Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 582648, 16 pages
doi:10.1155/2008/582648

Research Article

Hard Real-Time Performances in Multiprocessor-Embedded Systems Using ASMP-Linux

Emiliano Betti,[1] Daniel Pierre Bovet,[1] Marco Cesati,[1] and Roberto Gioiosa[1,2]

[1] *System Programming Research Group, Department of Computer Science, Systems, and Production, University of Rome "Tor Vergata", Via del Politecnico 1, 00133 Rome, Italy*
[2] *Computer Architecture Group, Computer Science Division, Barcelona Supercomputing Center (BSC), c/ Jordi Girona 31, 08034 Barcelona, Spain*

Correspondence should be addressed to Roberto Gioiosa, gioiosa@sprg.uniroma2.it

Received 30 March 2007; Accepted 15 August 2007

Recommended by Ismael Ripoll

Multiprocessor systems, especially those based on multicore or multithreaded processors, and new operating system architectures can satisfy the ever increasing computational requirements of embedded systems. ASMP-LINUX is a modified, high responsiveness, open-source hard real-time operating system for multiprocessor systems capable of providing high real-time performance while maintaining the code simple and not impacting on the performances of the rest of the system. Moreover, ASMP-LINUX does not require code changing or application recompiling/relinking. In order to assess the performances of ASMP-LINUX, benchmarks have been performed on several hardware platforms and configurations.

Copyright © 2008 Emiliano Betti et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

1. INTRODUCTION

This article describes a modified Linux kernel called ASMP-Linux, *ASymmetric MultiProcessor Linux*, that can be used in embedded systems with hard real-time requirements. ASMP-Linux provides real-time capabilities while maintaining the software architecture relatively simple. In a conventional (symmetric) kernel, I/O devices and CPUs are considered alike, since no assumption is made on the system's load. Asymmetric kernels, instead, consider real-time processes and related devices as privileged and shield them from other system activities.

The main advantages offered by ASMP-Linux to real-time applications are as follows.

(i) Deterministic execution time (upto a few hundreds of nanoseconds).
(ii) Very low system overhead.
(iii) High performance and high responsiveness.

Clearly, all of the good things offered by ASMP-Linux have a cost, namely, at least one processor core dedicated to real-time tasks. The current trend in processor design is leading to chips with more and more cores. Because the power consumption of single-chip multicore processors is not significantly higher than that of single-core processors, we can expect that in a near future many embedded systems will make use of multicore processors. Thus, we foresee that, even in embedded systems, the hardware requirements of real-time operating systems such as ASMP-Linux will become quite acceptable in a near future.

The rest of this article is organized as follows: Section 2 illustrates briefly the main characteristics of asymmetric kernels; Section 3 describes the evolution of single-chip multicore processors; Section 4 illustrates the main requirements of hard real-time systems; Section 5 gives some details about how ASMP-Linux is implemented; Section 6 lists the tests performed on different computers and the results obtained; and finally, Section 7 suggests future work to be performed in the area of asymmetric kernels.

2. ASYMMETRIC MULTIPROCESSOR KERNELS

The idea of dedicating some processors of a multiprocessor system to real-time tasks is not new. In an early description of the ARTiS system included in [1], processors are classified as realtime and nonreal-time. Real-time CPUs execute nonpreemptible code only, thus tasks running on these

processors perform predictably. If a task wants to enter into a preemptible section of code on a real-time processor, ARTiS will automatically migrate this task to a nonreal-time processor.

Furthermore, dedicated load-balancing strategies allow all CPUs to be fully exploited. In a more recent article by the same group [2], processes have been divided into three groups: highest priority (RT0), other real-time Linux tasks (RT1+), and nonreal-time tasks; furthermore, a basic library has been implemented to provide functions that allow programmers to register and unregister RT0 tasks. Since ARTiS relies on the standard Linux interrupt handler, the system latency may vary considerably: a maximum observed latency of 120 microseconds on a 4-way Intel architecture-64 (IA-64) heavily loaded system has been reported in [2].

A more drastic approach to reduce the fluctuations in the latency time has been proposed independently in [3, 4]. In this approach, the source of real-time events is typically a hardware device that drives an IRQ signal not shared with other devices. The ASMP system is implemented by binding the real-time IRQ and the real-time tasks to one or more "shielded" processors, which never handle nonreal-time IRQs or tasks. Of course, the nonreal-time IRQs and nonreal-time tasks are handled by the other processors in the system. As discussed in [3, 4], the fluctuations of the system latency are thus significantly reduced.

It is worth noting that, since version 2.6.9 relased in October 2004, the standard Linux kernel includes a boot parameter *(isolcpus)* that allows the system administrator to specify a list of "isolated" processors: they will never be considered by the scheduling algorithm, thus they do not normally run any task besides the per-CPU kernel threads. In order to force a process to migrate on an isolated CPU, a programmer may make use of the Linux-specific system call *sched_setaffinity()*. The Linux kernel also includes a mechanism to bind a specific IRQ to one or more CPUs; therefore, it is easy to implement an ASMP mechanism using a standard Linux kernel.

However, the implementation of ASMP discussed in this article, ASMP-Linux, is not based on the *isolcpus* boot parameter. A clear advantage of ASMP-Linux is that the system administrator can switch between SMP and ASMP mode at run time, without rebooting the computer. Moreover, as explained in Section 5.2, ASMP-Linux takes care of avoiding load rebalancing for asymmetric processors, thus it should be slightly more efficient than a system based only on *isolcpus*.

Although we will concentrate in this article on the real-time applications of asymmetric kernels, it is worth mentioning that these kernels are also used in other areas. As an example, some multicore chips introduced recently include different types of cores and require thus an asymmetric kernel to handle each core properly. The IBM cell broadband engine (BE) discussed in [5], for instance, integrates a 64-bit PowerPC processor core along with eight "synergistic processor cores." This multicore chip is the heart of the Sony PS3 playstation console, although other applications outside of the video game console market, such as medical imaging and rendering graphical data, are being considered.

3. MULTIPROCESSOR-EMBEDDED SYSTEMS

The increasing demand for computational power is leading embedded system developers to use general-purpose processors, such as ARM, Intel, AMD, or IBM's POWER, instead of microcontrollers or digital signal processors (DSPs).

Furthermore, many hardware vendors started to develop and market *system-on-chip* (SoC) devices, which usually include on the same integrated circuit one or more general-purpose CPUs, together with other specialized processors like DSPs, peripherals, communication buses, and memory. System-on-chip devices are particularly suited for embedded systems because they are cheaper, more reliable, and consume less power than their equivalent multichip systems. Actually, power consumption is considered as the most important constraint in embedded systems [6].

In the quest for the highest CPU performances, hardware developers are faced with a difficult dilemma. On one hand, the Moore law does not apply to computational power any more, that is, computational power is no longer doubling every 18 months as in the past. On the other hand, power consumption continues to increase more than linearly with the number of transistors included in a chip, and the Moore law still holds for the number of transistors in a chip.

Several technology solutions have been adopted to solve this dilemma. Some of them try to reduce the power consumption by sacrificing computational power, usually by means of *frequency scaling*, *voltage throttling*, or both. For instance, the Intel Centrino processor [7] may have a variable CPU clock rate ranging between 600 MHz and 1.5 GHz, which can be dynamically adjusted according to the computational needs.

Other solutions try to get more computational power from the CPU without increasing power consumption. For instance, a key idea was to increase the *instruction level parallelism* (ILP) inside a processor; this solution worked well for some years, but nowadays the penalty of a cache miss (which may stall the pipeline) or of a miss-predicted branch (which may invalidate the pipeline) has become way too expensive.

Chip-multithread (CMT)[8] processors aim to solve the problem from another point of view: they run different processes at the same time, assigning them resources dynamically according to the available resources and requirements. Historically the first CMT processor was a *coarse-grained multithreading* CPU (IBM RS64-II [9, 10]) introduced in 1998: in this kind of processor only one thread executes at any instance. Whenever that thread experiments a long-latency delay (such as a cache miss), the processor swaps out the waiting thread and starts to execute the second thread. In this way the machine is not idle during the memory transfers and, thus, its utilization increases.

Fine-grained multithreading processors improve the previous approach: in this case the processor executes the two threads in successive cycles, most of the time in a round-robin fashion. In this way, the two threads are executed at the same time but, if one of them encounters a long-latency event, its cycles are lost. Moreover, this approach requires more hardware resources duplication than the coarse-grained multithreading solution.

In *simultaneous multithreading* (SMT) processors two threads are executed at the same time, like in the fine-grained multithreading CPUs; however, the processor is capable of adjusting the rate at which it fetches instructions from one thread flow or the other one dynamically, according to the actual environmental situation. In this way, if a thread experiments a long-latency event, its cycles will be used by the other thread, hopefully without loosing anything.

Yet another approach consists of putting more processors on a chip rather than packing into a chip a single CPU with a higher frequency. This technique is called *chip-level multiprocessing* (CMP), but it is also known as "chip multiprocessor;" essentially it implements symmetric multiprocessing (SMP) inside a single VLSI integrated circuit. Multiple processor cores typically share a common second- or third-level cache and interconnections.

In 2001, IBM introduced the first chip containing two single-threaded processors (cores): the POWER4 [11]. Since that time, several other vendors have also introduced their multicore solutions: dual-core processors are nowadays widely used (e.g., Intel Pentium D [12], AMD Opteron [13], and Sun UltraSPARC IV [14] have been introduced in 2005); quad-core processors are starting to appear on the shelves (Intel Pentium D [15] was introduced in 2006 and AMD Barcelona will appear in late 2007); eight-core processors are expected in 2008.

