EURASIP Journal on Embedded Systems

Operating System Support for Embedded Real-Time Applications

Guest Editors: Alfons Crespo, Ismael Ripoll,
Michael Gonzalez Harbour, and Giuseppe Lipari

Operating System Support for Embedded Real-Time Applications

EURASIP Journal on Embedded Systems

Operating System Support for Embedded Real-Time Applications

Guest Editors: Alfons Crespo, Ismael Ripoll, Michael Gonzalez Harbour, and Giuseppe Lipari

Contents

Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 502768, 2 pages
doi:10.1155/2008/502768

Editorial

Operating System Support for Embedded Real-Time Applications

Alfons Crespo,[1] Ismael Ripoll,[1] Michael González-Harbour,[2] and Giuseppe Lipari[3]

[1] *Departament d'Informatica de Sistems i Computadors, Universidad Politécnica de València, 46022 Valencia, Spain*
[2] *Universidad de Cantabria, 39005 Santander, Spain*
[3] *Scuola Superiore Santa'Anna, 335612 Pisa, Italy*

Correspondence should be addressed to Alfons Crespo, acrespo@disca.upv.es

Received 18 February 2008; Accepted 18 February 2008

The rapid progress in processor and sensor technology combined with the expanding diversity of application fields is placing enormous demands on the facilities that an embedded operating system must provide.

Embedded systems can be defined as computing systems with tightly coupled hardware and software that are designed to perform a dedicated function. The word embedded reflects the fact that these systems are usually an integral part of a larger system.

We can find a large variety of applications where embedded systems play an important role, from small stand-alone systems, like a network router, to complex embedded systems supporting several operating execution environments as we can find in avionic applications.

This variety of applications also implies that the properties, platforms, and techniques on which embedded systems are based can be very different. The hardware needs can sometimes be achieved with the use of general purpose processors, but in many systems specific processors are required, for instance, specific DSP devices to perform fast signal processing. Memory management capabilities are necessary in some systems to provide memory protection and virtual memory. Special purpose interfaces are also needed to support a variety of external peripheral devices, energy consumption control, and so on.

Nowadays, the use of processor-based devices has increased dramatically for most of our activities, both professional and leisure. Mobile phones and PDAs are used extensively. Consumer electronics (set-top boxes, TVs, DVD players, etc.) have incorporated microprocessors as a core system component, instead of using specific hardware. This trend is expected to grow exponentially in the near future.

Embedded applications have some common features such as the following.

(i) Limited resources. They are often strong limitations regarding available resources. Mainly due to cost and size constraints related to mass production and strong industrial competition, the system resources as CPU, memory, devices have been designed to meet these requirements. As a result of these limitations, the system has to deal with an efficient use of the computational resources.

(ii) Real-time application requirements. Some of the applications to be run in these devices have temporal requirements. These applications are related with process control, multimedia processing, instrumentation, and so on, where the system has to act within a specified interval.

(iii) Embedded control systems. Most of the embedded systems perform control activities involving input data acquisition (sensing) and output delivery (actuation). Deterministic communications are also another important issue.

(iv) Quality of service. An efficient use of the system resources is a must in embedded systems. Feedback-based approaches are being used to adjust the performance or quality of service of the applications as a function of the available resources.

The challenge is how to implement applications that can execute efficiently on limited resource and that meet nonfunctional requirements such as timeliness, robustness, dependability, performance, and so on.

Moreover, applications on embedded systems include more and more functionalities in order to cope with the needs of the users in home environments, industry, leisure activities, vehicles, avionics, instrumentation, and so on. To offer services for these applications, a considerable effort has been made in research and development on innovative real-time operating systems architectures and services. Designers and developers of real-time operating systems have to consider many challenges arising from two opposite axes: efficient resource usage (processor, memory, energy, network bandwidth, etc.) and dynamic configuration and adaptation (component-based development and deployment, flexible scheduling, communications, etc.).

On the other hand, open-source operating system development has been recognized as a consolidated way to share experiences and developments in order to improve the quality and the reusability of the products. Currently, there are several distributions for embedded systems based on Linux and other open source developments.

This special issue focuses on new results of research work and development in the field of real-time operating systems for embedded applications with special emphasis on open source developments.

From the real-time operating system point of view, there are several topics that can be considered very relevant in the near future, illustrated as follows.

Partitioned systems

The development of embedded applications is entering into a new domain with the availability of new high-speed processors and low cost on-chip memory. As a result of these new developments in hardware, there is an interest in enabling multiple applications to share a single processor and memory. To facilitate such a model the execution time and memory space of each application must be protected from other applications in the system. Partitioning operating systems represents the future of secure systems. They have evolved to fulfill security and avionics requirements where predictability is extremely important. In avionics systems, for instance, running interrupts other than the system clock needed for cycling the partitions is discouraged. In a partitioning operating system, memory (and possibly CPU-time as well) is divided among statically allocated partitions in a fixed manner. The idea is to take a processor and make it appear as if there were several processors, thus completely isolating the subsystems. Within each partition, there may be multiple threads or processes, or both, if the operating system supports them. How these threads are scheduled depends on the implementation of the OS.

Innovative techniques

Innovative techniques in scheduling are needed to provide support for adjusting the load to the system needs, managing the dynamic allocation of memory under temporal and spatial constraints, managing energy to allow trading performance for reduced energy consumption in combination with the time constraints, providing fault-tolerance to deal with failure management and recovery, and so on.

Security

Embedded systems are getting more and more complex, dynamic, and open, while interacting with a progressively more demanding and heterogeneous environment. As a consequence, the reliability and security of these systems have become major concerns. An increasing number of external security attacks as well as design weaknesses in operating systems have resulted in large economic damages, which results in difficulties to attain user acceptance and getting accepted by the market. Consequently, there is a growing request from stakeholders in embedded systems to make available execution platforms which address both integrity and security concerns. For instance, it is important to avoid denial of service issues provoked by resource shortage (e.g., memory, CPU), while from an integrity viewpoint it is important to ensure availability of resources. It is also important to prevent malicious access to data created by another application.

Other aspects

Other aspects such as mutiprocessor system support, power-aware operating systems, real-time communications will have a relevant role in the next generation of embedded systems.

In this issue, several papers offer the particular vision of these issues. The first paper provides an approach of partitioned systems based on the L4 microkernel, whereas the second paper proposes a multiprocessor embedded system based on ASMP-Linux. The third and fourth papers deal with resource and reconfiguration management. The last two papers present application environments where the real-time operating systems present specific services to fulfill the requirements of these applications.

Alfons Crespo
Ismael Ripoll
Michael González-Harbour
Giuseppe Lipari

Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 234710, 14 pages
doi:10.1155/2008/234710

Research Article

A Real-Time Programmer's Tour of General-Purpose L4 Microkernels

Sergio Ruocco

Laboratorio Nomadis, Dipartimento di Informatica, Sistemistica e Comunicazione (DISCo), Università degli Studi di Milano-Bicocca, 20126 Milano, Italy

Correspondence should be addressed to Sergio Ruocco, ruocco@disco.unimib.it

Received 20 February 2007; Revised 26 June 2007; Accepted 1 October 2007

Recommended by Alfons Crespo

L4-embedded is a microkernel successfully deployed in mobile devices with soft real-time requirements. It now faces the challenges of tightly integrated systems, in which user interface, multimedia, OS, wireless protocols, and even software-defined radios must run on a single CPU. In this paper we discuss the pros and cons of L4-embedded for real-time systems design, focusing on the issues caused by the extreme speed optimisations it inherited from its general-purpose ancestors. Since these issues can be addressed with a minimal performance loss, we conclude that, overall, the design of real-time systems based on L4-embedded is possible, and facilitated by a number of design features unique to microkernels and the L4 family.

1. INTRODUCTION

Mobile embedded systems are the most challenging front of real-time computing today. They run full-featured operating systems, complex multimedia applications, and multiple communication protocols at the same time. As networked systems, they are exposed to security threats; moreover, their (inexperienced) users run untrusted code, like games, which pose both security and real-time challenges. Therefore, complete isolation from untrusted applications is indispensable for user data confidentiality, proper system functioning, and content-providers and manufacturer's IP protection.

In practice, today's mobile systems must provide functionalities equivalent to desktop and server ones, but with severely limited resources and strict real-time constraints. Conventional RTOSes are not well suited to meet these requirements: simpler ones are not secure, and even those with memory protection are generally conceived as embedded software platforms, not as operating system foundations.

L4-embedded [1] is an embedded variant of the general-purpose microkernel L4Ka::Pistachio (L4Ka) [2] that meets the above-mentioned requirements, and has been successfully deployed in mobile phones with soft real-time constraints. However, it is now facing the challenges of next-generation mobile phones, where applications, user interface, multimedia, OS, wireless protocols, and even software-defined radios must run on a single CPU.

Can L4-embedded meet such strict real-time constraints? It is thoroughly optimized and is certainly fast, but "real fast is not real-time" [3]. Is an entirely new implementation necessary, or are small changes sufficient? What are these changes, and what are the tradeoffs involved? In other words, can L4-embedded be real fast *and* real-time?

The aim of this paper is to shed some light on these issues with a thorough analysis of the L4Ka and L4-embedded internals that determine their temporal behaviour, to assess them as strengths or weaknesses with respect to real-time, and finally to indicate where research and development are currently focusing, or should probably focus, towards their improvement.

We found that (i) general-purpose L4 microkernels contain in their IPC path extreme optimisations which complicate real-time scheduling; however these optimisations can be removed with a minimal performance loss; (ii) aspects of the L4 design provide clear advantages for real-time applications. For example, thanks to the unified user-level scheduling for both interrupt and application threads, interrupt handlers and device drivers cannot impact system timeliness. Moreover, the interrupt subsystem provides a good foundation for user-level real-time scheduling.

Overall, although there is still work ahead, we believe that with few well-thought-out changes, general-purpose L4 microkernels can be used successfully as the basis of a significant class of real-time systems.

The rest of the paper is structured as follows. Section 2 introduces microkernels and the basic principles of their design, singling out the relevant ones for real-time systems. Section 3 describes the design of L4 and its API. Section 4 analyses L4-embedded and L4Ka internals in detail, their implications for real-time system design, and sketches future work. Finally, Section 5 concludes the paper.

2. MICROKERNELS

Microkernels are minimalist operating system kernels structured according to specific design principles. They implement only the smallest set of abstractions and operations that require privileges, typically address spaces, threads with basic scheduling, and message-based interprocess communication (IPC). All the other features which may be found in ordinary monolithic kernels (such as drivers, filesystems, paging, networking, etc.) but can run in user mode are implemented in user-level servers. Servers run in separate protected address spaces and communicate via IPC and shared memory using well-defined protocols.

The touted benefits of building an operating system on top of a microkernel are better modularity, flexibility, reliability, trustworthiness, and viability for multimedia and real-time applications than those possible with traditional monolithic kernels [4]. Yet operating systems based on first-generation microkernels like Mach [5] did not deliver the promised benefits: they were significantly slower than their monolithic counterparts, casting doubts on the whole approach. In order to regain some performance, Mach and other microkernels brought back some critical servers and drivers into the kernel protection domain, compromising the benefits of microkernel-based design.

A careful analysis of the real causes of Mach's lacklustre performance showed that the fault was not in the microkernel approach, but in its initial implementation [6]. The first-generation microkernels were derived by scaling down monolithic kernels, rather than from clean-slate designs. As a consequence, they suffered from poorly performing IPC and excessive footprint that thrashed CPU caches and translation lookaside buffers (TLBs). This led to a second generation of microkernels designed from scratch with a minimal and clean architecture, and strong emphasis on performance. Among them are Exokernels [7], L4 [6], and Nemesis [8].

Exokernels, developed at MIT in 1994-95, are based on the idea that kernel abstractions restrict flexibility and performance, and hence they must be *eliminated* [9]. The role of the exokernel is to securely multiplex hardware, and export *primitives* for applications to freely implement the abstractions that best satisfy their requirements.

L4, developed at GMD in 1995 as a successor of L3 [10], is based on a design philosophy less extreme than exokernels, but equally aggressive with respect to performance. L4 aims to provide flexibility and performance to an operating system via the least set of privileged abstractions.

Nemesis, developed at the University of Cambridge in 1993–95, has the aim of providing quality-of-service (QoS) guarantees on resources like CPU, memory, disk, and network bandwidth to multimedia applications.

Besides academic research, since the early 1980s the embedded software industry developed and deployed a number of microkernel-based RTOSes. Two prominent ones are QNX and GreenHills Integrity. QNX was developed in the early 1980s for the 80x86 family of CPUs [11]. Since then it evolved and has been ported to a number of different architectures. GreenHills Integrity is a highly optimised commercial embedded RTOS with a preemptable kernel and low-interrupt latency, and is available for a number of architectures.

Like all microkernels, QNX and Integrity as well as many other RTOSes rely on user-level servers to provide OS functionality (filesystems, drivers, and communication stacks) and are characterised by a small size.[1] However, they are generally conceived as a basis to run embedded applications, not as a foundation for operating systems.

2.1. Microkernels and real-time systems

On the one hand, microkernels are often associated with real-time systems, probably due to the fact that multimedia and embedded real-time applications running on resource-constrained platforms benefit from their small footprint, low-interrupt latency, and fast interprocess communication compared to monolithic kernels. On the other hand, the general-purpose microkernels designed to serve as a basis for workstation and server Unices in the 1990s were apparently meant to address real-time issues of a different nature and a coarser scale, as real-time applications on general-purpose systems (typically multimedia) had to compete with many other processes and to deal with large kernel latency, memory protection, and swapping.

Being a microkernel, L4 has intrinsic provisions for real-time. For example, user-level memory pagers enable application-specific paging policies. A real-time application can explicitly pin the logical pages that contain time-sensitive code and data in physical memory, in order to avoid page faults (also TLB entries should be pinned, though).

