.D. degree in computer science from the University of Pittsburgh in 2001. He is currently an associate professor in the Computer Science Department at George Mason University. He was a recipient of the US National Science Foundation (NSF) Faculty Early Career Development (CAREER) Award in 2006. His research interests include real-time systems, low-power computing, and fault tolerance. He is a member of the IEEE.

Arnoldo Díaz-Ramirez is a research professor in the department of Computer Systems at Tecnologico Nacional de Mexico/Instituto Tecnologico de Mexicali. He received the BS degree in computer sciences from Cetys University, Mexicali, Mexico, and the Masters degree in computer sciences from the same university. He received the PhD degree in computer sciences from Universitat Politecnica de Valencia, Spain, in 2006. His research interests include real-time systems, Internet of Things, wireless sensor networks, and ubiquiotous computing. He is member of the IEEE Computer Society.