In conclusion, we agree with McKenney's forecast [16] that in a near future many embedded systems will sport several CMP and/or CMT processors. In fact, the small increase in power consumption will likely be justified by the large increment of computational power available to the embedded system's applications. Furthermore, the actual trend in the design of system-on-chip devices suggests that in a near future such chips will include multicore processors. Therefore, the embedded system designers will be able to create boards having many processors almost "for free," that is, without the overhead of a much more complicated electronic layout or a much higher power consumption.

4. SATISFYING HARD REAL-TIME CONSTRAINTS USING LINUX

The term *real-time* pertains to computer applications whose correctness depends not only on whether the results are the correct ones, but also on the time at which the results are delivered. A *real-time system* is a computer system that is able to run real-time applications and fulfill their requirements.

4.1. Hard and soft real-time applications

Real-time systems and applications can be classified in several ways. One classification divides them in two classes: "hard" real-time and "soft" real-time.

A *hard real-time* system is characterized by the fact that meeting the applications' deadlines is the primary metric of success. In other words, failing to meet the applications' deadlines—timing requirements, quality of service, latency contraints, and so on—is a catastrophic event that must be absolutely avoided.

Conversely, a *soft real-time* system is suitable to run applications whose deadlines must be satisfied "most of the times," that is, the job carried on by a soft real-time application retains some value even if a deadline is passed. In soft real-time systems some design goals—such as achieving high average throughput—may be as important, or more important, than meeting application deadlines.

An example of hard real-time application is a missile defense system: whenever a radar detects an attacking missile, the real-time system has to compute all the information required for tracking, intercepting, and destroying the target missile. If it fails, catastrophic events might follow. A very common example of soft real-time application is a video stream decoder: the incoming data have to be decoded on the fly. If, for some reason, the decoder is not able to translate the stream before its deadline and a frame is lost, nothing catastrophic happens: the user will likely not even take notice of the missing frame (the human eye cannot distinguish images faster than 1/10 second).

Needless to say, hard real-time applications put much more time and resource constraints than soft real-time applications, thus they are critical to handle.

4.2. Periodic and event-driven real-time applications

Another classification considers two types of real-time applications: "periodic" and "event-driven."

As the name suggests, *periodic* applications execute a task periodically, and have to complete their job within a predefined deadline in each period. A nuclear power plant monitor is an example of a periodic hard real-time application, while a multimedia decoder is an example of a periodic soft real-time application.

Conversely, *event-driven* applications give raise to processes that spend most of the time waiting for some event. When an expected even occurs, the real-time process waiting for that event must wake up and handle it so as to satisfy the predefined time constraints. An example of event-driven hard real-time application is the missile defense system already mentioned, while a network router might be an example of an event-driven soft real-time application.

When dealing with periodic real-time applications, the operating system must guarantee a sufficient amount of resources—processors, memory, network bandwith, and so on—to each application so that it succeeds in meeting its deadlines. Essentially, in order to effectively implement a real-time system for periodic applications, a resource allocation problem must be solved. Clearly, the operating system will assign resources to processes according to their priorities, so that a high-priority task will have more resources than a low-priority task.

This article focuses on both event-driven and periodic hard real-time applications. Even if the former ones are supposed to be the most critical tasks to handle, in order to estimate the operating system overhead, some results for periodic real-time application are also provided in Section 6.

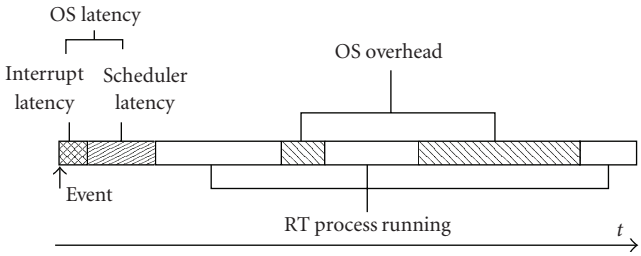


Figure 1: Jitter.

4.3. Jitter

In an ideal world, a real-time process would run undisturbed from the beginning to the end, directly communicating with the monitored device. In this situation a real-time process will require always the same amount of time (T) to complete its (periodic) task and the system is said to be *deterministic*.

In the real world, however, there are several software layers between the device and the real-time process, and other processes and services could be running at the same time. Moreover, other devices might require attention and interrupt the real-time execution. As a result, the amount of time required by the real-time process to complete its task is actually $T_x = T + \varepsilon$, where $\varepsilon \geq 0$ is a delay caused by the system. The variation of the values for ε is defined as the system's *jitter*, a measure of the non-determinism of a system.

Determinism is a key factor for hard real-time systems: the larger is the jitter, the less deterministic is the system's response. Thus, the jitter is also an indicator of the hard real-time capabilities of a system: if the jitter is greater than a critical threshold, the system cannot be considered as real-time. As a consequence, a real-time operating system must be designed so as to minimize the jitter observed by the real-time applications.

Jitter, by its nature, is not constant and makes the system behavior unpredictable; for this reason, real-time application developers must provide an estimated *worst-case execution time* (WCET), which is an upper bound (often quite pessimistic) of the real-time application execution time. A real-time application catches its deadline if $T_x \leq$ WCET.

As discussed in [17–19], short, unpredictable activities such as interrupts handling are the main causes of large jitter in computer systems. As shown in Figure 1, the jitter is composed by two main components: the "operating system overhead" and the "operating system latency."

The *operating system overhead* is the amount of time the CPU spends while executing system's code—for example, handling the hardware devices or managing memory—and code of other processes instead of the real-time process' code.

The *operating system latency* is the time elapsed between the instant in which an event is raised by some hardware device and the instant in which a real-time application starts handling that event.[1]

The definitions of overhead and latency are rather informal, because they overlap on some corner cases. For instance, the operating system overhead includes the time spent by the kernel in order to select the "best" process to run on each CPU; the component in charge of this activity is called *scheduler*. However, in a specific case, the time spent while executing the scheduler is not accounted as operating system overhead but rather as operating system latency: it happens when the scheduler is going to select precisely the process that carries on the execution of the real-time application. On the other hand, if some nonreal-time interrupts occur between a real-time event and the wakeup of the real-time applications, the time spent by the kernel while handling the nonreal-time interrupts shoud be accounted as overhead rather than latency.

As illustrated in Figure 1, operating system latency can be decomposed in two components: the "interrupt latency" and the "scheduler latency." Both of them reduce the system's determinism and, thus, its real-time capabilities.

The *interrupt latency* is the time required to execute the interrupt handler connected to the device that raised the interrupt, that is, the device that detected an event the real-time process has to handle. The *scheduler latency* is the time required by the operating system scheduler to select the real-time task as the next process to run and assign it the CPU.

4.4. Hardware jitter

There is a lower bound on the amount of nondeterminism in any embedded system that makes use of general-purpose processors. Modern *custom of-the-shelf* (COTS) processors are intrinsically parallel and so complex that it is not possible to predict when an instruction will complete. For example, cache misses, branch predictors, pipeline, out-of-order execution, and speculative accesses could significantly alter the execution time of an in-flying instruction. Moreover, some shared resources, such as the PCI bus or the memory controller, require an arbitration among the hardware devices, that is, a lock mechanism. There are also "intrinsic indeterministic buses" used to connect devices to the system or system to system, such as Ethernet or PCI buses [20].

Nonetheless, most of the real-time embedded systems that require high computational power make use of COTS processors—mainly for their high performance/cost ratio—implicitly giving strict determinism up. As a matter of fact, commercial and industrial real-time systems often follow the *five-nines rule*: the system is considered hard real-time if a real-time application catches its deadline the 99.999% of the times.

The indeterminism caused by the hardware cannot be reduced by the software, thus no real-time operating system (including ASMP-Linux) can have better performances than those of the underlying hardware. In other words, the execution time T_x of a real-time task will always be affected by

[1] Periodic real-time applications, too, suffer from operating system latency. For example, the operating system latency may cause a periodic application to wake up with some delay with respect to the beginning of its real-time period.

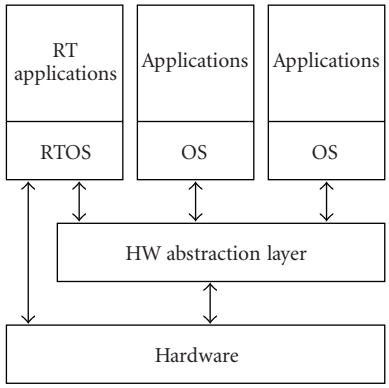


Figure 2: Horizontally partitioned operating system.

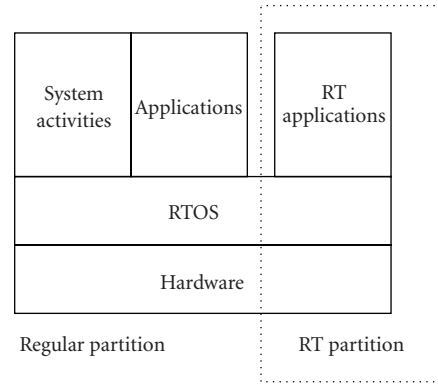


Figure 3: Vertically partitioned operating system.

some jitter, no matter of how good the real-time operating system is. However, ASMP-Linux aims to contribute to this jitter as less as possible.