The microkernel design principle that is more helpful for real-time is user-level device drivers [12]. In-kernel drivers can disrupt time-critical scheduling by disabling interrupts at arbitrary points in time for an arbitrary amount of time, or create deferred workqueues that the kernel will execute at unpredictable times. Both situations can easily occur, for example, in the Linux kernel, and only very recently they have started to be tackled [13]. Interrupt disabling is just one of the many critical issues for real-time in monolithic kernels. As we will see in Section 4.7, the user-level device driver model of L4 avoids this and other problems. Two other L4 features intended for real-time support are IPC timeouts, used for time-based activation of threads (on timeouts see

[1] Recall that the "micro" in microkernel refers to its economy of concepts compared to monolithic kernels, not to its memory footprint.

Sections 3.5 and 4.1), and *preempters*, handlers for time faults that receive preemption notification messages.

In general, however, it still remains unclear whether the above-mentioned second-generation microkernels are well suited for all types of real-time applications. A first examination of exokernel and Nemesis scheduling APIs reveals, for example, that both hardwire scheduling policies that are disastrous for at least some classes of real-time systems and cannot be avoided from the user level. Exokernel's primitives for CPU sharing achieve "fairness [by having] applications pay for each excess time slice consumed by forfeiting a subsequent time slice" (see [14], page 32). Similarly, Nemesis' CPU allocation is based on a "simple QoS specification" where applications "specify neither priorities nor deadlines" but are provided with a "particular *share* of the processor over some short time frame" according to a (replaceable) scheduling algorithm. The standard Nemesis scheduling algorithm, named Atropos, "internally uses an earliest deadline first algorithm to provide this share guarantee. However, the deadlines on which it operates are *not* available to or specified by the application" [8].

Like many RTOSes, L4 contains a priority-based scheduler hardwired in the kernel. While this limitation can be circumvented with some ingenuity via user-level scheduling [15] at the cost of additional context-switches, "all that is wired in the kernel cannot be modified by higher levels" [16]. As we will see in Section 4, this is exactly the problem with some L4Ka optimisations inherited by L4-embedded, which, while being functionally correct, trade predictability and freedom from policies for performance and simplicity of implementation, thus creating additional issues that designers must be aware of, and which time-sensitive systems must address.

3. THE L4 MICROKERNEL

L4 is a second-generation microkernel that aims at high flexibility and maximum performance, but without compromising security. In order to be fast, L4 strives to be small by design [16], and thus provides only the least set of fundamental abstractions and the mechanisms to control them: address spaces with memory-mapping operations, threads with basic scheduling, and synchronous IPC.

The emphasis of L4 design on smallness and flexibility is apparent in the implementation of IPC and its use by the microkernel itself. The basic IPC mechanism is used not only to transfer messages between user-level threads, but also to deliver interrupts, asynchronous notifications, memory mappings, thread startups, thread preemptions, exceptions and page faults. Because of its pervasiveness, but especially its impact on OS performance experienced with first-generation microkernels, L4 IPC has received a great deal of attention since the very first designs [17] and continues to be carefully optimised today [18].

3.1. The L4 microkernel specification

In high-performance implementations of system software there is an inherent contrast between maximising the performance of a feature on a specific implementation of an architecture and its portability to other implementations or across architectures. L4 faced these problems when transitioning from 80486 to the Pentium, and then from Intel to various RISC, CISC, and VLIW 32/64 bit architectures.

L4 addresses this problem by relying on a specification of the microkernel. The specification is crafted to meet two apparently conflicting objectives. The first is to guarantee full compatibility and portability of user-level software across the matrix of microkernel implementations and processor architectures. The second is to leave to kernel engineers the maximum leeway in the choice of architecture-specific optimisations and tradeoffs among performance, predictability, memory footprint, and power consumption.

The specification is contained in a reference manual [19] that details the hardware-independent L4 API and 32/64 bit ABI, the layout of public kernel data structures such as the user thread control block (UTCB) and the kernel information page (KIP), CPU-specific extensions to control caches and frequency, and the IPC protocols to handle, among other things, memory mappings and interrupts at the user-level.

In principle, every L4 microkernel implementation should adhere to its specification. In practice, however, some deviations can occur. To avoid them, the L4-embedded specification is currently being used as the basis of a regression test suite, and precisely defined in the context of a formal verification of its implementation [20].

3.2. The L4 API and its implementations

L4 evolved over time from the original L4/x86 into a small family of microkernels serving as vehicles for OS research and industrial applications [19, 21]. In the late 1990s, because of licensing problems with then-current kernel, the L4 community started the Fiasco [22, 23] project, a variant of L4 that, during its implementation, was made preemptable via a combination of lock-free and wait-free synchronisation techniques [24]. DROPS [25] (Dresden real-time operating system) is an OS personality that runs on top of Fiasco and provides further support for real-time besides the preemptability of the kernel, namely a scheduling framework for periodic real-time tasks with known execution times distributions [26].

Via an entirely new kernel implementation Fiasco tackled many of the issues that we will discuss in the rest of the paper: timeslice donation, priority inversion, priority inheritance, kernel preemptability, and so on [22, 27, 28]. Fiasco solutions, however, come at the cost of higher kernel complexity and an IPC overhead that has not been precisely quantified [28].

Unlike the Fiasco project, our goal is not to develop a new real-time microkernel starting with a clean slate and freedom from constraints, but to analyse and improve the real-time properties of NICTA::Pistachio-embedded (L4-embedded), an implementation of the N1 API specification [1] already deployed in high-end embedded and mobile systems as a virtualisation platform [29].

Both the L4-embedded specification and its implementation are largely based on L4Ka::Pistachio version 0.4 (L4Ka) [2], with special provisions for embedded systems such as

a reduced memory footprint of kernel data structures, and some changes to the API that we will explain later. Another key requirement is IPC performance, because it directly affects virtualisation performance.

Our questions are the following ones: can L4-embedded support "as it is" real-time applications? Is an entirely new implementation necessary, or can we get away with only small changes in the existing one? What are these changes, and what are the tradeoffs involved?

In the rest of the paper, we try to give an answer to these questions by discussing the features of L4Ka and L4-embedded that affect the applications' temporal behaviour on uniprocessor systems (real-time on SMP/SMT systems entails entirely different considerations, and its treatment is outside the scope of this paper). They include scheduling, synchronous IPC, timeouts, interrupts, and asynchronous notifications.

Please note that this paper mainly focuses on L4Ka::Pistachio version 0.4 and L4-embedded N1 microkernels. For the sake of brevity, we will refer to them as simply L4, but the reader should be warned that much of the following discussion applies only to these two versions of the kernel. In particular, Fiasco makes completely different design choices in many cases. For reasons of space, however, we cannot go in depth. The reader should refer to the above-mentioned literature for further information.

3.3. Scheduler

The L4 API specification defines a 256-level, fixed-priority, time-sharing round-robin (RR) scheduler. The RR scheduling policy runs threads in priority order until they block in the kernel, are preempted by a higher priority thread, or exhaust their timeslice. The standard length of a timeslice is 10 ms but can be set between ϵ (the shortest possible timeslice) and ∞ with the `Schedule()` system call. If the timeslice is different from ∞, it is rounded to the minimum granularity allowed by the implementation that, like ϵ, ultimately depends on the precision of the algorithm used to update it and to verify its exhaustion (on timeslices see Sections 4.1, 4.4, and 4.5). Once a thread exhausts its timeslice, it is enqueued at the end of the list of the running threads of the same priority, to give other threads a chance to run. RR achieves a simple form of fairness and, more importantly, guarantees progress.

FIFO is a scheduling policy closely related to RR that does not attempt to achieve fairness and thus is somewhat more appropriate for real-time. As defined in the POSIX 1003.1b real-time extensions [30], FIFO-scheduled threads run until they relinquish control by yielding to another thread or by blocking in the kernel. L4 can emulate FIFO with RR by setting the threads' priorities to the same level and their timeslices to ∞. However, a maximum of predictability is achieved by assigning only one thread to each priority level.

3.4. Synchronous IPC

L4 IPC is a rendezvous in the kernel between two threads that partner to exchange a message. To keep the kernel simple and fast, L4 IPC is synchronous: there are no buffers or message ports, nor double copies, in and out of the kernel. Each partner performs an `Ipc(dest, from_spec, &from)` syscall that is composed of an optional send phase to the *dest* thread, followed by an optional receive phase from a thread specified by the *from_spec* parameter. Each phase can be either blocking or nonblocking. The parameters *dest* and *from_spec* can take values among all standard thread ids. There are some special thread ids, among which there are *nilthread* and *anythread*. The *nilthread* encodes "send-only" or "receive-only" IPCs. The *anythread* encodes "receive from any thread" IPCs.

Under the assumptions that IPC syscalls issued by the two threads cannot execute simultaneously, and that the first invoker requests a blocking IPC, the thread blocks and the scheduler runs to pick a thread from the ready queue. The first invoker remains blocked in the kernel until a suitable partner performs the corresponding IPC that transfers a message and completes the communication. If the first invoker requests a nonblocking IPC and its partner is not ready (i.e., not blocked in the kernel waiting for it), the IPC aborts immediately and returns an error.

A convenience API prescribed by the L4 specification provides wrappers for a number of common IPC patterns encoding them in terms of the basic syscall. For example, `Call(dest)`, used by clients to perform a simple IPC to servers, involves a blocking send to thread *dest*, followed by a blocking receive from the same thread. Once the request is performed, servers can reply and then block waiting for the next message by using `ReplyWait(dest, &from_tid)`, an IPC composed of a nonblocking send to *dest* followed by a blocking receive from *anythread* (the send is nonblocking as typically the caller is waiting, thus the server can avoid blocking trying to send replies to malicious or crashed clients). To block waiting for an incoming message one can use `Wait()`, a send to *nilthread* and a blocking receive from *anythread*. As we will see in Section 4.4, for performance optimisations the threads that interact in IPC according to some of these patterns are scheduled in special (and sparsely documented) ways.

L4Ka supports two types of IPC: standard IPC and long IPC. Standard IPC transfers a small set of 32/64-bit message registers (MRs) residing in the UTCB of the thread, which is always mapped in the physical memory. Long IPC transfers larger objects, like strings, which can reside in arbitrary, potentially unmapped, places of memory. Long IPC has been removed from L4-embedded because it can page-fault and, on nonpreemptable kernels, block interrupts and the execution of other threads for a large amount of time (see Section 4.7). Data transfers larger than the set of MRs can be performed via multiple IPCs or shared memory.

3.5. IPC timeouts

IPC with timeouts cause the invoker to block in the kernel until either the specified amount of time has elapsed or the partner completes the communication. Timeouts were originally intended for real-time support, and also as a way for clients to recover safely from the failure of servers by aborting a pending request after a few seconds (but a good way to

determine suitable timeout values was never found). Timeouts are also used by the `Sleep`() convenience function, implemented by L4Ka as an IPC to the current thread that times out after the specified amount of microseconds. Since timeouts are a vulnerable point of IPC [31], they unnecessarily complicate the kernel, and more accurate alternatives can be implemented by a time server at user level, they have been removed from L4-embedded (Fiasco still has them, though).

3.6. User-level interrupt handlers

L4 delivers a hardware interrupt as a synchronous IPC message to a normal user-level thread which registered with the kernel as the *handler thread* for that interrupt. The interrupt messages appear to be sent by special in-kernel *interrupt threads* set up by L4 at registration time, one per interrupt. Each interrupt message is delivered to exactly one handler, however a thread can be registered to handle different interrupts. The timer tick interrupt is the only one managed internally by L4.

The kernel handles an interrupt by masking it in the interrupt controller (IC), preempting the current thread, and performing a sequence of steps equivalent to an IPC `Call`() from the in-kernel interrupt thread to the user-level handler thread. The handler runs in user-mode with its interrupt disabled, but the other interrupts enabled, and thus it can be preempted by higher-priority threads, which possibly, but not necessarily, are associated with other interrupts. Finally, the handler signals that it finished servicing the request with a `Reply`() to the interrupt thread, that will then unmask the associated interrupt in the IC (see Section 4.7).

3.7. Asynchronous notification

Asynchronous notification is a new L4 feature introduced in L4-embedded, not present in L4Ka. It is used by a sender thread to notify a receiver thread of an event. It is implemented via the IPC syscall because it needs to interact with the standard synchronous IPC (e.g., applications can wait with the same syscall for either an IPC or a notification). However, notification is neither blocking for the sender, nor requires the receiver to block waiting for the notification to happen. Each thread has 32 (64 on 64-bit systems) notification bits. The sender and the receiver must agree beforehand on the semantics of the event, and which bit signals it. When delivering asynchronous notification, L4 does not report the identity of the notifying thread: unlike in synchronous IPC, the receiver is only informed of the event.

4. L4 AND REAL-TIME SYSTEMS

The fundamental abstractions and mechanisms provided by the L4 microkernel are implemented with data structures and algorithms chosen to achieve speed, compactness, and simplicity, but often disregarding other nonfunctional aspects, such as timeliness and predictability, which are critical for real-time systems.

In the following, we highlight the impact of some aspects of the L4 design and its implementations (mainly L4Ka and L4-embedded, but also their ancestors), on the temporal behaviour of L4-based systems, and the degree of control that user-level software can exert over it in different cases.

Table 1: Timer tick periods.