4.5. System partitioning

The real-time operating system must guarantee that, when a real-time application requires some resource, that resource is made available to the application as soon as possible. The operating system should also ensure that the resources shared among all processes—the CPU itself, the memory bus, and so on—are assigned to the processes according to a policy that takes in consideration the priorities among the running processes.

As long as the operating system has to deal only with processes, it is relatively easy to preempt a running process and assign the CPU to another, higher-priority process. Unfortunately, external events and operating system critical activities, required for the correct operation of the whole system, occur at unpredictable times and are usually associated with the highest priority in the system. Thus, for example, an external interrupt could delay a critical, hard real-time application that, deprived of the processor, could eventually miss its deadline. Even if the application manages to catch its deadline, the operating system may introduce a factor of nondeterminism that is tough to predict in advance.

Therefore, handling both external events and operating system critical activities while guaranteeing strict deadlines is the main problem in real-time operating systems. Multiprocessor systems make this problem even worse, because operating system activities are much more complicated.

In order to cope with this problem, real-time operating systems are usually partitioned horizontally or vertically. As illustrated in Figure 2, *horizontally partitioned operating systems* have a bottom layer (called *hardware abstraction layer*, or HAL) that virtualizes the real hardware; on top of this layer there are several *virtual machines*, or *partitions*, running a standard or modified operating system, one for each application's domain; finally, applications run into their own domain as they were running on a dedicated machine.

In horizontally partitioned operating systems the real-time application have an abstract view of the system; external events are caught by the hardware abstraction layer and propagated to the domains according to their priorities. While it seems counterintuitive to use virtual machines for hard real-time applications, this approach works well in most of the cases, even if the hardware abstraction layer—in particular the partitions scheduler or the interrupt dispatcher—might introduce some overhead. Several Linux-based real-time operating systems such as RTAI [21] (implemented on top of *Adeos* [22]) and some commercial operating systems like Wind River's VxWorks [23] use this software architecture.

In contrast with the previous approach, in a *vertically partitioned operating system* the resources that are crucial for the execution of the real-time applications are directly assigned to the application themselves, with no software layer in the middle. The noncritical applications and the operating system activities not related to the real-time tasks are not allowed to use the reserved resources. This schema is illustrated in Figure 3.

Thanks to this approach, followed by ASMP-Linux, the real-time specific components of the operating system are kept simple, because they do not require complex partition schedulers or virtual interrupt dispatchers. Moreover, the performances of a real-time application are potentially higher with respect to those of the corresponding application in a horizontally partitioned operating system, because there is no overhead due to the hardware abstraction layer. Finally, in a vertically partitioned operating system, the nonreal-time components never slow down unreasonably, because these components always have their own hardware resources different from the resources used by the real-time tasks.

5. IMPLEMENTATION OF ASMP-LINUX

ASMP-Linux has been originally developed as a patch for the 2.4 Linux kernel series in 2002 [3]. After several revisions and major updates, ASMP-Linux is now implemented as a patch for the Linux kernel 2.6.19.1, the latest Linux kernel version available when this article has been written.

One of the design goals of ASMP-Linux is simplicity: because Linux developers introduce quite often significant changes in the kernel, it would be very difficult to maintain

the ASMP-Linux patch if it would be intrusive or overly complex. Actually, most of the code specific to ASMP-Linux is implemented as an independent kernel module, even if some minor changes in the core kernel code—mainly in the scheduler, as discussed in Section 5.2—are still required.

Another design goal of ASMP-Linux is architecture-independency: the patch can be easily ported to many different architectures, besides the IA-32 architecture that has been adopted for the experiments reported in Section 6.

It should be noted, however, that in a few cases ASMP-Linux needs to interact with the hardware devices (for instance, when dealing with the local timer, as explained in Section 5.3). In these cases, ASMP-Linux makes use of the interfaces provided by the standard Linux kernel; those interfaces are, of course, architecture-dependent but they are officially maintained by the kernel developers.

It is also worth noting that what ASMP-Linux can or cannot do depends ultimately on the characteristics of the underlying system architecture. For example, in the IBM's POWER5 architecture disabling the in-chip circuit that generates the local timer interrupt (the so-called *decrementer*) also disables all other external interrupts. Thus, the designer of a real-time embedded system must be aware that in some general-purpose architectures it might be simply impossible to mask all sources of system jitter.

ASMP-Linux is released under the version 2 of the GNU General Public License [24], and it is available at http://www.sprg.uniroma2.it/asmplinux.

5.1. System partitions

ASMP-Linux is a vertically partitioned operating system. Thus, as explained in Section 4.5, it implements two different kinds of partitions as follows.

System partition

It executes all the nonreal-time activities, such as daemons, normal processes, interrupt handling for noncritical devices, and so on.

Real-time partition

It handles some real-time tasks, as well as any hardware device and driver that is crucial for the real-time performances of that tasks.

In an ASMP-Linux system there is exactly one system partition, which may consist of several processors, devices, and processes; moreover, there should always exist at least one real-time partition (see Figure 4). Additional real-time partitions might also exist, each handling one specific real-time application.

Each real-time partition consists of a processor (called *shielded CPU*, or shortly s-cpu), $n_{\text{irq}} \geq 0$ IRQ lines assigned to that processor and corresponding to the critical hardware devices handled in the partition, and $n_{\text{task}} \geq 0$ real-time processes (there could be no real-time process in the partition; this happens when the whole real-time algorithm is coded inside an interrupt handler).

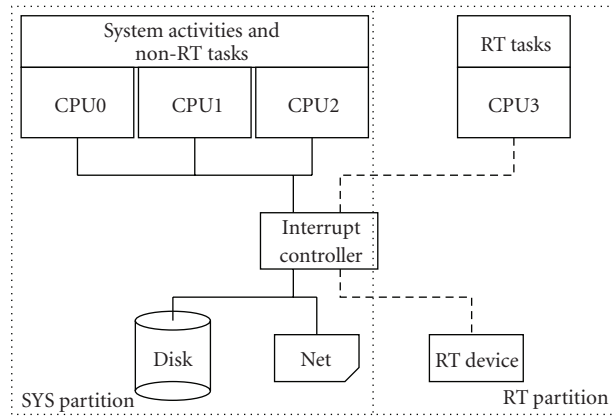


Figure 4: ASMP-Linux partitioning.

Each real-time partition is protected from any external event or activity that does not belong to the real-time task running on that partition. Thus, for example, no conventional process can be scheduled on a shielded CPU and no normal interrupt can be delivered to that processor.

5.2. Process handling

The bottom rule of ASMP-Linux while managing processes is as follows.

Every process assigned to a real-time partition must run only in that partition; furthermore, every process that does not belong to a real-time partition cannot run on that partition.

It should be noted, however, that a real-time partition always include a few peculiar nonreal-time processes. In fact, the Linux kernel design makes use of some processes, called *kernel threads*, which execute only in Kernel Mode and perform system-related activities. Besides the *idle process*, a few kernel threads such as *ksoftirqd* [25] are duplicated across all CPUs, so that each CPU executes one specific instance of the kernel thread. In the current design of ASMP-Linux, the per-CPU kernel threads still remain associated with the shielded CPUs, thus they can potentially compete with the real-time tasks inside the partition. As we will see in Section 6, this design choice has no significant impact on the operating system overhead and latency.

The ASMP-Linux patch is not intrusive because the standard Linux kernel already provides support to select which processes can execute on each CPU. In particular, every process descriptor contains a field named *cpus_allowed*, which is a bitmap denoting the CPUs that are allowed to execute the process itself. Thus, in order to implement the asymmetric behaviour, the bitmaps of the real-time processes are modified so that only the bit associated with the corresponding shielded CPU is set; conversely, the bitmaps of the nonreal-time processes are modified so that the bits of all shielded CPUs are cleared.

A real-time partition might include more than one real-time process. Scheduling among the real-time partition is still achieved by the standard Linux scheduler, so the

standard Linux static and dynamic priorities are honored. In this case, of course, it is up to the developer of the real-time application to ensure that the deadlines of each process are always catched.

The list of real-time processes assigned to a real-time partition may also be empty: this is intended for those applications that do not need to do anything more than handling the interrupts coming from some hardware device. In this case, the device handled in a real-time partition can be seen as a *smart device*, that is, a device with the computational power of a standard processor.

The ASMP-Linux patch modifies in a few places the scheduling algorithm of the standard Linux kernel. In particular, since version 2.6, Linux supports the so-called *scheduling domains* [25]: the processors are evenly partitioned in domains, which are kept balanced by the kernel according to the physical characteristics of the CPUs. Usually, the load in CMP and CMT processors will be equally spread on all the physical chips. For instance, in a system having two physical processors *chip0* and *chip1*, each of which being a 2-way CMT CPU, the standard Linux scheduler will try to put two running processes so as to assign one process to the first virtual processor of *chip0* and the other one to the first virtual processor of *chip1*. Having both processes running on the same chip, one on each virtual processor, would be a waste of resource: an entire physical chip kept idle.

However, load balancing among scheduling domains is a time-consuming, unpredictable activity. Moreover, it is obviously useless for shielded processors, because only predefined processes can run on each shielded CPU. Therefore, the ASMP-Linux patch changes the load balancing algorithm so that shielded CPUs are always ignored.

5.3. Interrupts handling

As mentioned in Section 4.3, interrupts are the major cause of jitter in real-time systems, because they are generated by hardware devices asynchronously with respect to the process currently executed on a CPU. In order to understand how ASMP-Linux manages this problem, a brief introduction on how interrupts are delivered to processors is required.