Version	Architecture	Timer tick (μs)
L4-embedded N1	StrongARM	10 000
L4-embedded N1	XScale	5000
L4::Ka Pistachio 0.4	Alpha	976
L4::Ka Pistachio 0.4	AMD64	1953
L4::Ka Pistachio 0.4	IA-32	1953
L4::Ka Pistachio 0.4	PowerPC32	1953
L4::Ka Pistachio 0.4	Sparc64	2000
L4::Ka Pistachio 0.4	PowerPC64	2000
L4::Ka Pistachio 0.4	MIPS64	2000
L4::Ka Pistachio 0.4	IA-64	2000
L4::Ka Pistachio 0.4	StrongARM/XScale	10 000

4.1. Timer tick interrupt

The timer tick is a periodic timer interrupt that the kernel uses to perform a number of time-dependent operations. On every tick, L4-embedded and L4Ka subtract the tick length from the remaining timeslice of the current thread and preempt it if the result is less than zero (see Algorithm 1). In addition, L4Ka also inspects the wait queues for threads whose timeout has expired, aborts the IPC they were blocked on and marks them as runnable. On some platforms L4Ka also updates the kernel internal time returned by the `SystemClock`() syscall. Finally, if any thread with a priority higher than the current one was woken up by an expired timeout, L4Ka will switch to it immediately.

Platform-specific code sets the timer tick at kernel initialisation time. Its value is observable (but not changeable) from user space in the `SchedulePrecision` field of the `ClockInfo` entry in the KIP. The current values for L4Ka and L4-embedded are in Table 1 (note that the periods can be trivially made uniform across platforms by editing the constants in the platform-specific configuration files).

In principle the timer tick is a kernel implementation detail that should be irrelevant for applications. In practice, besides consuming energy each time it is handled, its granularity influences in a number of observable ways the temporal behaviour of applications.

For example, the real-time programmer should note that, while the L4 API expresses the IPC timeouts, timeslices, and `Sleep`() durations in microseconds, their actual accuracy depends on the tick period. A timeslice of 2000 μs lasts 2 ms on SPARC, PowerPC64, MIPS, and IA-64, nearly 3 ms on Alpha, nearly 4 ms on IA-32, AMD64, and PowerPC32, and finally 10 ms on StrongARM (but 5 ms in L4-embedded running on XScale). Similarly, the resolution of `SystemClock`() is equal to the tick period (1–10 ms) on most architectures, except for IA-32, where it is based on the time-stamp counter (TSC) register that increments with CPU clock pulses. Section 4.5 discusses other consequences.

```
void scheduler_t :: handle_timer_interrupt(){
...
 /* Check for not infinite timeslice and expired */
  if ((current->timeslice_length != 0) &&
   ((get_prio_queue(current)->current_timeslice
       -= get_timer_tick_length()) <= 0))
  {
   // We have end-of-timeslice.
   end_of_timeslice (current);
  }
...
```

Algorithm 1: L4 kernel/src/api/v4/schedule.cc.

Timing precision is an issue common to most operating systems and programming languages, as timer tick resolution used to be "good enough" for most time-based operating systems functions, but clearly is not for real-time and multimedia applications. In the case of L4Ka, a precise implementation would simply reprogram the timer for the earliest timeout or end-of-timeslice, or read it when providing the current time. However, if the timer I/O registers are located outside the CPU core, accessing them is a costly operation that would have to be performed in the IPC path each time a thread blocks with a timeout shorter than the current one (recent IA-32 processors have an on-core timer which is fast to access, but it is disabled when they are put in deeper sleep modes).

L4-embedded avoids most of these issues by removing support for IPC timeouts and the `SystemClock()` syscall from the kernel, and leaving the implementation of precise timing services to user level. This also makes the kernel faster by reducing the amount of work done in the IPC path and on each tick. Timer ticks consume energy, thus will likely be removed in future versions of L4-embedded, or made programmable based on the timeslice. Linux is recently evolving in the same direction [32]. Finally, malicious code can exploit easily-predictable timer ticks [33].

4.2. IPC and priority-driven scheduling

Being synchronous, IPC causes priority inversion in real-time applications programmed incorrectly, as described in the following scenario. A high-priority thread A performs IPC to a lower-priority thread B, but B is busy, so A blocks waiting for it to partner in IPC. Before B can perform the IPC that unblocks A, a third thread C with priority between A and B becomes ready, preempts B and runs. As the progress of A is impeded by C, which runs in its place despite having a lower priority, this is a case of *priority inversion*. Since priority inversion is a classic real-time bug, RTOSes contain special provisions to alleviate its effects [34]. Among them are priority inheritance (PI) and priority ceiling (PC), both discussed in detail by Liu [35]; note that the praise of PI is not unanimous: Yodaiken [36] discusses some cons.

The L4 research community investigated various alternatives to support PI. A naïve implementation would extend IPC and scheduling mechanisms to track temporary dependencies established during blocking IPCs from higher- to lower-priority threads, shuffle priorities accordingly, resume execution, and restore them once IPC completes. Since an L4-based system executes thousands of IPCs per second, the introduction of systematic support for PI would also impose a fixed cost on nonreal-time threads, possibly leading to a significant impact on overall system performance. Fiasco supports PI by extending L4's IPC and scheduling mechanisms to donate priorities through scheduling contexts that migrate between tasks that interact in IPC [28], but no quantitative evaluation of the overhead that this approach introduces is given.

Elphinstone [37] proposed an alternative solution based on statically structuring the threads and their priorities in such a way that a high-priority thread never performs a potentially blocking IPC with a lower-priority busy thread. While this solution fits better with the L4 static priority scheduler, it requires a special arrangement of threads and their priorities which may or may not be possible in all cases. To work properly in some corner cases this solution also requires keeping the messages on the incoming queue of a thread sorted by the static priority of their senders. Greenaway [38] investigated, besides scheduling optimisations, the costs of sorted IPC, finding that it is possible "...to implement priority-based IPC queueing with little effect on the performance of existing workloads."

A better solution to the problem of priority inversion is to encapsulate the code of each critical section in a server thread, and run it at the priority of the highest thread which may call it. Caveats for this solution are ordering of incoming calls to the server thread and some of the issues discussed in Section 4.4, but overall they require only a fraction of the cost of implementing PI.

4.3. Scheduler

The main issue with the L4 scheduler is that it is hardwired both in the specification and in the implementation. While it is fine for most applications, sometimes it might be convenient to perform scheduling decisions at the user level [39], feed the scheduler with application hints, or replace it with

different ones, for example, deadline-driven or time-driven. Currently the API does not support any of them.

Yet, the basic idea of microkernels is to provide applications with mechanisms and abstractions which are sufficiently expressive to build the required functionality at the user level. Is it therefore possible, modulo the priority inheritance issues discussed in Section 4.2, to perform priority-based real-time scheduling only relying on the standard L4 scheduler? Yes, but only if two optimisations common across most L4 microkernel implementations are taken into consideration: the short-circuiting of the scheduler by the IPC path, and the simplistic implementation of timeslice donation. Both are discussed in the next two sections.

4.4. IPC and scheduling policies

L4 invokes the standard scheduler to determine which thread to run next when, for example, the current thread performs a yield with the `ThreadSwitch(nilthread)` syscall, exhausts its timeslice, or blocks in the IPC path waiting for a busy partner. But a scheduling decision is also required when the partner is ready, and as a result at the end of the IPC more than one thread can run. Which thread should be chosen? A straightforward implementation would just change the state of the threads to runnable, move them to the ready list, and invoke the scheduler. The problem with this is that it incurs a significant cost along the IPC critical path.

L4 minimises the amount of work done in the IPC path with two complementary optimisations. First, the IPC path makes scheduling decisions without running the scheduler. Typically it switches directly to one of the ready threads according to policies that possibly, but not necessarily, take their priorities into account. Second, it marks as non-runnable a thread that blocks in IPC, but defers its removal from the ready list to save time. The assumption is that it will soon resume, woken up by an IPC from its partner. When the scheduler eventually runs and searches the ready list for the highest-priority runnable thread, it also moves any blocked thread it encounters into the waiting queue. The first optimisation is called *direct process switch*, the second *lazy scheduling*; Liedke [17] provides more details.

Lazy scheduling just makes some queue operations faster. Except for some pathological cases analysed by Greenaway [38], lazy scheduling has only second-order effects on real-time behaviour, and as such we will not discuss it further. Direct process switch, instead, has a significant influence on scheduling of priority-based real-time threads, but since it is seen primarily as an optimisation to avoid running the scheduler, the actual policies are sparsely documented, and missing from the L4 specification. We have therefore analysed the different policies employed in L4-embedded and L4Ka, reconstructed the motivation for their existence (in some cases the policy, the motivation, or both, changed as L4 evolved), and summarised our findings in Table 2 and the following paragraphs. In the descriptions, we adopt this convention: "A" is the *current* thread, that sends to the *dest* thread "B" and receives from the *from* thread "C." The policy applied depends on the type of IPC performed:

`Send()` at the end of a send-only IPC two threads can be run: the sender A or the receiver B; the current policy respects priorities and is cache-friendly, so it switches to B only if it has higher priority, otherwise continues with A. Since asynchronous notifications in L4-embedded are delivered via a send-only IPC, they follow the same policy: a waiting thread B runs only if it has higher priority than the notifier A, otherwise A continues.

`Receive()` thread A that performs a receive-only IPC from C results in a direct switch to C.

`Call()` client A which performs a call IPC to server B results in a direct switch of control to B.

`ReplyWait()` server A that responds to client B, and at the same time receives the next request from client C, results in a direct switch of control to B only if it has a strictly higher priority than C, otherwise control switches to C.

Each policy meets a different objective. In `Send()` it strives to follow the scheduler policy: the highest priority thread runs — in fact it only approximates it, as sometimes A may *not* be the highest-priority runnable thread (a consequence of timeslice donation: see Section 4.5). L4/MIPS [40] was a MIPS-specific version of L4 now superseded by L4Ka and L4-embedded. In its early versions the policy for `Send()` was set to continue with the receiver B to optimise a specific OS design pattern used at the time; in later versions the policy changed to always continue with the sender A to avoid priority inversion.

In other cases, the policies at the two sides of the IPC cooperate to favour brief IPC-based thread interactions over the standard thread scheduling by running the ready IPC partner on the timeslice of the current thread (also for this see Section 4.5).

Complex behaviour

Complex behaviour can emerge from these policies and their interaction. As the IPC path copies the message from sender to receiver in the final part of the send phase, when B receives from an already blocked A, the IPC will first switch to A's context in the kernel. However, once it has copied the message, the control may or may not immediately go back to B. In fact, because of the IPC policies, what will actually happen depends on the type of IPC A is performing (send-only, or send+receive), which of its partners are ready, and their priorities.

A debate that periodically resurfaces in the L4 community revolves around the policy used for the `ReplyWait()` IPC (actually the policy applies to any IPC with a send phase followed by a receive phase, of which `ReplyWait()` is a case with special arguments). If both B and C can run at the end of the IPC, and they have the same priority, the current policy arbitrarily privileges C. One effect of this policy is that a loaded server, although it keeps servicing requests, it limits the progress of the clients who were served and could resume execution. A number of alternative solutions which

Table 2: Scheduling policies in general-purpose L4 microkernels (∗ = see text).

Situation	When/where applied	Scheduling policy	switch_to(...)
ThreadSwitch (to)	application syscall	timeslice donation	to
ThreadSwitch (*nilthread*)	application syscall	scheduler	(highest pri. ready)
End of timeslice (typically 10 ms)	timer tick handler runs scheduler	scheduler	(highest pri. ready)
send(dest) blocks (no partner)	ipc send phase runs scheduler	scheduler	(highest pri. ready)
recv(from) blocks (no partner)	ipc recv phase runs scheduler	scheduler	(highest pri. ready)
send(dest) [Send()]	ipc send phase	direct process switch	maxpri(current, dest)
send(dest) [Send()] L4/MIPS	ipc send phase	timeslice donation	dest
send(dest) [Send()] L4/MIPS*	ipc send phase	(arbitrary)	current
recv(from) [Receive()]	ipc recv phase	timeslice donation	from
send(dest)+recv(dest) [Call()]	ipc send phase	timeslice donation	dest
send(dest)+recv(*anythread*) [ReplyWait()]	ipc recv phase	direct process switch	maxpri (dest,*anythread*)*
send(dest)+recv(from)	ipc recv phase	direct process switch	maxpri (dest, from)
Kernel interrupt path	handle_interrupt()	direct process switch	maxpri(current, handler)
Kernel interrupt path	handle_interrupt()*	timeslice donation	handler
Kernel interrupt path	irq_thread() completes Send()	timeslice donation	handler
Kernel interrupt path	irq_thread(), irq after Receive()	(as handle_interrupt())	(as handle_interrupt())
Kernel interrupt path L4-embedded	irq_thread(), no irq after Receive()	scheduler	(highest pri. ready)
Kernel interrupt path L4Ka	irq_thread(), no irq after Receive()	direct process switch	idle_thread

meet different requirements are under evaluation to be implemented in the next versions of L4-embedded.

Temporary priority inversion

In the `Receive`() and `Call`() cases, if A has higher priority than C, the threads with intermediate priority between A and C will not run until C blocks, or ends its timeslice. Similarly, in the `ReplyWait`() case, if A has higher priority than the thread that runs (either B or C, say X), other threads with intermediate priority between them will not run until X blocks, or ends its timeslice. In all cases, if the intermediate threads have a chance to run before IPC returns control to A, they generate temporary priority inversion for A (this is the same real-time application bug discussed in Section 4.2).

Direct switch in QNX

Notably, also the real-time OS QNX Neutrino performs a direct switch in synchronous IPCs when data transfer is involved [41]:

> Synchronous message passing
>
> This inherent blocking synchronises the execution of the sending thread, since the act of requesting that the data be sent also causes the sending thread to be blocked and the receiving thread to be scheduled for execution. This happens without requiring explicit work by the kernel to determine which thread to run next (as would be the case with most other forms of IPC). Execution and data move directly from one context to another.