Most uniprocessor and multiprocessor systems include one or more *Interrupt Controller* chips, which are capable to route interrupt signals to the CPUs according to predefined routing policies. Two routing policies are commonly found: either the Interrupt Controller propagates the next interrupt signal to one specific CPU (using, e.g., a round-robin scheduling), or it propagates the signal to all the CPUs. In the latter case, the CPU that first stops the execution of its process and starts to execute the interrupt handler sends an acknowledgement signal to the Interrupt Controller, which frees the others CPUs from handling the interrupt. Figure 5 shows a typical configuration for a multiprocessor system based on the IA-32 architecture.

A shielded process must receive only interrupts coming from selected, crucial hardware devices, otherwise, the real-time application executing on the shielded processor will be affected by some unpredictable jitter. Fortunately, recent Interrupt Controller chips—such as the *I/O Advanced Programmable Interrupt Controller* (I/O-APIC) found in the Intel architectures—can be programmed so that interrupt signals coming from specific IRQ lines are forwarded to a set of predefined processors.

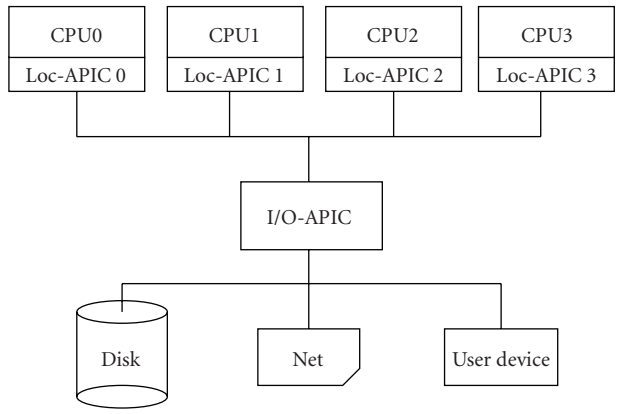


Figure 5: A SMP using Intel IO-APIC.

Thus, the ASMP-Linux patch instruments the Interrupt Controller chips to forward general interrupts only to non-shielded CPUs, while the interrupts assigned to a given real-time partition are sent only to the corresponding shielded CPU.

However, a shielded processor can also receive interrupt signals that do not come from an Interrupt Controller at all. In fact, modern processors include an internal interrupt controller—for instance, in the Intel processors this component is called *Local APIC*. This internal controller receives the signals coming from the external Interrupt Controllers and sends them to the CPU core, thus interrupting the current execution flow. However, the internal controller is also capable to directly exchange interrupt signals with the interrupt controllers of the other CPUs in the system; these interrupts are said *interprocessor interrupts*, or IPI. Finally, the internal controller could also generate a periodic self-interrupt, that is, a clock signal that will periodically interrupt the execution flow of its own CPU core. This interrupt signal is called *local timer*.

In multiprocessor systems based on Linux, interprocessor interrupts are commonly used to force all CPUs in the system to perform synchronization procedures or load balancing across CPUs while the local timer is used to implement the time-sharing policy of each processor. As discussed in the previous sections, in ASMP-Linux load balancing never affects shielded CPUs. Furthermore, it is possible to disable the local timer of a shielded CPU altogether. Of course, this means that the time-sharing policy across the processes running in the corresponding real-time partition is no longer enforced, thus the real-time tasks must implement some form of cooperative scheduling.

5.4. Real-time inheritance

During its execution, a process could invoke kernel services by means of system calls. The ASMP-Linux patch slightly

modifies the service routines of a few system calls, in particular, those related to process creation and removal: *fork()*, *clone()*, and *exit()*. In fact, those system calls affect some data structures introduced by ASMP-Linux and associated with the process descriptor.

As a design choice, a process transmits its property of being part of a real-time partition to its children; it also maintains that property even when executing an *exec()*-like system call. If the child does not actually need the real-time capabilities, it can move itself to the nonreal-time partition (see next section).

5.5. ASMP-linux interface

ASMP-Linux provides a simple */proc* file interface to control which CPUs are shielded, as well as the real-time processes and interrupts attached to each shielded CPU. The interface could have been designed as system calls but this choice would have made the ASMP-Linux patch less portable (system calls are universally numbered) and more intrusive.

Let us suppose that the system administrator wants to shield the second CPU of the system (CPU1), and that she wants to assign to the new real-time partition the process having PID X and the interrupt vector N. In order to do this, she can simply issue the following shell commands:

echo 1 > /proc/asmp/cpu1/shielded

echo X > /proc/asmp/cpu1/pids

echo N > /proc/asmp/cpu1/irqs.

The first command makes CPU1 shielded.[2] The other two commands assign to the shielded CPU the process and the interrupt vector, respectively. Of course, more processes or interrupt vectors can be added to the real-time partition by writing their identifiers into the proper *pids* and *irqs* files as needed.

To remove a process or interrupt vector it is sufficient to write the corresponding identifier into the proper */proc* file prefixed by the minus sign ("−"). Writing 0 into */proc/asmp/cpu1/shielded* file turns the real-time partition off: any process or interrupt in the partition is moved to the nonreal-time partition, then the CPU is unshielded.

The */proc* interface also allows the system administrator to control the local timers of the shielded CPUs. Disabling the local timer is as simple as typing:

echo 0 > /proc/asmp/cpu1/localtimer

The value written in the *localtimer* file can be either zero (timer disabled) or a positive scaling factor that represents how many ticks—that is, global timer interrupts generated by the Programmable Interval Timer chip—must elapse before the local timer interrupt is raised. For instance, writing the value ten into the *localtimer* file sets the frequency of the local timer interrupt to 1/10 of the frequency of the global timer interrupts.

Needless to say, these operations on the */proc* interface of ASMP-Linux can also be performed directly by the User Mode programs through the usual *open()* and *write()* system calls.

[2] Actually, the first command could be omitted in this case, because issuing either the second command or the third one will implicitly shield the target CPU.

Table 1: Characteristics of the test platforms.

ID	Architecture	CPUs	Freq. GHz	RAM GB
S1	IA-32 SMP HT	8 virt.	1.50	3
S2	IA-32 SMP	4 phys.	1.50	3
S3	IA-32 CMP	2 cores	1.83	1

6. EXPERIMENTAL DATA

ASMP-Linux provides a good foundation for an hard real-time operating system on multiprocessor systems. To validate this claim, we performed two sets of experiments.

The first test, described in Section 6.2, aims to evaluate the operating system overhead of ASMP-Linux: the execution time of a real-time process executing a CPU-bound computation is measured under both ASMP-Linux and a standard Linux 2.6.19.1 kernel, with different system loads, and on several hardware platforms.

The second test, described in Section 6.3, aims to evaluate the operating system latency of ASMP-Linux: the local timer is reprogrammed so as to raise an interrupt signal after a predefined interval of time; the interrupt handler wakes a sleeping real-time process up. The difference between the expected wake-up time and the actual wake-up time is a measure of the operating system latency. Again, the test has been carried on under both ASMP-Linux and a standard Linux 2.6.19.1 kernel, with different system loads, and on several hardware platforms.

6.1. Experimental environments

Two different platforms were used for the experiments; Table 1 summarizes their characteristics and configurations.

The first platform is a 4-way SMP Intel Xeon HT [26] system running at 1.50 GHz; every chip consists of two virtual processors (HT stands for *HyperThreading Technology* [27]). The operating system sees each virtual processor as a single CPU. This system has been tested with HT enabled (configuration "S1") and disabled (configuration "S2").

The second platform (configuration "S3") is a desktop computer based on a 2-way CMT Intel processor running at 1.83 Ghz. The physical processor chip contains two cores [28]. This particular version of the processor is the one used in laptop systems, optimized for power consumption.

The Intel Xeon HT processor is a coarse-grained multithreading processor; on the other side, the Intel Dual Core is a multicore processor (see Section 3). These two platforms cover the actual spectrum of modern CMP/CMT processors.

We think that both hyperthreaded processors and low-power versions of multicore processors are of particular interest to embedded system designers. In fact, one of the

biggest problems in embedded systems is heat dissipation. Hyperthreaded processors have relatively low power consumption, because the virtual processors in the chips are not full CPUs and the running threads are not really executed in parallel. Furthermore, low-power versions of COTS processors have often been used in embedded systems precisely because they make the heat dissipation problems much easier to solve.

For each platform, the following system loads have been considered.

- IDL The system is mostly *idle*: no User Mode process is runnable beside the real-time application being tested. This load has been included for comparison with the other system loads.
- CPU *CPU load*: the system is busy executing kp CPU-bound processes, where p is the number of (virtual) processors in the system, and k is equal to 16 for the first test, and to 128 for the second test.
- AIO *Asynchronous I/O load*: the system is busy executing kp I/O-bound processes, where k and p are defined as above. Each I/O-bound process continuously issues nonblocking write operations on disk.
- SIO *Synchronous I/O load*: the system is busy executing kp I/O-bound processes, where k and p are defined as above. Each I/O-bound process continuously issues synchronous (blocking) write operations on disk.
- MIX *Mixed load*: the system is busy executing $(k/2)p$ CPU-bound processes, $(k/2)p$ asynchronous I/O-bound processes, and $(k/2)p$ synchronous I/O-bound processes, where k and p are defined as above.

Each of these workloads has a peculiar impact on the operating system overhead. The CPU workload is characterized by a large number of processes that compete for the CPUs, thus the overhead due to the scheduler is significant. In the AIO workload, the write operations issued by the processes are asynchronous, but the kernel must serialize the low-level accesses to the disks in order to avoid data corruption. Therefore, the AIO workload is characterized by a moderate number of disk I/O interrupts and a significant overhead due to data moving between User Mode and Kernel Mode buffers. The SIO workload is characterized by processes that raise blocking write operations to disk: each process sleeps until the data have been written on the disk. This means that, most of the times, the processes are waiting for an external event and do not compete for the CPU. On the other hand, the kernel must spend a significant portion of time handling the disk I/O interrupts. Finally, in the MIX workload the kernel must handle many interrupts, it must move large amounts of data, and it must schedule many runnable processes.

For each platform, we performed a large number of iterations of the tests by using the following.

- N A normal *(SCHED_NORMAL)* Linux process (just for comparison).
- R_w A "real-time" *(SCHED_FIFO)* Linux process statically bound on a CPU that also gets all extenal interrupt signals of the system.
- R_b A "real-time" *(SCHED_FIFO)* Linux process statically bound on a CPU that does not receive any external interrupt signal.
- A_{on} A process running inside a real-time ASMP-Linux partition with local timer interrupts enabled.
- A_{off} A process running inside a real-time ASMP-Linux partition with local timer interrupts disabled.

The IA-32 architecture cannot reliably distribute the external interrupt signals across all CPUs in the system (this is the well-known "I/O APIC annoyance" problem). Therefore, two sets of experiments for real-time processes have been performed: R_w represents the worst possible case, where the CPU executing the real-time process handles all the external interrupt signals; R_b represents the best possible case, where the CPU executing the real-time process handles no interrupt signal at all (except the local timer interrupt). The actual performance of a production system is in some point between the two cases.

6.2. Evaluating the OS overhead

The goal of the first test is to evaluate the operating system overhead of ASMP-Linux. In order to achieve this, a simple, CPU-bound conventional program has been developed. The program includes a function performing n millions of integer arithmetic operations on a tight loop (n has been chosen, for each test platform, so that each iteration lasts for about 0.5 seondc); this function is executed 1000 times, and the execution time of each invocation is measured.

The program has been implemented in five versions (N, R_w, R_b, A_{on}, and A_{off}), and each program version has been executed on all platforms (S1, S2, and S3).

As discussed in Section 4.3, the data T_x coming from the experiments are the real-execution times resulting from the base time T effectively required for the computation plus any delay induced by the system. Generally speaking, $T_x = T + \varepsilon_h + \varepsilon_l + \varepsilon_o$, where ε_h is a nonconstant delay introduced by the hardware, ε_l is due to the operating system latency, and ε_o is due to the operating system overhead. The variations of the values $\varepsilon_h + \varepsilon_l + \varepsilon_o$ give raise to the jitter of the system. In order to understand how the operating system overhead ε_o affects the execution time, estimations for T, ε_h, and ε_l are required.

In order to evaluate T and ε_h, the "minimum" execution time required by the function—the *base time*—has been computed on each platform by running the program with interrupts disabled, that is, exactly as if the operating system were not present at all. The base time corresponds to $T + \varepsilon_h$; however, the hardware jitter for the performed experiments is negligible (roughly, some tens of nanoseconds, on the average) because the test has been written so that it makes little use of the caches and no use at all of memory, it does not execute long latency operation, and so on. Therefore, we can safely assume that $\varepsilon_h \approx 0$ and that the base time is essentially the value T. On the S1 and S2 platforms, the measured base time was 466.649 milliseconds, while on the S3 platform the measured base time was 545.469 milliseconds.

Finally, because the test program is CPU-bound and never blocks waiting for an interrupt signal, the impact of

Table 2: Operating system overheads for the MIX workload (in milliseconds).

(a) Configuration S1

Proc	Avg	StDev	Min	Max
N	20275.784	6072.575	12.796	34696.051
R_w	28.459	12.945	10.721	48.837
R_b	27.461	9.661	3.907	42.213
A_{on}	30.262	8.306	8.063	41.099
A_{off}	27.847	7.985	6.427	38.207

(b) Configuration S2

Proc	Avg	StDev	Min	Max
N	18513.615	5996.971	1.479	33993.351
R_w	4.215	0.226	3.913	10.146
R_b	1.420	0.029	1.393	1.554
A_{on}	1.490	0.044	1.362	1.624
A_{off}	0.000	0.000	0.000	0.000

(c) Configuration S3

Proc	Avg	StDev	Min	Max
N	20065.194	6095.807	0.606	32472.931
R_w	3.477	0.024	3.431	3.603
R_b	0.554	0.031	0.525	0.807
A_{on}	0.556	0.032	0.505	0.811
A_{off}	0.000	0.000	0.000	0.000

the operating system latency on the execution time is very small ($\varepsilon_l \approx 0$). One can thus assume that $T_x \approx T + \varepsilon_o$.

Therefore, in order to evaluate ε_o, the appropriate base times has been subtracted from the measured execution times. These differences are statistically summarized for the MIX workload in Table 2.[3]

Platform S1 shows how the asymmetric approach does not provide real-time performance for HyperThreading architectures. In fact, in those processors, the amount of shared resources is significant, therefore, a real-time application running on a virtual processor cannot be executed in a deterministic way regardless of the application running on the other virtual processors.

The test results for platform S2 and S3, instead, clearly state that ASMP-Linux does an excellent job in reducing the impact of operating system overhead on real-time applications.

Platform S3 is the most interesting to us because provides a good *performance/cost* ratio (where *cost* is intended in both money and power consumption senses). For lack of space, in the following analysis we will focus on that platform, unless other platforms are clearly stated.

Figure 8 shows how platform S3 performs with the different workloads and test cases R_w, R_b, A_{on}, and A_{off} (we do not show results from the N test case because its times are several orders of magnitude higher than the others). Each box in the figure represents the maximum overhead measured in all experiments performed on the specific workload and test case. Since the maximum overhead might be considered as a rough estimator for the real-time characteristics of the system, it can be inferred that all workloads present the same pattern: A_{off} is better than A_{on}, which in turn is roughly equivalent to R_b, which is finally much better than R_w. Since all workloads are alike, from now on we will specifically discuss the MIX workload—likely the most representative of a real-world scenario.

Figure 6 shows the samples measured on system S3 with MIX workload. Each dot represents a single test; its y-coordinate corresponds to the difference betweeen the measured time and the base value, as reported in Table 2. (Notice that the y-axis in the four plots have different scales).

The time required to complete each iteration of the test varies according to the operating system overhead experimented in that measurement: for example, in system S3, with a MIX workload, each difference can be between 3.431 milliseconds and 3.603 milliseconds for the R_w test case.

Figure 6(a) clearly shows that at the beginning of the test the kernel was involved in some activity, which terminated after about 300 samples. We identified this activity in creating the processes that belonged to the workload: after some time all the processes have been created and that activity is no longer present. Figure 6(a) also shows how, for the length of the experiment, all the samples are affected by jitter, thus they are far from the theoretical performance of the platform.

Figures 6(b) and 6(c) show that the operating system overhead mainly consists of some short, regular activities: we identify those activities with the local timer interrupt (which,

[3] Results for all workloads are reported in [29].