IPC fastpath

Another optimisation of the L4 IPC is the *fastpath*, a hand-optimised, architecture-specific version of the IPC path which can very quickly perform the simplest and most common IPCs: transfer untyped data in registers to a specific thread that is ready to receive (there are additional requirements to fulfill: for more details, Nourai [42] discusses in depth a fastpath for the MIPS64 architecture). More complex IPCs are routed to the standard IPC path (also called the *slowpath*) which handles all the cases and is written in C. The fastpath/slowpath combination does not affect real-time scheduling, except for making most of the IPCs faster (more on this in Section 4.6). However, for reasons of scheduling consistency, it is important that if the fastpath performs a scheduling decision, then it replicates the same policies employed in the slowpath discussed above and shown in Table 2.

4.5. Timeslice donation

An L4 thread can donate the rest of its timeslice to another thread, performing the so-called *timeslice donation* [43]. The thread receiving the donation (recipient) runs briefly: if it does not block earlier, it runs ideally until the donor timeslice ends. Then the scheduler runs and applies the standard scheduling policy that may preempt the recipient and, for example, run another thread of intermediate priority between it and the donor which was ready to run since before the donation.

L4 timeslice donations can be explicit or implicit. Explicit timeslice donations are performed by applications with the `ThreadSwitch(to_tid)` syscall. They were initially intended by Liedtke to support user-level schedulers, but never used for that purpose. Another use is in mutexes

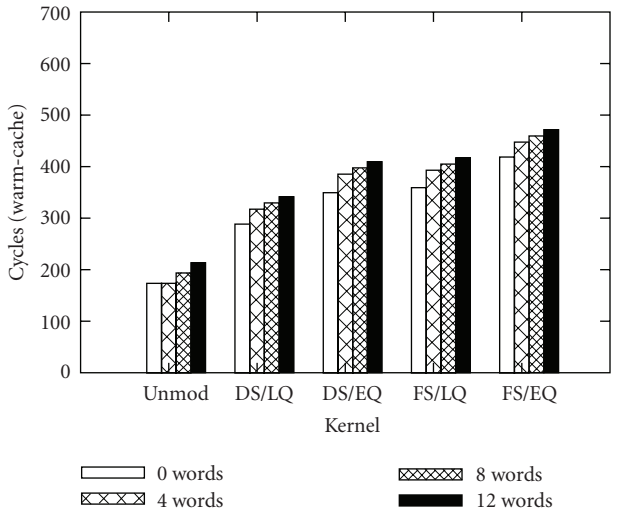


Figure 1: Raw IPC costs versus optimisations.

that — when contended — explicitly donate timeslices to their holder to speed-up the release of the mutex. Implicit timeslice donations happen in the kernel when the IPC path (or the interrupt path, see Section 4.7) transfers control to a thread that is ready to rendezvous. Note, however, that although implicit timeslice donation and direct process switch conflate in IPC, they have very different purposes. Direct process switch optimises scheduling in the IPC critical path. Timeslice donation favours threads interacting via IPC over standard scheduling. Table 2 summarises the instances of timeslice donation found in L4Ka and L4-embedded.

This is the theory. In practice, both in L4Ka and L4-embedded, a timeslice donation will *not* result in the recipient running for the rest of the donor timeslice. Rather, it will run *at least* until the next timer tick, and *at most* for *its own* timeslice, before it is preempted and normal scheduling is restored. The actual timeslice of the donor (including a timeslice of ∞) is not considered at all in determining how long the recipient runs.

This manifest deviation from what is stated both in the L4Ka and L4-embedded specifications [19] (and implied by the established term "timeslice donation") is due to a simplistic implementation of timeslice accounting. In fact, as discussed in Section 4.1 and shown in Algorithm 1, the scheduler function called by the timer tick handler simply decrements the timeslice of the current thread. It neither keeps track of the donation it may have received, nor does it propagate them in case donations are nested. In other words, what currently happens upon timeslice donation in L4Ka and L4-embedded is better characterised as a *limited timer tick donation*. The current terminology could be explained by earlier L4 versions which had timeslices and timerticks of coinciding lengths. Fiasco correctly donates timeslices at the price of a complex implementation [28] that we cannot discuss here for space reasons. Finally, Liedtke [44] argued that kernel fine-grained time measurement can be cheap.

The main consequence of timeslice donation is the temporary change of scheduling semantics (i.e., priorities are temporarily disregarded). The other consequences depend on the relative length of donor timeslices and timer tick. If both threads have a normal timeslice and the timer tick is set to the same value, the net effect is just about the same. If the timer tick is shorter than the donor timeslice, what gets donated is statistically much less, and definitely platform-dependent (see Table 1). The different lengths of the donations on different platforms can resonate with particular durations of computations, and result in occasional large differences in performance which are difficult to explain. For example, the performance of I/O devices (that may deliver time-sensitive data, e.g., multimedia) decreases dramatically if the handlers of their interrupts are preempted before finishing and are resumed after a few timeslices. Whether this will happen or not can depend on the duration of a donation from a higher priority interrupt dispatcher thread. Different lengths of the donations can also conceal or reveal race conditions and priority inversions caused by IPC (see Section 4.4).

4.6. IPC performance versus scheduling predictability

As discussed in Sections 4.4 and 4.5, general-purpose L4 microkernels contain optimisations that complicate priority-driven real-time scheduling. A natural question arises: how much performance is gained by these optimisations? Would it make sense to remove these optimisations in favour of priority-preserving scheduling? Elphinstone et al. [45], as a follow-up to [46] (subsumed by this paper), investigated the performance of L4-embedded (version 1.3.0) when both the direct switch (DS) and lazy scheduling (LQ, lazy queue manipulation) optimisations are removed, thus yielding a kernel which schedules threads strictly following their priorities. For space reasons, here we briefly report the findings, inviting the reader to refer to the paper for the rest of the details. Benchmarks have been run on an Intel XScale (ARM) PXA 255 CPU at 400 MHz.

Figure 1 shows the results of ping-pong, a tight loop between a client thread and server thread which exchange a fixed-length message. Unmod is the standard L4-embedded kernel with all the optimisations enabled, including the fastpath; the DS/LQ kernel has the same optimisations, except that, as the experimental scheduling framework, it lacks a fastpath implementation, in this and the subsequent kernels all IPCs are routed through the slowpath; the DS/EQ kernel performs direct switch and *eager* queuing (i.e., it disables lazy queuing). The FS/LQ and FS/EQ kernels perform *full scheduling* (i.e., *respect* priorities in IPC), and lazy and eager queuing, respectively. Application-level performance has been evaluated using the Re-aim benchmark suite run in a Wombat, Iguana/L4 system (see the paper for the full Re-Aim results and their analysis).

Apparently, the IPC "de-optimisation" gains scheduling predictability but reduces the raw IPC performance. However, its impact at the application level is limited. In fact, it has been found that "...the performance gains [due to the two optimisations] are modest. As expected, the overhead of IPC depends on its frequency. Removing the optimisations reduced [Re-aim] system throughput by 2.5% on average, 5%

in the worst case. Thus, the case for including the optimisations at the expense of real-time predictability is weak for the cases we examined. For much higher IPC rate applications, it might still be worthwhile." Summarising [45], it is possible to have a real-time friendly, general-purpose L4 microkernel without the issues caused by priority-unaware scheduling in the IPC path discussed in Section 4.4, at the cost of a moderate loss of IPC performance. Based on these findings, scheduling in successors of L4-embedded will be revised.

4.7. Interrupts

In general, the causes of interrupt-related glitches are the most problematic to find and are the most costly to solve. Some of them result from subtle interactions between how and when the hardware architecture generates interrupt requests and how and when the kernel or a device driver decides to mask, unmask or handle them. For these reasons, in the following paragraphs we first briefly summarise the aspects of interrupts critical for real-time systems. Then we show how they influence real-time systems architectures. Finally, we discuss the ways in which L4Ka and L4-embedded manage interrupts and their implications for real-time systems design.

Interrupts and real-time

In a real-time system, interrupts have two critical roles. First, when triggered by timers, they mark the passage of real-time and specific instants when time-critical operations should be started or stopped. Second, when triggered by peripherals or sensors in the environment, they inform the CPU of asynchronous events that require immediate consideration for the correct functioning of the system. Delays in interrupt handling can lead to jitter in time-based operations, missed deadlines, and the lateness or loss of time-sensitive data.

Unfortunately, in many general-purpose systems (e.g., Linux) both drivers and the kernel itself can directly or indirectly disable interrupts (or just pre-emption, which has a similar effect on time-sensitive applications) at unpredictable times, and for arbitrarily long times. Interrupts are disabled not only to maintain the consistency of shared data structures, but also to avoid deadlocks when taking spin locks and to avoid unbounded priority inversions in critical sections. Code that manipulates hardware registers according to strictly timed protocols should disable interrupts to avoid interferences.

Interrupts and system architecture

In Linux, interrupts and device drivers can easily interfere with real-time applications. A radical solution to this problem is to interpose between the kernel and the hardware a layer of privileged software that manages interrupts, timers and scheduling. This layer, called "real-time executive" in Figure 2, can range from a interrupt handler to a full-fledged RTOS (see [47–50], among others) and typically provides a real-time API to run real-time applications at a priority higher than the Linux kernel, which runs, de-privileged, as a low-priority thread. For obvious reasons, this is known as the *dual-kernel* approach. A disadvantage of some of these earlier real-time Linux approaches like RT-Linux and RTAI is that real-time applications seem (documentation is never very clear) to share their protection domain and privileges among themselves, with the real-time executive, or with Linux and its device drivers, leading to a complex and vulnerable system exposed to bugs in any of these subsystems.

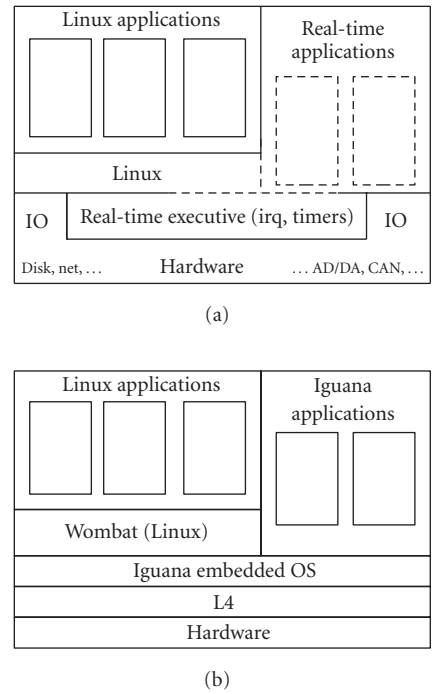


Figure 2: (a) A typical real-time Linux system. (b) Wombat, Iguana OS, and L4.

L4-embedded, combined with the Iguana [51] embedded OS and Wombat [29] (virtualised Linux for Iguana) leads to a similar architecture (Figure 2(b)) but with full memory protection and control of privileges for all components enforced by the embedded OS and the microkernel itself. Although memory protection does not come for free, it has been already proven that a microkernel-based RTOS can support real-time Linux applications in separate address spaces at costs, in terms of interrupt delays and jitter, comparable to those of blocked interrupts and caches, costs that seem to be accepted by designers [27]. However the situation in the Linux field is improving. XtratuM overcomes the protection issue by providing instead "a memory map per OS, enabling memory isolation among different OSes" [48], thus an approach more similar to Figure 2(b). In a different effort, known as the *single-kernel* approach, the standard Linux kernel is modified to improve its real-time capabilities [52].

L4 interrupts

As introduced in Section 3.6, L4 converts all interrupts (but the timer tick) into IPC messages, which are sent to a user-level thread which will handle them. The internal interrupt

path comprises three routines: the generic `irq_thread`(), the generic `handle_interrupt`(), and a low-level, platform-specific handler that manages the IC.

When L4 receives an interrupt, the platform-specific handler disables it in the IC and calls `handle_interrupt`(), which creates an interrupt IPC message and, if the user-level handler is not waiting for it, enqueues the message in the handler's message queue, marks the in-kernel interrupt thread as runnable (we will see its role shortly), and returns to the current (interrupted) thread. If instead the handler is waiting, and the current thread is an interrupt kernel thread or the idle thread, `handle_interrupt`() switches directly to the handler, performing a *timeslice donation*. Finally, if the handler is waiting and the current thread was neither an interrupt thread nor the idle thread, it does *direct process switch* and switches to the handler only if it has higher priority than the current thread, otherwise it moves the interrupt thread in the ready queue, and switches to the current thread, like the IPC path does for a `Send`() (see Sections 4.4 and 4.5).

Each in-kernel interrupt thread (one for each IRQ line) executes `irq_thread`(), a simple endless loop that performs two actions in sequence. It delivers a pending message to a user-level interrupt handler which became ready to receive, and then blocks waiting to receive its reply when it processed the interrupt. When it arrives, `irq_thread`() re-enables the interrupt in the IC, marks itself halted and, if a new interrupt is pending, calls `handle_interrupt`() to deliver it (which will suspend the interrupt thread and switch to the handler, if it is waiting). Finally, it yields to another ready thread (L4-embedded) or to the idle thread (L4Ka). In other words, the interrupt path *mimics* a `Call`() IPC. The bottom part of Table 2 summarises the scheduling actions taken by the interrupt paths of the L4Ka and L4-embedded microkernels.

Advantages

In L4-based systems, only the microkernel has the necessary privileges to enable and disable interrupts globally in the CPU and selectively in the interrupt controller. All user-level code, including drivers and handlers, has control only over the interrupts it registered for, and can disable them only either by simply not replying to an interrupt IPC message, or by deregistering altogether, but cannot mask any other interrupt or all of them globally (except by entering the kernel, which, in some implementations, e.g., L4Ka, currently disables interrupts).