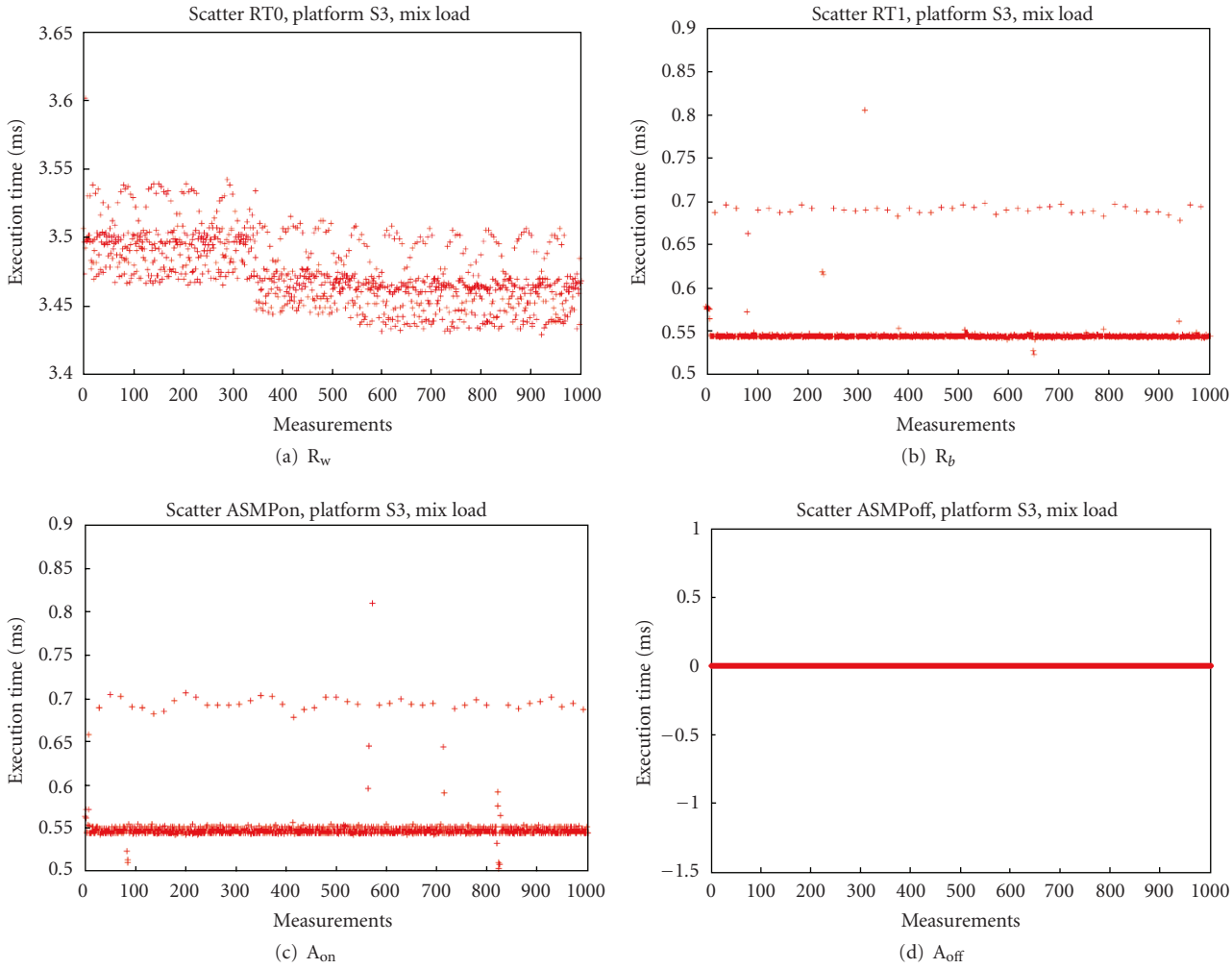


FIGURE 6: Scatter graphics for system S3, MIX workload.

in fact, is not present in Figure 6(d)). Every millisecond the local timer raised an interrupt (the highest priority kernel activity) and the CPU executed the interrupt handler instead of the real-time application. It can be noticed that R_b performs slightly better than A_{on}. As a matter of fact, the two test cases are very similar: in R_b the scheduler always selects the test program because it has *SCHED_FIFO* priority while in A_{on} the scheduler selects the "best" runnable process in the real-time ASMP partition—this normally means thatthe test program itself, but in a few cases, might also select a per-CPU kernel thread, whose execution makes the running time of the test program longer.

It is straightforward to see in Figure 6(d) how ASMP-Linux in the A_{off} version has a deterministic behavior, with no jitter, and catches the optimal performance that can be achieved on the platform (i.e., the *base time* mentioned above). On the other hand, using ASMP-Linux in the A_{on} version only provides soft real-time performance, comparable with those of R_b.

Figure 7 shows inverse cumulative densitive function (CDF) of the probability (x-axis) that a given sample is less than or equal to a threshold execution time (y-axis). For example, in Figure 7(a), the probability that the overhead in the test is less than or equal to 3.5 milliseconds is about 80%. We think this figure clearly explains how the operating system overhead can damage the performance of a real-time system. Different operating system activities introduce different amounts of jitter during the execution of the test, resulting in a nondeterministic response time. Moreover, the figure states how the maximum overhead can be significantly higher than the average operating system overhead. Once again, the figure shows how ASMP-Linux in the A_{on} version is only suitable for soft real-time application as well as R_b. On the other hand, A_{off} provides hard real-time performance.

6.3. *Evaluating the operating system latency*

The goal of the second test is to evaluate the operating system latency of ASMP-Linux . In order to achieve this, the local timer (see Section 5.3) has been programmed so as to emulate an external device that raises interrupts to be handled by a real-time application.

In particular, a simple program that sleeps until awakened by the operating system has been implemented in five

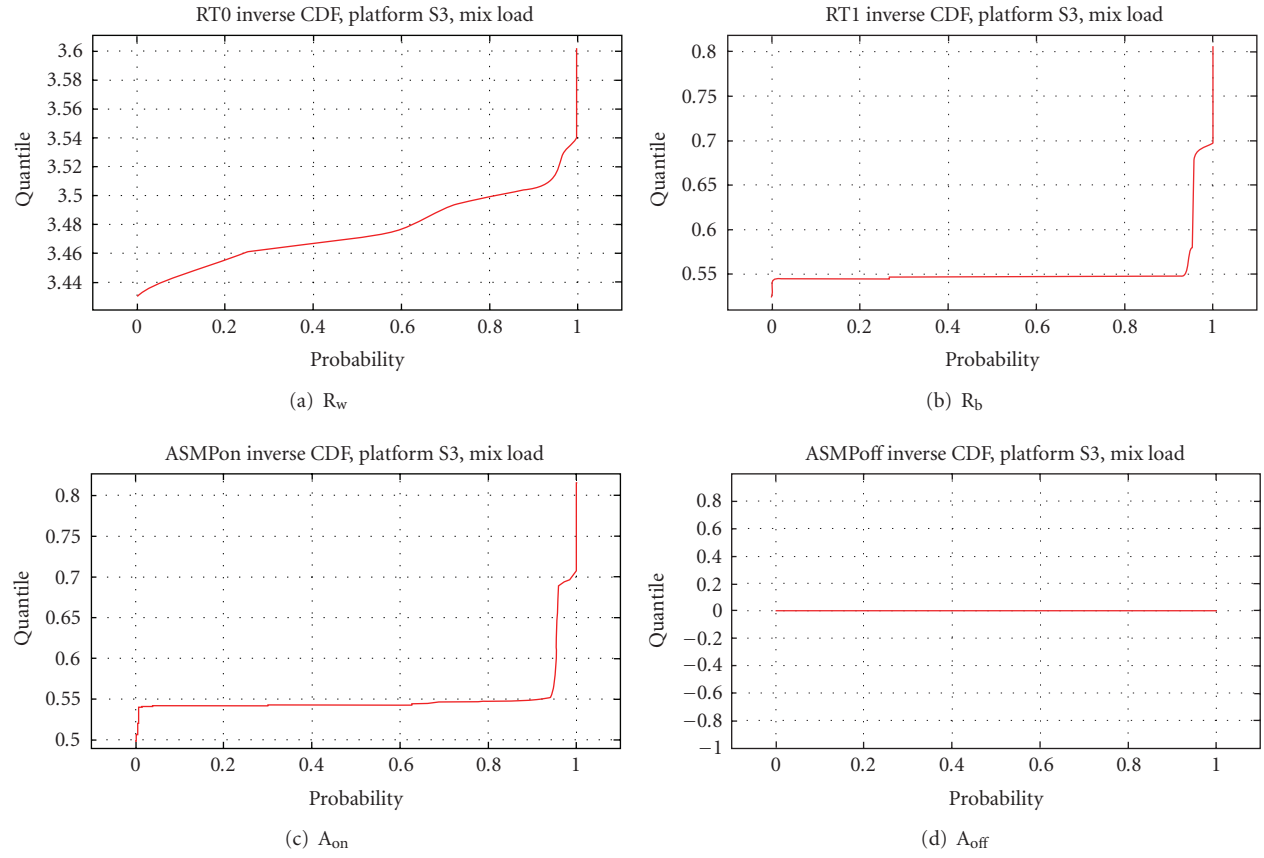


FIGURE 7: Inverse density functions for overhead on system S3, MIX workload.

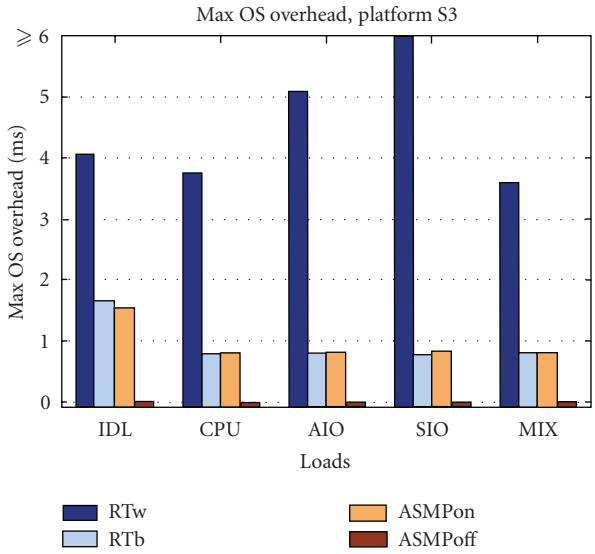


FIGURE 8: OS maximum overhead comparison.

versions (N, R_w, R_b, A_{on}, and A_{off}). Moreover, a kernel module has been developed in order to simulate a hardware device: it provides a device file that can be used by a User Mode program to get data as soon as they are available. The kernel module reprograms the local timer to raise a one-shot interrupt signal after a predefined time interval. The corresponding interrupt handler awakes the process blocked on the device file and returns to it a measure of the time elapsed since the timer interrupts occurred.

The data coming from the experiments yield the time elapsed since the local timer interrupt is raised and the User Mode program starts to execute again. Each test has been repeated 10 000 times; the results are statistically summarized for the MIX workload in Table 3.[4]

The delay observed by the real-time application is $\varepsilon_h + \varepsilon_l + \varepsilon_o$. Assuming as in the previous test $\varepsilon_h \approx 0$, the observed delay is essentially due to the operating system overhead and to the operating system latency. Except for the case "N," one can also assume that the operating system overhead is very small because, after being awoken, the real-time application does not do anything but issuing another read operation from the device file. This means that the probability of the real-time process being interrupted by any kernel activity in such a small amount of time is very small. In fact, the real-time application is either the only process that can run on the processor (A_{on} and A_{off}), or it has always greater priority than the other processes in the system (R_w and R_b). Thus, once awakened, the real-time task is selected right away by the kernel scheduler and no other process can interrupt it. Therefore, the delays shown in Table 3 are essentially due to

[4] Results for all workloads are reported in [29].