An important consequence of these facts is that L4-based real-time systems do not need to trust drivers and handlers time-wise, since they cannot programmatically disable all interrupts or preemption. More importantly, at the user-level, mutual exclusion between a device driver and its interrupt handler can be done using concurrency-control mechanisms that do not disable preemption or interrupts like spin locks must do in the kernel. Therefore, user-level driver-handler synchronisations only have a *local* effect, and thus neither unpredictably perturb the timeliness of other components of the system, nor contribute to its overall latency.

Another L4 advantage is the unification of the scheduling of applications and interrupt handlers. Interrupts can negatively impact the timeliness of a system in different ways, but at least for the most common ones, L4 allows simple solutions which we discuss in the following.

A typical issue is the long-running handler, either because it is malicious, or simply badly written as is often the case. Even if it cannot disable interrupts, it can still starve the system by running as the highest-priority ready thread. A simple remedy to bound its effects is to have it scheduled at the same priority as other threads, if necessary tune its timeslice, and rely on L4's round-robin scheduling policy which ensures global progress (setting its priority lower than other threads would unfairly starve the device).

A second issue is that critical real-time threads must not be delayed by less important interrupts. In L4, low-priority handlers cannot defer higher priority threads by more than the time spent by the kernel to receive each user-registered interrupt once and queue the IPC message. Also the periodic timer tick interrupt contributes to delaying the thread, but for a bounded and reasonably small amount of time.

Consider finally a third common issue: thrashing caused by interrupt overload, where the CPU spends all its time handling interrupts and nothing else. L4 prevents this by design, since after an interrupt has been handled, it is the handler which decides if and when to handle the next pending interrupt, not the microkernel. In this case, even if the handler runs for too long because it has no provision for overload, it can still be throttled via scheduling as discussed above.

Notably, in all these cases it is not necessary to trust drivers and handlers to guarantee that interrupts will not disrupt in one way or another the timeliness of the system. In summary, running at user-level makes interrupt handlers and device drivers in L4-based systems *positively constrained*, in the sense that their behaviour — as opposed to the in-kernel ones — cannot affect the OS and applications beyond what is allowed by the protection and scheduling policies set for the system.

Disadvantages

Interrupt handling in L4 has two drawbacks: latency and, in some implementations, a nonpreemptable kernel. The interrupt latency is defined as the time between when the interrupt is asserted by the peripheral and the first instruction of its handler is executed. The latency is slightly higher for L4 user-level handlers than for traditional in-kernel ones since even in the best-case scenario more code runs and an additional context switch is performed. Besides, even if thoroughly optimised, L4 (like Linux and many RTOSes) does not provide specific real-time guarantees (i.e., a firm upper bound in cycles), neither for its API in general, nor for its IPC in particular. That said, interrupt latency in L4 is bounded and "smaller" than normal IPC (as the interrupt IPC path is much simpler than the full IPC path).

Both kernel preemptability and latency in L4-embedded are currently considered by researchers and developers. The current implementation of L4Ka disables interrupts while in kernel mode. Since in the vast majority of cases the time spent in the kernel is very short, especially if compared to monolithic kernels, and preempting the kernel has about

the same cost as a fast system call, L4-embedded developers maintain that the additional complexity of making it *completely* preemptable is not justified.

However, the L4Ka kernel takes a very long time to "unmap" a set of deeply nested address spaces, and this increases both the interrupt and the preemption worst-case latencies. For this reason, in L4-embedded the virtual memory system has been reworked to move part of the memory management to user level, and introduce preemption points where interrupts are enabled in long-running kernel operations. Interestingly, also the Fiasco kernel implementation has a performance/preemptability tradeoff. It is located in the registers-only IPC path, which in fact runs with interrupts disabled and has explicit preemption points [53].

A notable advantage of a nonpreemptable kernel, or one with very few preemption points, is that it can be formally verified to be correct [20].

With respect to latency, Mehnert et al. [27] investigate the worst-case interrupt latency in Fiasco. An ongoing research project [54] aims, among other things, at precisely characterising the L4-embedded interrupt latency via a detailed timing analysis of the kernel. Finally, to give an idea of the interrupt latencies achieved with L4-embedded, work on the interrupt fast path (an optimised version of the interrupt IPC delivery path) for a MIPS 2000/3000-like superscalar CPU reduced the interrupt latency by a factor of 5, from 425 cycles for the original C version down to 79 cycles for the hand-optimised assembly version.

User-level real-time reflective scheduling

The L4 interrupt subsystem provides a solid foundation for user-level real-time scheduling. For example, we exploited the features and optimisations of L4-embedded described in this paper to realise a *reflective* scheduler, a time-driven scheduler based on reflective abstractions and mechanisms that allow a system to perform *temporal reflection*, that is to explicitly model and control its temporal behaviour as an object of the system itself [55].

Since the reflective scheduler is time-driven, and to synchronise uses mutexes and asynchronous notifications, its correctness is not affected by direct process switch optimisation in the IPC path. Our implementation does not require any changes to the microkernel itself, and thus does not impact on its performance. On a 400 MHz XScale CPU it performs time-critical operations with both accuracy and precision of the order of microseconds [56]. To achieve this performance we implemented the reflective scheduler using a user-level thread always ready to receive the timer IPC, and with a priority higher than normal applications. In this configuration, L4 can receive the interrupt and perform a direct process switch from kernel mode to the user level thread without running the kernel scheduler. As a consequence, the latency of the reflective scheduler is low and roughly constant, and can be compensated. A simple closed-loop feedback calibration reduces it from 52–57 μs to $-4-+3\,\mu$s. For space reasons we cannot provide more details. Please find in Ruocco [56] the full source code of the scheduler and the real-time video analysis application we developed to evaluate it.

5. CONCLUSIONS

In this paper we discussed a number of aspects of L4 microkernels, namely L4-embedded and its general-purpose ancestor L4::Ka Pistachio, that are relevant for the design of real-time systems. In particular we showed why the optimisations performed in the IPC path complicate real-time scheduling. Fortunately, these optimisations can be removed with a small performance loss at application level, achieving real-time friendly scheduling in L4-embedded. Finally, we highlighted the advantages of the L4 interrupt subsystem for device drivers and real-time schedulers operating at the user level. Overall, although there is still work ahead, we believe that with few well-thought-out changes microkernels like L4-embedded can be used successfully as the basis of a significant class of real-time systems, and their implementation is facilitated by a number of design features unique to microkernels and the L4 family.

ACKNOWLEDGMENTS

The author wants to thank Hal Ashburner, Peter Chubb, Dhammika Elkaduwe, Kevin Elphinstone, Gernot Heiser, Robert Kaiser, Ihor Kuz, Adam Lackorzynski, Geoffrey Lee, Godfrey van der Linden, David Mirabito, Gabriel Parmer, Stefan Petters, Daniel Potts, Marco Ruocco, Leonid Ryzhyk, Carl van Schaik, Jan Stoess, Andri Toggenburger, Matthew Warton, Julia Weekes, and the anonymous reviewers. Their feedback improved both the content and style of this paper.

REFERENCES

[1] National ICT Australia, *NICTA L4-embedded kernel reference manual version N1*, October 2005, http://ertos.nicta.com.au/software/kenge/pistachio/latest/refman.pdf.

[2] L4Ka Team, "L4Ka: pistachio kernel," http://l4ka.org/projects/pistachio/.

[3] P. S. Langston, "Report on the workshop on micro-kernels and other kernel architectures," April 1992, http://www.langston.com/Papers/uk.pdf.

[4] J. Liedtke, "Towards real microkernels," *Communications of the ACM*, vol. 30, no. 9, pp. 70–77, 1996.

[5] D. B. Golub, R. W. Dean, A. Forin, and R. F. Rashid, "UNIX as an application program," in *Proceedings of the Summer USENIX*, pp. 87–59, Anaheim, California, USA, June 1990.

[6] J. Liedtke, "On μ-kernel construction," in *Proceedings of the 15th Symposium on Operating Systems Principles (SOSP '95)*, pp. 237–250, Copper Mountain, Colorado, USA, December 1995.

[7] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr., "Exokernel: an operating system architecture for application-level resource management," in *Proceedings of the 15th Symposium on Operating Systems Principles (SOSP '95)*, pp. 251–266, Copper Mountain, Colorado, USA, December 1995.

[8] I. M. Leslie, D. McAuley, R. Black, et al., "The design and implementation of an operating system to support distributed multimedia applications," *IEEE Journal on Selected Areas in Communications*, vol. 14, no. 7, pp. 1280–1296, 1996.

[9] D. R. Engler and M. F. Kaashoek, "Exterminate all operating system abstractions," in *Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HotOS '95)*, pp. 78–85, Orcas Island, Washington, USA, May 1995.

[10] J. Liedtke, "A persistent system in real use experience of the first 13 years," in *Proceedings of the 3rd International Workshop on Object Orientation in Operating Systems (IWOOOS '93)*, pp. 2–11, Asheville, North Carolina, USA, December 1993.

[11] D. Hildebrand, "An architectural overview of QNX," in *Proceedings of the Workshop on Micro-Kernels and Other Kernel Architectures*, pp. 113–126, Berkeley, California, USA, 1992.

[12] B. Leslie, P. Chubb, N. Fitzroy-Dale, et al., "User-level device drivers: achieved performance," *Journal of Computer Science and Technology*, vol. 20, no. 5, pp. 654–664, 2005.

[13] T. P. Baker, A. Wang, and M. J. Stanovich, "Fitting Linux device drivers into an analyzable scheduling framework," in *Proceedings of the 3rd Workshop on Operating System Platforms for Embedded Real-Time Applications*, Pisa, Italy, July 2007.

[14] D. Engler, *The Exokernel operating system architecture*, Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, Massachussets, USA, 1999.

[15] J. Kamada, M. Yuhara, and E. Ono, "User-level realtime scheduler exploiting kernel-level fixed priority scheduler," in *Proceedings of the International Symposium on Multimedia Systems*, Yokohama, Japan, March 1996.

[16] J. Liedtke, "μ-kernels must and can be small," in *Proceedings of the 15th International Workshop on Object Orientation in Operating Systems (IWOOOS '96)*, pp. 152–161, Seattle, Washington, USA, October 1996.

[17] J. Liedtke, "Improving IPC by kernel design," in *Proceedings of the 14th Symposium on Operating Systems Principles (SOSP '03)*, pp. 175–188, Asheville, North Carolina, USA, December 2003.

[18] G. Gray, M. Chapman, P. Chubb, D. Mosberger-Tang, and G. Heiser, "Itanium — a system implementor's tale," in *Proceedings of the USENIX Annual Technical Conference (ATEC '05)*, pp. 265–278, Anaheim, California, USA, April 2005.

[19] L4 headquarters, "L4 kernel reference manuals," June 2007, http://l4hq.org/docs/manuals/.

[20] G. Klein, M. Norrish, K. Elphinstone, and G. Heiser, "Verifying a high-performance micro-kernel," in *Proceedings of the 7th Annual High-Confidence Software and Systems Conference*, Baltimore, Maryland, USA, May 2007.

[21] R. Kaiser and S. Wagner, "Evolution of the PikeOS microkernel," in *Proceedings of the 1st International Workshop on Microkernels for Embedded Systems (MIKES '07)*, Sydney, Australia, March 2007.

[22] H. Härtig and M. Roitzsch, "Ten years of research on L4-based real-time systems," in *Proceedings of the 8th Real-Time Linux Workshop*, Lanzhou, China, 2006.

[23] M. Hohmuth, "The Fiasco kernel: requirements definition," Tech. Rep. TUD-FI98-12, Technische Universität Dresden, Dresden, Germany, December 1998.

[24] M. Hohmuth and H. Härtig, "Pragmatic nonblocking synchronization for real-time systems," in *Proceedings of the USENIX Annual Technical Conference*, pp. 217–230, Boston, Massachussets, USA, June 2001.

[25] H. Härtig, R. Baumgartl, M. Borriss, et al., "DROPS: OS support for distributed multimedia applications," in *Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*, pp. 203–209, Sintra, Portugal, September 1998.

[26] C. J. Hamann, J. Löser, L. Reuther, S. Schönberg, J. Wolter, and H. Härtig, "Quality-assuring scheduling — using stochastic behavior to improve resource utilization," in *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS '01)*, pp. 119–128, London, UK, December 2001.

[27] F. Mehnert, M. Hohmuth, and H. Härtig, "Cost and benefit of separate address spaces in real-time operating systems," in *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS '02)*, pp. 124–133, Austin, Texas, USA, December, 2002.

[28] U. Steinberg, J. Wolter, and H. Härtig, "Fast component interaction for real-time systems," in *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS '05)*, pp. 89–97, Palma de Mallorca, Spain, July 2005.

[29] B. Leslie, C. van Schaik, and G. Heiser, "Wombat: a portable user-mode Linux for embedded systems," in *Proceedings of the 6th Linux Conference*, Canberra, Australia, April 2005.

[30] ISO/IEC, *The POSIX 1003.1 Standard*, 1996.

[31] J. S. Shapiro, "Vulnerabilities in synchronous IPC designs," in *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 251–262, Oakland, California, USA, May 2003.

[32] S. Siddha, V. Pallipadi, and A. Van De Ven, "Getting maximum mileage out of tickless," in *Proceedings of the Ottawa Linux Symposium*, Ottawa, Canada, June 2007.

[33] D. Tsafrir, Y. Etsion, and D. G. Feitelson, "Secretly monopolizing the CPU without superuser privileges," in *Proceedings of the 16th USENIX Security Symposium*, Boston, Massachussets, USA, April 2007.

[34] Mike Jones, "What really happened on Mars Rover Pathfinder," *The Risks Digest*, vol. 19, no. 49, 1997, based on David Wilner's keynote address of 18th IEEE Real-Time Systems Symposium (RTSS '97), December, 1997, San Francisco, California, USA. http://research.microsoft.com/~mbj/Mars_Pathfinder/.