Table 3: Operating system latencies for the MIX workload (in microseconds).

(a) Configuration S1

Proc	Avg	StDev	Min	Max
N	13923.606	220157.013	6.946	$5.001 \cdot 10^6$
R_w	10.970	8.458	6.405	603.272
R_b	10.027	5.292	6.506	306.497
A_{on}	8.074	1.601	6.683	20.877
A_{off}	8.870	1.750	6.839	23.230

(b) Configuration S2

Proc	Avg	StDev	Min	Max
N	24402.723	331861.500	4.904	$4.997 \cdot 10^6$
R_w	5.996	1.249	4.960	39.982
R_b	5.511	1.231	4.603	109.964
A_{on}	5.120	0.275	4.917	9.370
A_{off}	5.441	0.199	5.207	6.716

(c) Configuration S3

Proc	Avg	StDev	Min	Max
N	182577.713	936480.576	1.554	$9.095 \cdot 10^6$
R_w	1.999	1.619	1.722	66.883
R_b	1.756	0.650	1.548	63.985
A_{on}	1.721	0.034	1.674	3.228
A_{off}	1.639	0.025	1.602	2.466

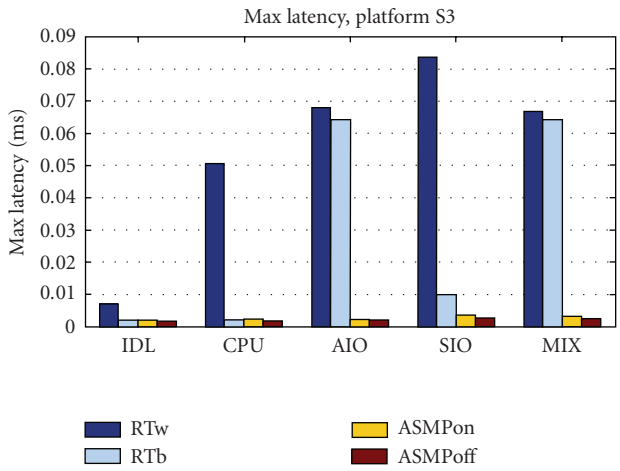


Figure 9: OS maximum latency comparison.

both the interrupt and the scheduler latency of the operating system.

Figure 9 shows how platform S3 performs with the different workloads and test cases R_w, R_b, A_{on}, and A_{off} (we do not show results from the N test case because its times are several orders of magnitude higher than the others). Each box in the figure represents the maximum latency measured in all experiments performed on the specific workload and test cases.

As we said, the probability that some kernel activity interrupts the real-time application is very small, yet not null. An interprocessor interrupt (IPI) could still be sent from one processor to the one running the real-time application (even for the R_b test) in order to force process load balancing. This is, likely, what happened to R_w and R_b, since they experiment a large, isolated maximum.

As in the previous test, from now on we will restrict ourselves in discussing the MIX workload, which we think is representative of all the workloads (see [29] for the complete data set).

Figure 10 shows the samples measured on system S3 with MIX workload. Each dot represents a single test; its y-coordinate corresponds to the latency time, as reported in Table 3. (The y-axis in the four plots have different scales; thus, e.g., the scattered points in Figure 10(d) would appear as a straight horizontal line on Figure 10(b)).

Figure 11 shows inverse cumulative densitive function (CDF) of the probability (x-axis) that a given sample is less than or equal to a threshold execution time (y-axis). For example, in Figure 11(d), the probability that the latency measured in the test is less than or equal to 1.6 microseconds is about 98%. In the A_{off} test case a small jitter is still present; nonetheless, it is so small that it could be arguably tolerated in many real-time scenarios.

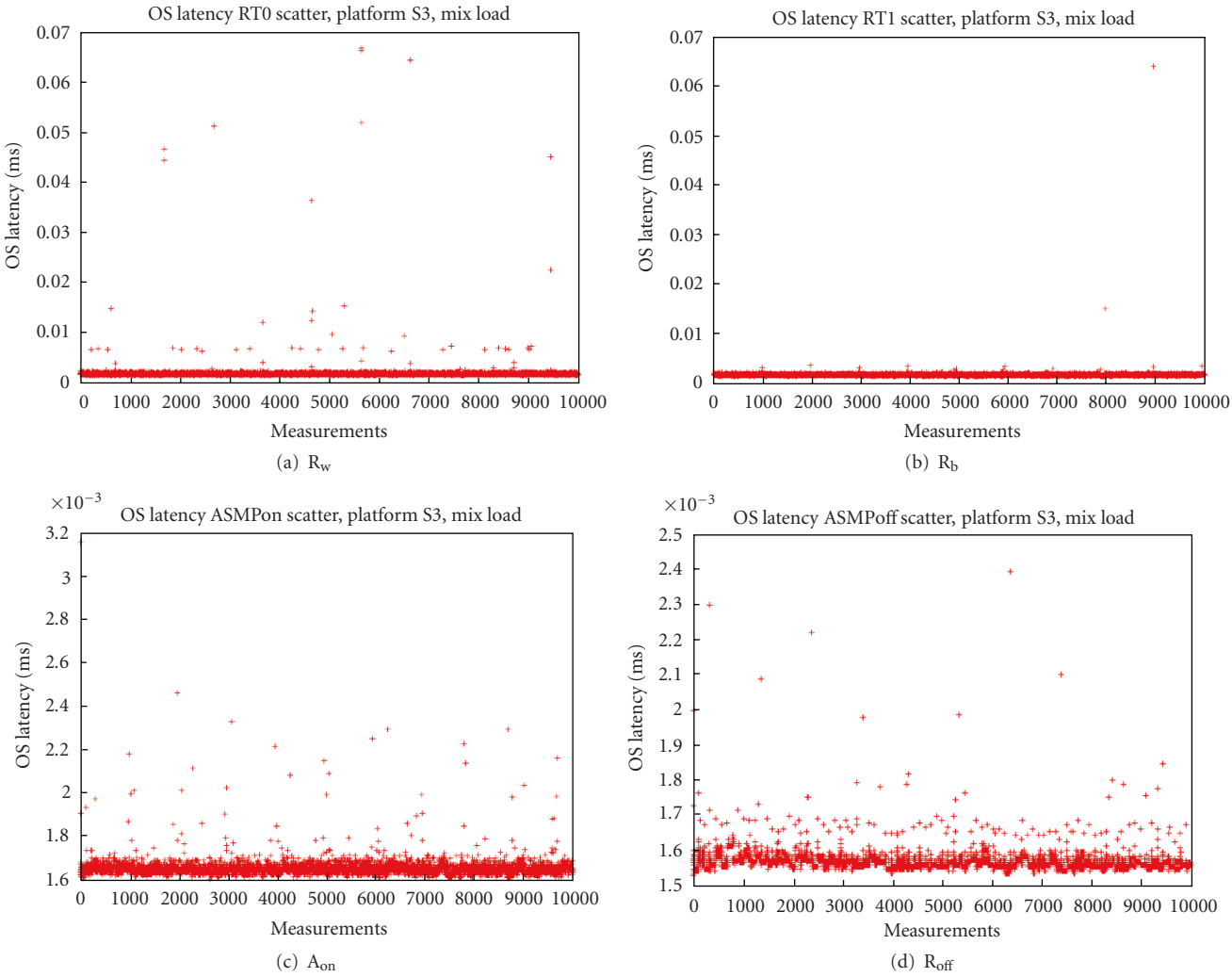


Figure 10: Scatter graphics for system S3, MIX workload.

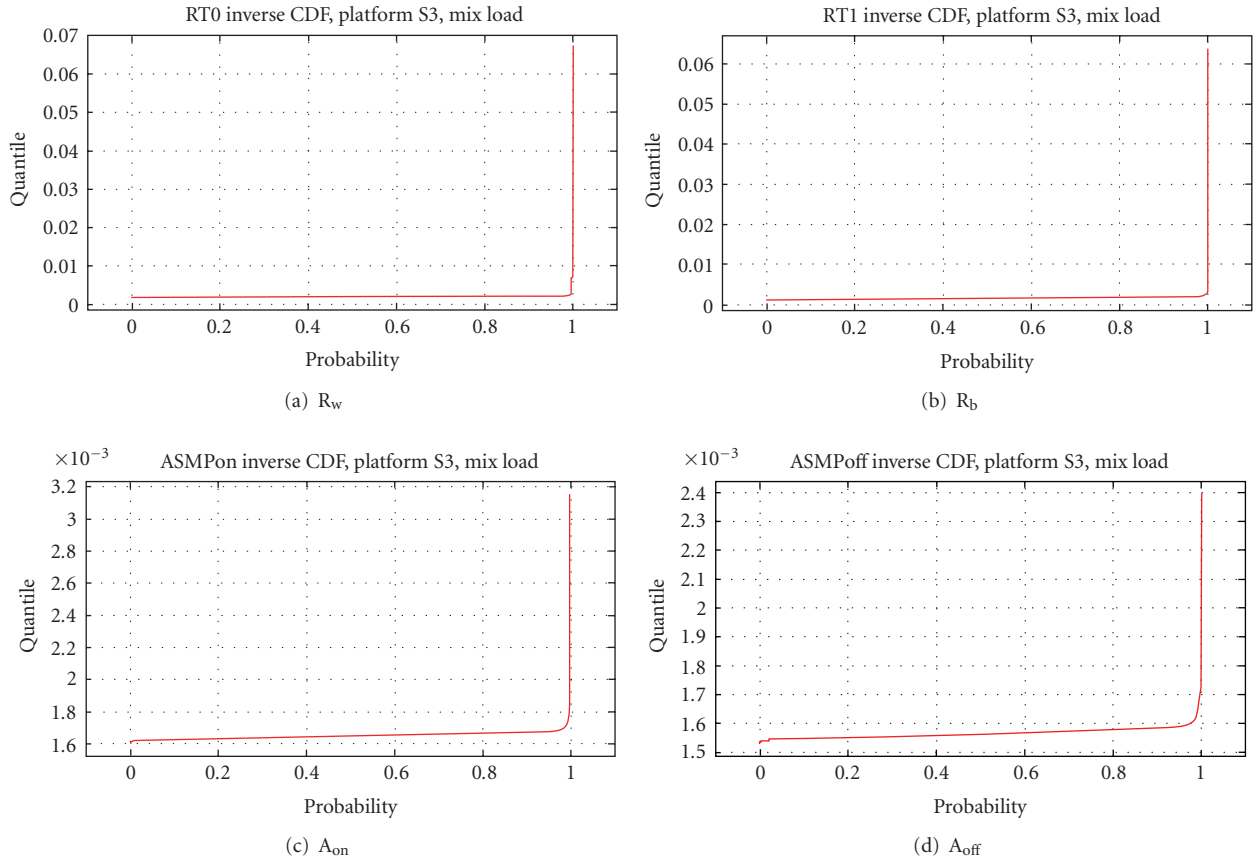
6.4. *Final consideration*

The goal of these tests was to evaluate ASMP-Linux on different Platforms. In fact, each Platform has benefits and drawbacks: for example, Platform S1 is the less power consuming architecture because the virtual processors are not full CPUs; however, ASMP-Linux does not provide hard real-time performances on this Platform. Conversely, ASMP-Linux provides hard real-time performances when running on Platform S2 but this Platform is the most expensive in terms of cost, surface, and power consumption, thus we do not think it will fit well with embedded systems' constraints. Platform S3 is a good tradeoff between the previous two Platforms: ASMP-Linux still provides hard real-time performance even if the two cores share some resources, resulting in reduced chip surface and power consumption. Moreover, the tested processor has been specifically designed for power-critical system (such as laptops), thus we foreseen it will be largely used in embedded systems, as it happened with its predecessor single-core version.

7. CONCLUSIONS AND FUTURE WORK

In this paper we introduced ASMP-Linux as a fundamental component for an hard real-time operating system based on the Linux kernel for MP-embedded systems. We first introduced the notion of jitter and classified it in *hardware delay* and operating system *latency* and *overhead*. Then, we explained the asymmetric philosophy of ASMP-Linux and its internal details as well as how real-time applications might not catch their deadline because of jitter. Finally, we presented our experiment environments and tests: the test results show how ASMP-Linux is capable of minimizing both operating system overhead and latency, thus providing deterministic results for the tested applications.

Even if these results are encouraging, ASMP-Linux is not a complete hard real-time operating system. We are planning to add more features to ASMP-Linux in order to achieve this goal. Specifically, we are planning to add a new scheduler class for hard real-time applications that run on a shielded partition. Moreover, we plan to merge the ASMP-Linux kernel patch with the new timekeeping architecture that has



(a) R_w (b) R_b

(c) A_{on} (d) A_{off}

FIGURE 11: Inverse density functions for overhead on system S3, MIX workload.

been introduced in the Linux 2.6.21 kernel version, in particular, with the high-resolution timers and the dynamic ticks: this will improve the performances of periodic real-time tasks. Finally, we will provide interpartition channels so that hard real-time applications can exchange data with nonreal-time applications running in the system partition without affecting the hard real-time performances of the critical tasks.

REFERENCES

[1] M. Momtchev and P. Marquet, "An open operating system for intensive signal processing," Tech. Rep. 2001-08, Lab-Oratoire d'Informatique Fondamentale de Lille, Université des Sciences et Technologies de Lille, Villeneuve d'Ascq Cedex, France, 2001.

[2] P. Marquet, E. Piel, J. Soula, and J.-L. Dekeyser, "Implementation of ARTiS, an asymmetric real-time extension of SMP Linux," in *The 6th Real-Time Linux Workshop*, Singapore, November 2004.

[3] R. Gioiosa, "Asymmetric kernels for multiprocessor systems," M.S. thesis, University of Rome, Tor Vergata, Italy, October 2002.

[4] S. Brosky and S. Rotolo, "Shielded processors: guaranteeing sub-millisecond response in standard Linux," in *The 4th Real-Time Linux Workshop*, Boston, Mass, USA, December 2002.

[5] IBM Corp., "The Cell project at IBM Research," http://www.research.ibm.com/cell/home.html.

[6] L. Eggermont, Ed., *Embedded Systems Roadmap*, STW Technology Foundation, 2002, http://www.stw.nl/Programmas/Progress/ESroadmap.htm.

[7] Intel Corp., "Intel Core2 Duo Mobile Processor Datasheet," ftp://download.intel.com/design/mobile/SPECUPDT/31407917.pdf, 2006.

[8] H. M. Mathis, A. E. Mericas, J. D. McCalpin, R. J. Eickemeyer, and S. R. Kunkel, "Characterization of simultaneous multithreading (SMT) efficiency in POWER5," *IBM Journal of Research and Development*, vol. 49, no. 4-5, pp. 555–564, 2005.

[9] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel, "A multithreaded PowerPC processor for commercial servers," *IBM Journal of Research and Development*, vol. 44, no. 6, pp. 885–898, 2000.

[10] S. Storino, A. Aipperspach, J. Borkenhagen, et al., "Commercial multi-threaded RISC processor," in *Proceedings of the IEEE 45th International Solid-State Circuits Conference, Digest of Technical Papers (ISSCC '98)*, pp. 234–235, San Francisco, Calif, USA, February 1998.

[11] J. M. Tendler, J. S. Dodson, J. S. Fields Jr., H. Le, and B. Sinharoy, "POWER4 system microarchitecture," *IBM Journal of Research and Development*, vol. 46, no. 1, pp. 5–25, 2002.

[12] Intel Corp., "Intel Pentium D Processor 900 sequence and Intel Pentium Processor Extreme Edition 955, 965 datasheet," ftp://download.intel.com/design/PentiumXE/datashts/31030607.pdf, 2006.

[13] Advanced Micro Devices, "AMD Opteron™ Processor Product Data Sheet," http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/23932.pdf, 2006.

[14] Sun Microsystems, "UltraSPARC® IV Processor Architecture Overview," http://www.sun.com/processors/whitepapers/us4_whitepaper.pdf, February 2004.
[15] Intel Corp., "Intel Quad-Core processors," http://www.intel.com/technology/quad-core/index.htm.
[16] P. McKenney, "SMP and embedded real-time," *Linux Journal*, vol. 2007, no. 153, p. 1, 2007, http://www.linuxjournal.com/article/9361.
[17] R. Gioiosa, F. Petrini, K. Davis, and F. Lebaillif-Delamare, "Analysis of system overhead on parallel computers," in *Proceedings of the 4th IEEE International Symposium on Signal Processing and Information Technology (ISSPIT '04)*, pp. 387–390, Rome, Italy, December 2004.
[18] F. Petrini, D. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: achieving optimal performance on the 8,192 processors of ASCI Q," in *ACM/IEEE Conference Supercomputing (SC '03)*, p. 55, Phoenix, Ariz, USA, November 2003.
[19] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick, "System noise, OS clock ticks, and fine-grained parallel applications," in *Proceedings of the 19th ACM International Conference on Supercomputing (ICS '05)*, pp. 303–312, ACM Press, June 2005.
[20] S. Schönberg, "Impact of PCI-bus load on applications in a PC architecture," in *Proceedings of the 24th IEEE International Real-Time Systems Symposium (RTSS '03)*, pp. 430–438, Cancun, Mexico, December 2003.
[21] L. Dozio and P. Mantegazza, "Real time distributed control systems using RTAI," in *Proceedings of the 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '03)*, pp. 11–18, Hokkaido, Japan, May 2003.
[22] K. Yaghmour, "Adaptive domain environment for operating systems," http://www.opersys.com/ftp/pub/Adeos/adeos.pdf, 2001.
[23] Wind River, "VxWorks programmer guide," http://www.windriver.com/, 2003.
[24] Free Software FoundationInc., "GNU General Public License, version 2," http://www.gnu.org/licenses/gpl2.html, June 1991.
[25] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, O'Reilly Media, 3rd edition, 2005.
[26] Intel Corp., "Intel Xeon processors," http://download.intel.com/design/Xeon/datashts/30624902.pdf.
[27] Intel Corp., "Hyper-Threading technology," http://www.intel.com/technology/platform-technology/hyper-threading/index.htm.
[28] Intel Corp., "Intel Core2 Extreme Processor X6800 and Intel Core2 Duo Desktop Processor E6000 sequence datasheet," http://www.intel.com/design/processor/datashts/313278.htm, 2006.
[29] E. Betti, D. P. Bovet, M. Cesati, and R. Gioiosa, "Hard real-time performances in multiprocessor embedded systems using ASMP-Linux," Tech. Rep. TR002, System Programming Research Group, Department of Computer Science, Systems and Production, University of Rome "Tor Vergata", Rome, Italy, 2007, http://www.sprg.uniroma2.it/asmplinux/.