[35] J. W. S. Liu, *Real-Time Systems*, Prentice-Hall, Upper Saddle River, New Jersey, USA, 2000.

[36] V. Yodaiken, "Against priority inheritance," http://www.yodaiken.com/papers/inherit.pdf.

[37] K. Elphinstone, "Resources and priorities," in *Proceedings of the 2nd Workshop on Microkernels and Microkernel-Based Systems*, K. Elphinstone, Ed., Lake Louise, Alta, Canada, October 2001.

[38] D. Greenaway, "From 'real fast' to real-time: quantifying the effects of scheduling on IPC performance," B. Sc. thesis, School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia, 2007.

[39] J. Stoess, "Towards effective user-controlled scheduling for microkernel-based systems," *Operating Systems Review*, vol. 41, no. 4, pp. 59–68, 2007.

[40] G. Heiser, "Inside L4/MIPS anatomy of a high-performance microkernel," Tech. Rep. 2052, School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia, 2001.

[41] QNX Neutrino IPC, http://www.qnx.com/developers/docs/6.3.0SP3/neutrino/sys_arch/kernel.html#NTOIPC.

[42] A. Nourai, "A physically-addressed L4 kernel," B. Eng. thesis, School of Computer Science & Engineering, The University of New South Wales, Sydney, Australia, 2005, http://www.disy.cse.unsw.edu.au/.

[43] B. Ford and S. Susarla, "CPU inheritance scheduling," in *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI '96)*, pp. 91–105, Seattle, Washington, USA, October 1996.

[44] J. Liedtke, "A short note on cheap fine-grained time measurement," *Operating Systems Review*, vol. 30, no. 2, pp. 92–94, 1996.

[45] K. Elphinstone, D. Greenaway, and S. Ruocco, "Lazy queueing and direct process switch — merit or myths?" in *Proceedings of the 3rd Workshop on Operating System Platforms for Embedded Real-Time Applications*, Pisa, Italy, July 2007.

[46] S. Ruocco, "Real-time programming and L4 microkernels," in *Proceedings of the 2nd Workshop on Operating System Platforms for Embedded Real-Time Applications*, Dresden, Germany, July 2006.

[47] "Xenomai — implementing a RTOS emulation framework on GNU/Linux," 2004, http://www.xenomai.org/documentation/branches/v2.3.x/pdf/xenomai.pdf.

[48] M. Masmano, I. Ripoll, and C. Crespo, "An overview of the XtratuM nanokernel," in *Proceedings of the Workshop on Operating System Platforms for Embedded Real-Time Applications*, Palma de Mallorca, Spain, July 2005.

[49] Politecnico di Milano — Dipartimento di Ingegneria Aerospaziale, "RTAI the real-time application interface for Linux," http://www.rtai.org/.

[50] V. Yodaiken and M. Barabanov, "A real-time Linux," http://citeseer.ist.psu.edu/article/yodaiken97realtime.html.

[51] Iguana OS, http://www.ertos.nicta.com.au/iguana/.

[52] S. Rostedt and D. V. Hart, "Internals of the RT Patch," in *Proceedings of the Ottawa Linux Symposium*, Ottawa, Canada, January 2007.

[53] R. Reusner, "Implementierung eines echtzeit-IPC-Pfades mit unterbrechungspunkten für L4/Fiasco," Diplomarbeit Thesis, Fakultät Informatik, Technische Universität Dresden, Dresden, Germany, 2006.

[54] S. M. Petters, P. Zadarnowski, and G. Heiser, "Measurements or static analysis or both?" in *Proceedings of the 7th Workshop Worst-Case Execution-Time Analysis*, Pisa, Italy, July 2007.

[55] S. Ruocco, *Temporal reflection*, Ph.D. thesis, Università degli Studi di Milano, Milano, Italy, 2004.

[56] S. Ruocco, "User-level fine-grained adaptive real-time scheduling via temporal reflection," in *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS '06)*, pp. 246–256, Rio de Janeiro, Brazil, December 2006.

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.

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.

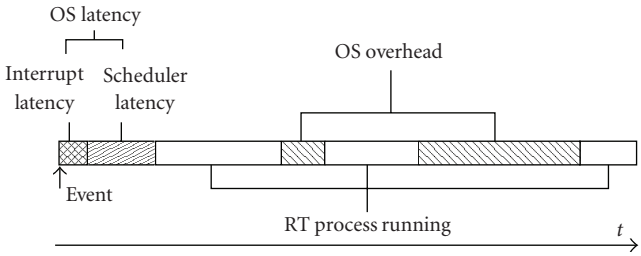


Figure 1: Jitter.

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.

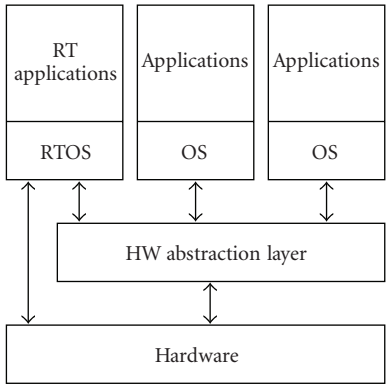


Figure 2: Horizontally partitioned operating system.

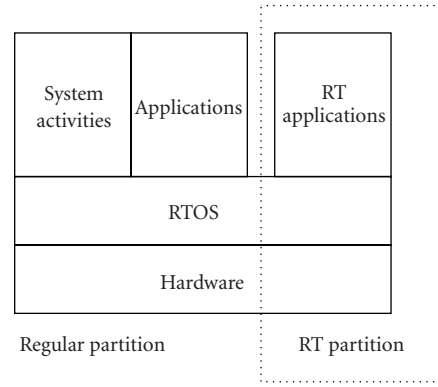


Figure 3: Vertically partitioned operating system.

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).

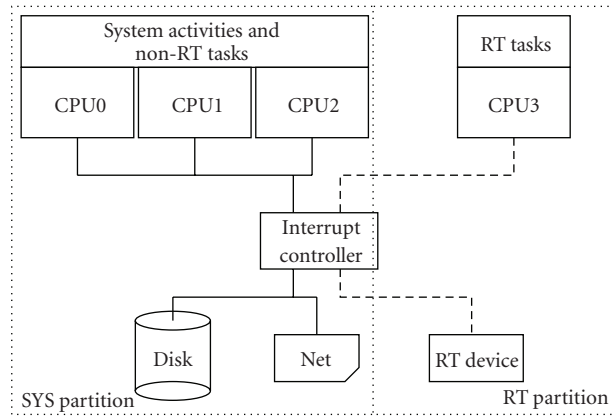


Figure 4: ASMP-Linux partitioning.

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.

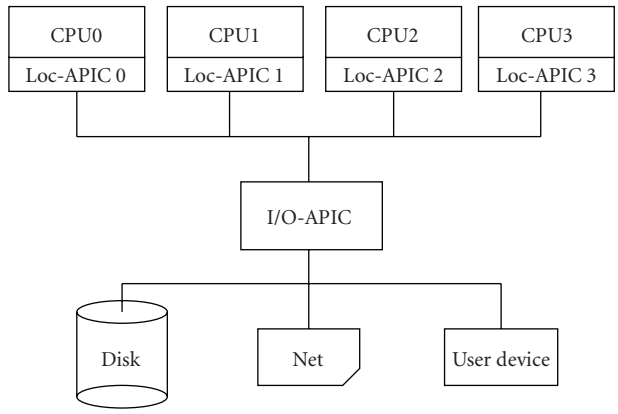


Figure 5: A SMP using Intel IO-APIC.

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].

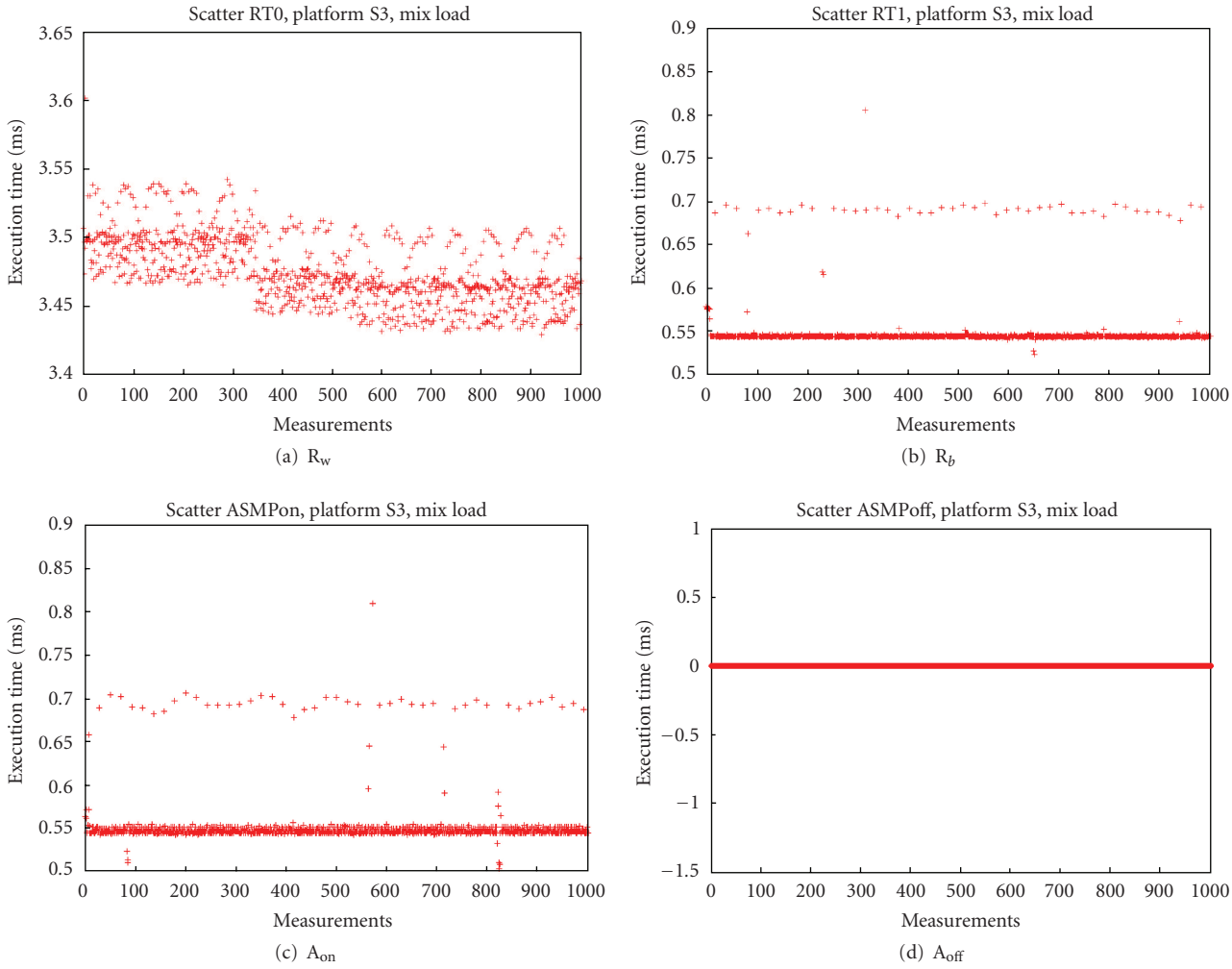


Figure 6: Scatter graphics for system S3, MIX workload.

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

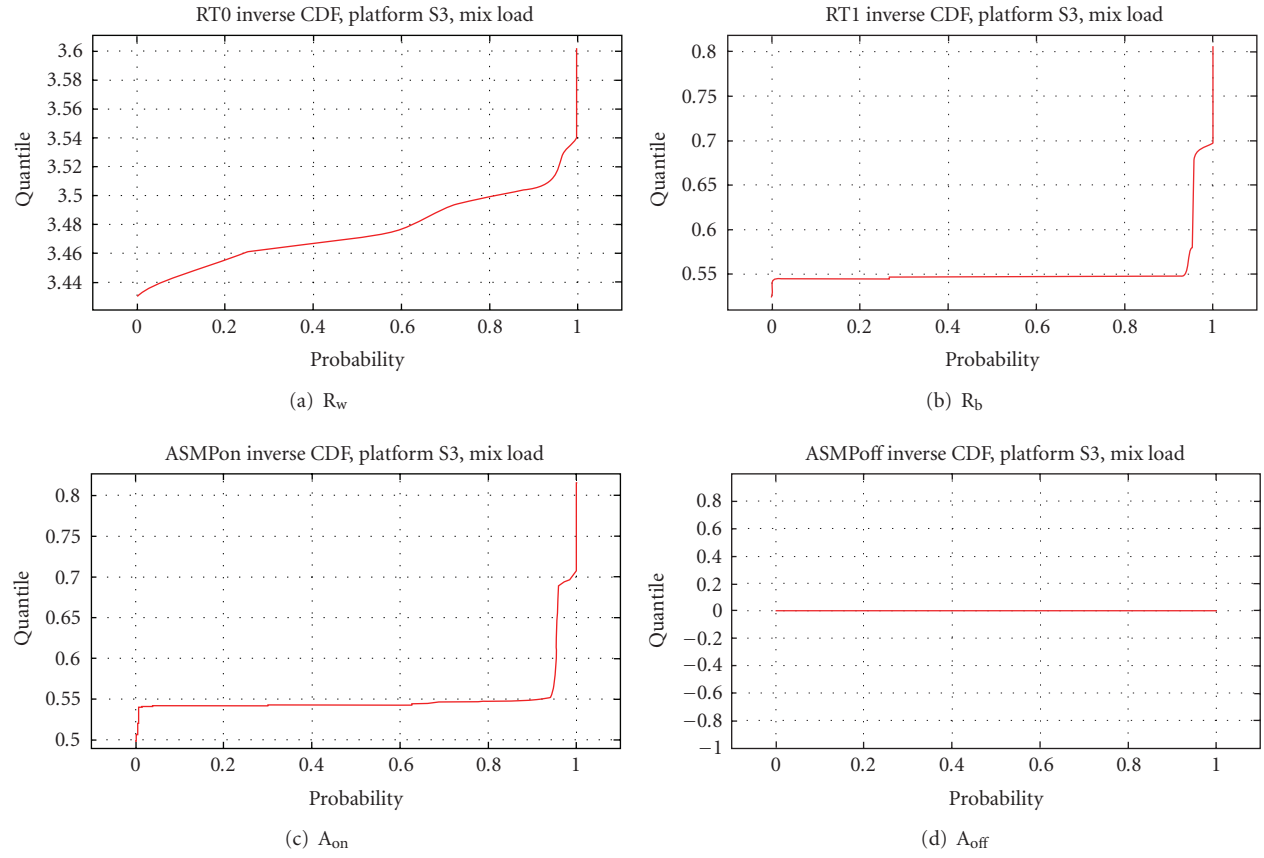


FIGURE 7: Inverse density functions for overhead on system S3, MIX workload.

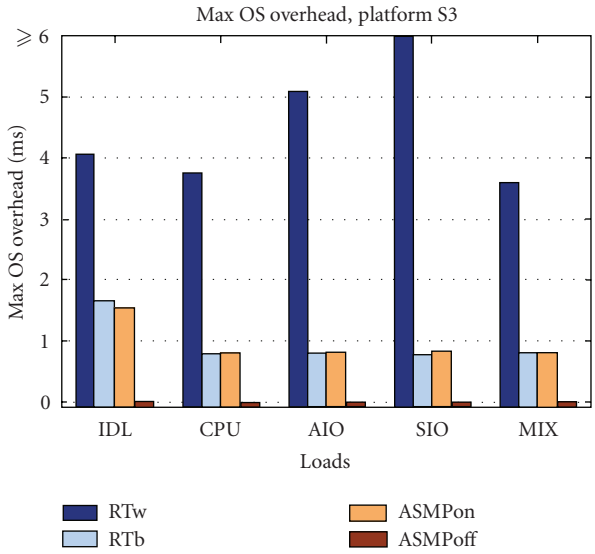


FIGURE 8: OS maximum overhead comparison.

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

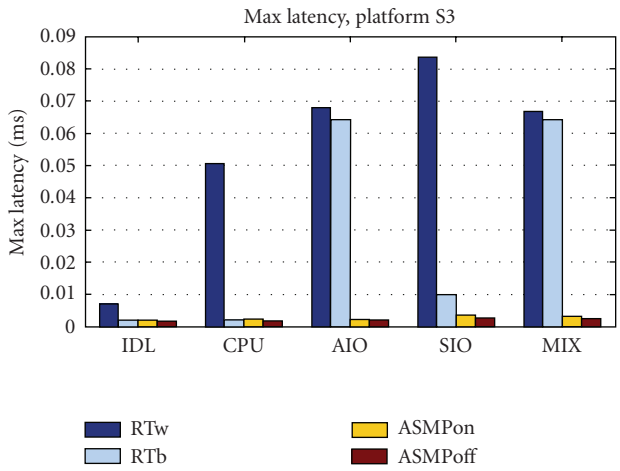


Figure 9: OS maximum latency comparison.

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.

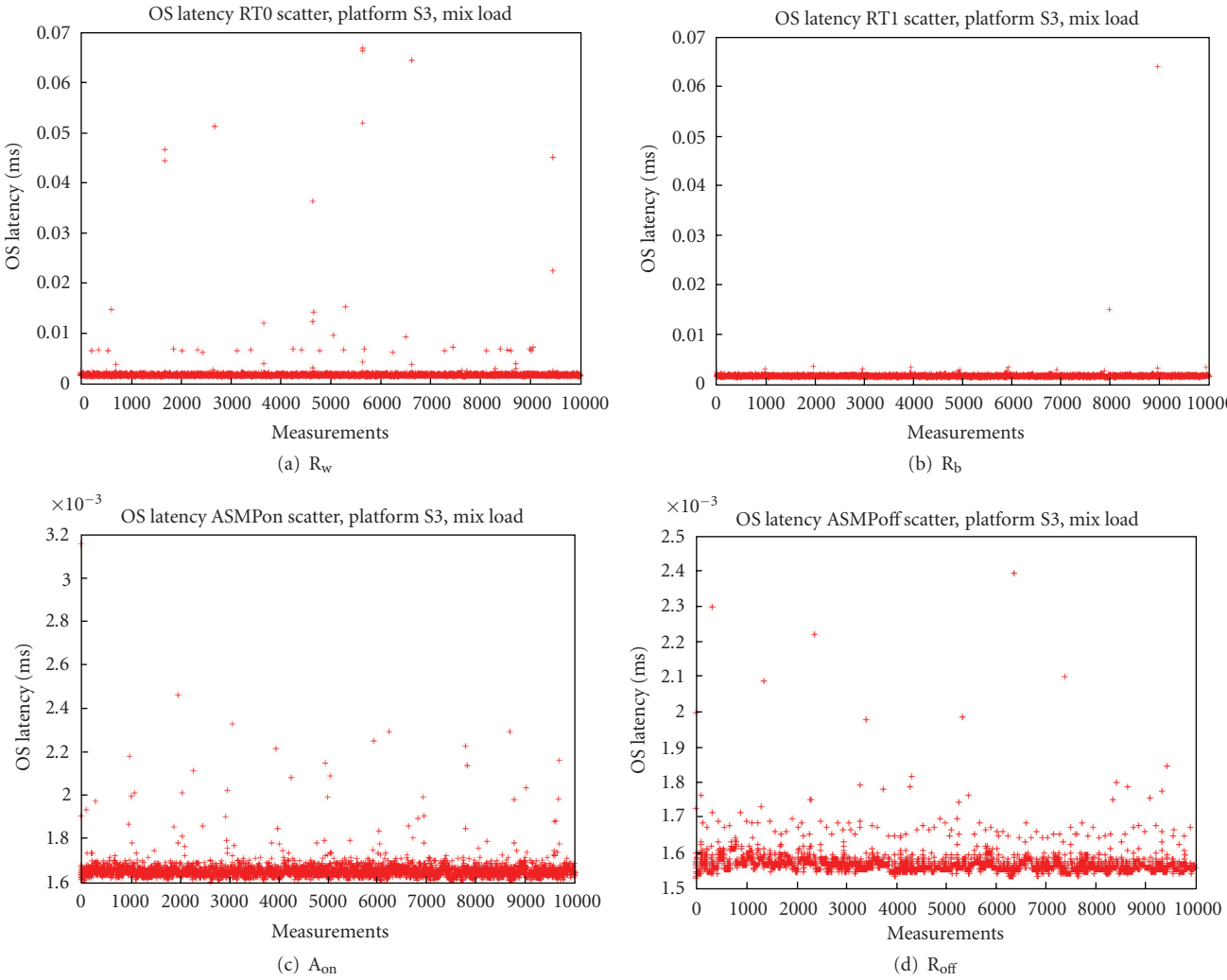


Figure 10: Scatter graphics for system S3, MIX workload.

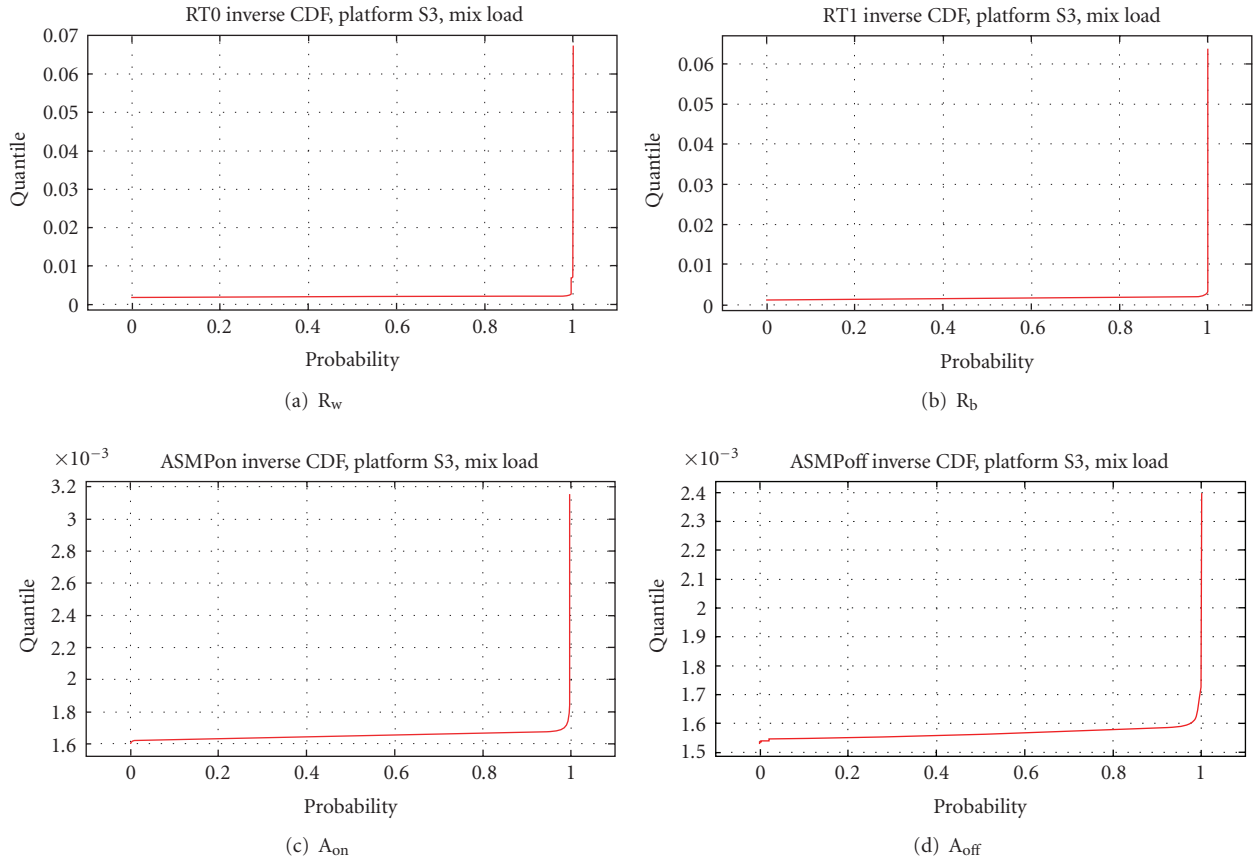
6.4. *Final consideration*

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/.

Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 250895, 20 pages
doi:10.1155/2008/250895

Research Article

Design and Performance Evaluation of an Adaptive Resource Management Framework for Distributed Real-Time and Embedded Systems

Nishanth Shankaran,[1] Nilabja Roy,[1] Douglas C. Schmidt,[1] Xenofon D. Koutsoukos,[1] Yingming Chen,[2] and Chenyang Lu[2]

[1] *The Electrical Engineering and Computer Science Department, Vanderbilt University, Nashville, TN 37235, USA*
[2] *Department of Computer Science and Engineering, Washington University, St. Louis, MO 63130, USA*

Correspondence should be addressed to Nishanth Shankaran, nshankar@dre.vanderbilt.edu

Received 8 February 2007; Revised 6 November 2007; Accepted 2 January 2008

Recommended by Michael Harbour

Achieving end-to-end quality of service (QoS) in distributed real-time embedded (DRE) systems require QoS support and enforcement from their underlying operating platforms that integrates many real-time capabilities, such as QoS-enabled network protocols, real-time operating system scheduling mechanisms and policies, and real-time middleware services. As standards-based quality of service (QoS) enabled component middleware automates integration and configuration activities, it is increasingly being used as a platform for developing open DRE systems that execute in environments where operational conditions, input workload, and resource availability cannot be characterized accurately a priori. Although QoS-enabled component middleware offers many desirable features, however, it historically lacked the ability to allocate resources efficiently and enable the system to adapt to fluctuations in input workload, resource availability, and operating conditions. This paper presents three contributions to research on adaptive resource management for component-based open DRE systems. First, we describe the structure and functionality of the resource allocation and control engine (RACE), which is an open-source adaptive resource management framework built atop standards-based QoS-enabled component middleware. Second, we demonstrate and evaluate the effectiveness of RACE in the context of a representative open DRE system: NASA's magnetospheric multiscale mission system. Third, we present an empirical evaluation of RACE's scalability as the number of nodes and applications in a DRE system grows. Our results show that RACE is a scalable adaptive resource management framework and yields a predictable and high-performance system, even in the face of changing operational conditions and input workload.

1. INTRODUCTION

Distributed real-time and embedded (DRE) systems form the core of many large scale mission-critical domains. In these systems, achieving end-to-end quality of service (QoS) requires integrating a range of real-time capabilities, such as QoS-enabled network protocols, real-time operating system scheduling mechanisms and policies, and real-time middleware services, across the system domain. Although existing research and solutions [1, 2] focus on improving the performance and QoS of individual capabilities of the system (such as operating system scheduling mechanism and policies), they are not sufficient for DRE systems as these systems require integrating a range of real-time capabilities across the system domain. Conventional QoS-enabled middleware technologies, such as real-time CORBA [3] and the real-time Java [4], have been used extensively as an operating platforms to build DRE systems as they support explicit configuration of QoS aspects (such as priority and threading models), and provide many desirable real-time features (such as priority propagation, scheduling services, and explicit binding of network connections).

QoS-enabled middleware technologies have traditionally focused on DRE systems that operate in *closed* environments where operating conditions, input workloads, and resource availability are known in advance and do not vary significantly at run-time. An example of a closed DRE system is an avionics mission computer [5], where the penalty of

not meeting a QoS requirement (such as deadline) can result in the failure of the entire system or mission. Conventional QoS-enabled middleware technologies are insufficient, however, for DRE systems that execute in *open* environments where operational conditions, input workload, and resource availability cannot be characterized accurately a priori. Examples of open DRE systems include shipboard computing environments [6], multisatellite missions [7]; and intelligence, surveillance, and reconnaissance missions [8].

Specifying and enforcing end-to-end QoS is an important and challenging issue for open systems DRE due to their unique characteristics, including (1) constraints in multiple resources (e.g., limited computing power and network bandwidth) and (2) highly fluctuating resource availability and input workload. At the heart of achieving end-to-end QoS are resource management techniques that enable open DRE systems to *adapt* to dynamic changes in resource availability and demand. In earlier work, we developed adaptive resource management *algorithms* (such as EUCON [9], DEUCON [10], HySUCON [11], and FMUF [12]) and *architectures*, such as HiDRA [13] based on control-theoretic techniques. We then developed FC-ORB [14], which is a QoS-enabled adaptive middleware that implements the EUCON algorithm to handle fluctuations in application workload and system resource availability.

A limitation with our prior work, however, is that it tightly coupled resource management algorithms within particular middleware platforms, which made it hard to enhance the algorithms without redeveloping significant portions of the middleware. For example, since the design and implementation of FC-ORB were closely tied to the EUCON adaptive resource management algorithm, significant modifications to the middleware were needed to support other resource management algorithms, such as DEUCON, HySUCON, or FMUF. Object-oriented frameworks have traditionally been used to factor out many reusable general-purpose and domain-specific services from DRE systems and applications [15]; however, to alleviate the tight coupling between resource management algorithms and middleware platforms and improve flexibility, this paper presents an *adaptive resource management framework* for open DRE systems. Contributions of this paper to the study of adaptive resource management solutions for open DRE systems include the following.

(i) *The design of a resource allocation and control engine (RACE)*, which is a fully customizable and configurable adaptive resource management framework for open DRE systems. RACE decouples adaptive resource management algorithms from the middleware implementation, thereby enabling the usage of various resource management algorithms without the need for redeveloping significant portions of the middleware. RACE can be configured to support a range of algorithms for adaptive resource management without requiring modifications to the underlying middleware. To enable the seamless integration of resource allocation and control algorithms into DRE systems, RACE enables the deployment and configuration of feedback control loops. RACE, therefore, complements theoretical research on adaptive resource management algorithms that provide a model and theoretical analysis of system performance.

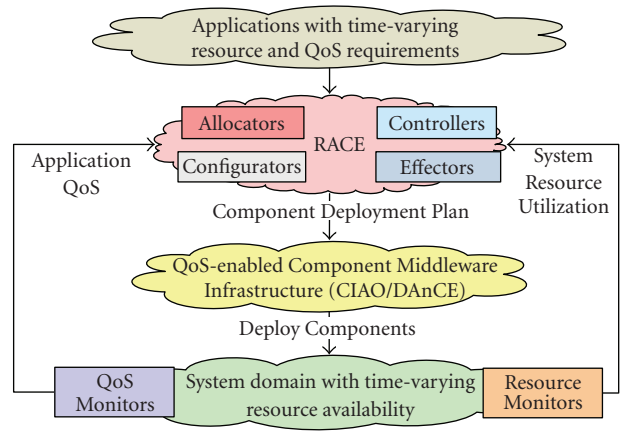


Figure 1: A resource allocation and control engine (RACE) for open DRE systems.

As shown in Figure 1, RACE provides (1) *resource monitors* that track utilization of various system resources, such as CPU, memory, and network bandwidth; (2) *QoS monitors* that track application QoS, such as end-to-end delay; (3) *resource allocators* that allocate resource to components based on their resource requirements and current availability of system resources; (4) *configurators* that configure middleware QoS parameters of application components; (5) *controllers* that compute end-to-end adaptation decisions based on control algorithms to ensure that QoS requirements of applications are met; and (6) *effectors* that perform controller-recommended adaptations.

(ii) *Evaluate the effectiveness of RACE in the context of NASA's magnetospheric multiscale system (MMS) mission*, which is representative open DRE system. The MMS mission system consists of a constellation of spacecrafts that maintain a specific formation while orbiting over a region of scientific interest. In these spacecrafts, availability of resource such as processing power (CPU), storage, network bandwidth, and power (battery) are limited and subjected to run-time variations. Moreover, resource utilization by, and input workload of, applications that execute in this system cannot be accurately characterized a priori. This paper evaluates the adaptive resource management capabilities of RACE in the context of this representative open DRE system. Our results demonstrate that when adaptive resource management algorithms for DRE systems are implemented using RACE, they yield a predictable and high-performance system, even in the face of changing operational conditions and workloads.

(iii) *The empirical evaluation of RACE's scalability* as the number of nodes and applications in a DRE system grows. Scalability is an integral property of a framework as it determines the framework's applicability. Since open DRE systems comprise large number of nodes and applications, to determine whether RACE can be applied to such systems, we empirically evaluate RACE's scalability as the number of applications and nodes in the system increases. Our results

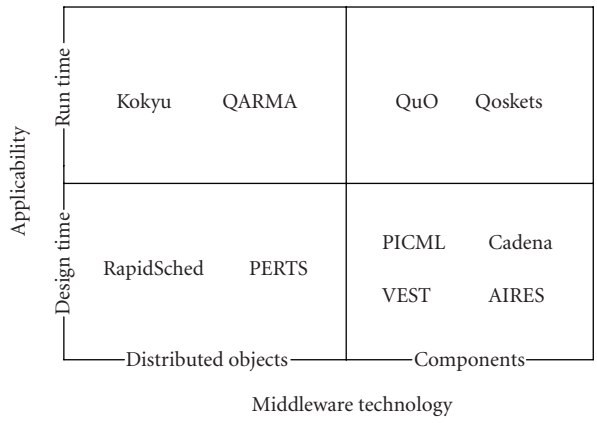


Figure 2: Taxonomy of related research.

demonstrate that RACE scales as well as the number of applications and nodes in the system increases, and therefore can be applied to a wide range of open DRE systems.

The remainder of the paper is organized as follows: Section 2 compares our research on RACE with related work; Section 3 motivates the use of RACE in the context of a representative DRE system case study; Section 4 describes the architecture of RACE and shows how it aids in the development of the case study described in Section 3; Section 5 empirically evaluates the performance of the DRE system when control algorithms are used in conjunction with RACE and also presents an empirical measure of RACE's scalability as the number of applications and nodes in the system grows; and Section 6 presents concluding remarks.

2. RESEARCH BACKGROUND AND RELATED WORK COMPARISON

This section presents an overview of existing middleware technologies that have been used to develop open DRE system and also compares our work on RACE with related research on building open DRE systems. As in Figure 2 and described below, we classify this research along two orthogonal dimensions: (1) QoS-enabled DOC middleware versus QoS-enabled component middleware, and (2) design-time versus run-time QoS configuration, optimization, analysis, and evaluation of constraints, such as timing, memory, and CPU.

2.1. Overview of conventional and QoS-enabled DOC middleware

Conventional middleware technologies for distributed object computing (DOC), such as the object management group (OMG)'s CORBA [16] and Sun's Java RMI [17], encapsulates and enhances native OS mechanisms to create reusable network programming components. These technologies provide a layer of abstraction that shields application developers from the low-level platform-specific details and define higher-level distributed programming models whose reusable API's and components automate and extend native OS capabilities.

Conventional DOC middleware technologies, however, address only *functional* aspects of system/application development such as how to define and integrate object interfaces and implementations. They do not address QoS aspects of system/-application development such as how to (1) define and enforce application timing requirements, (2) allocate resources to applications, and (3) configure OS and network QoS policies such as priorities for application processes and/or threads. As a result, the code that configures and manages QoS aspects often become entangled with the application code. These limitations with conventional DOC middleware have been addressed by the following run-time platforms and design-time tools.

(i) *Run-time:* early work on resource management middleware for shipboard DRE systems presented in [18, 19] motivated the need for adaptive resource management middleware. This work was further extended by QARMA [20], which provides resource management as a *service* for existing QoS-enabled DOC middleware, such as RT-CORBA. Kokyu [21] also enhances RT-CORBA QoS-enabled DOC middleware by providing a portable middleware scheduling framework that offers flexible scheduling and dispatching services. Kokyu performs feasibility analysis based on estimated worst case execution times of applications to determine if a set of applications is *schedulable*. Resource requirements of applications, such as memory and network bandwidth, are not captured and taken into consideration by Kokyu. Moreover, Kokyu lacks the capability to track utilization of various system resources as well as QoS of applications. To address these limitations, research presented in [22] enhances QoS-enabled DOC middleware by combining Kokyu and QARMA.

(ii) *Design-time:* RapidSched [23] enhances QoS-enabled DOC middleware, such as RT-CORBA, by computing and enforcing distributed priorities. RapidSched uses PERTS [24] to specify real-time information, such as deadline, estimated execution times, and resource requirements. Static schedulability analysis (such as rate monotonic analysis) is then performed and priorities are computed for each CORBA object in the system. After the priorities are computed, RapidSched uses RT-CORBA features to enforce these computed priorities.

2.2. Overview of conventional and QoS-enabled component middleware

Conventional component middleware technologies, such as the CORBA component model (CCM) [25] and enterprise Java beans [26, 27], provide capabilities that addresses the limitation of DOC middleware technologies in the context of system design and development. Examples of additional capabilities offered by conventional component middleware compared to conventional DOC middleware technology include (1) standardized interfaces for application component interaction, (2) model-based tools for deploying and interconnecting components, and (3) standards-based mechanisms for installing, initializing, and configuring application

components, thus separating concerns of application development, configuration, and deployment.

Although conventional component middleware support the design and development of large scale distributed systems, they do not address the QoS limitations of DOC middleware. Therefore, conventional component middleware can support large scale enterprise distributed systems, but not DRE systems that have the stringent QoS requirements. These limitations with conventional component-based middleware have been addressed by the following run-time platforms and design-time tools.

(i) *Run-time:* QoS provisioning frameworks, such as QuO [28] and Qoskets [8, 29, 30], help ensure desired performance of DRE systems built atop QoS-enabled DOC middleware and QoS-enabled component middleware, respectively. When applications are designed using Qoskets (1) resources are dynamically (re)allocated to applications in response to changing operational conditions and/or input workload and (2) application parameters are fine-tuned to ensure that allocated resources are used effectively. With this approach, however, applications are augmented explicitly at design-time with Qosket components, such as monitors, controllers, and effectors. This approach thus requires redesign and reassembly of existing applications built without Qoskets. When applications are generated at run-time (e.g., by intelligent mission planners [31]), this approach would require planners to augment the applications with Qosket components, which may be infeasible since planners are designed and built to solve mission goals and not perform such platform-/middleware-specific operations.

(ii) *Design-time:* Cadena [32] is an integrated environment for developing and verifying component-based DRE systems by applying static analysis, model-checking, and lightweight formal methods. Cadena also provides a component assembly framework for visualizing and developing components and their connections. VEST [33] is a design assistant tool based on the *generic modeling environment* [34] that enables embedded system composition from component libraries and checks whether timing, memory, power, and cost constraints of real-time and embedded applications are satisfied. AIRES [35] is a similar tool that provides the means to map design-time models of component composition with real-time requirements to run-time models that weave together timing and scheduling attributes. The research presented in [36] describes a design assistant tool, based on MAST [37], that comprises a DSML and a suite of analysis and system QoS configuration tools and enables composition, schedulability analysis, and assignment of operating system priority for application components.

Some design-time tools, such as AIRES, VEST, and those presented in [36], use *estimates*, such as estimated worst case execution time, estimated CPU, memory, and/or network bandwidth requirements. These tools are targeted for systems that execute in *closed* environments, where operational conditions, input workload, and resource availability can be characterized accurately a priori. Since RACE tracks and manages utilization of various system resources, as well as application QoS, it can be used in conjunction with these tools to build open DRE systems.

2.3. Comparing RACE with related work

Our work on RACE extends earlier work on QoS-enabled DOC middleware by providing an adaptive resource management framework for open DRE systems built atop QoS-enabled component middleware. DRE systems built using RACE benefit from (1) adaptive resource management capabilities of RACE and (2) additional capabilities offered by QoS-enabled component middleware compared to QoS-enabled DOC middleware, as discussed in Section 2.2.

Compared to related research presented in [18–20], RACE is an adaptive resource management framework that can be customized and configured using model-driven deployment and configuration tools such as the *platform-independent component modeling language* (PICML) [38]. Moreover, RACE provides adaptive resource and QoS management capabilities more transparently and nonintrusively than Kokyu, QuO, and Qoskets. In particular, it allocates CPU, memory, and networking resources to application components and tracks and manages utilization of various system resources, as well as application QoS. In contrast to our own earlier work on QoS-enabled DOC middleware, such as FC-ORB [14] and HiDRA [13], RACE is a QoS-enabled component middleware framework that enables the deployment and configuration of feedback control loops in DRE systems.

In summary, RACE's novelty stems from its combination of (1) design-time model-driven tools that can both design applications and customize and configure RACE itself, (2) QoS-enabled component middleware run-time platforms, and (3) research on control-theoretic adaptive resource management. RACE can be used to deploy and manage component-based applications that are composed at design-time via model-driven tools, as well as at run-time by *intelligent mission planners* [39], such as SA-POP [31].

3. CASE STUDY: MAGNETOSPHERIC MULTISCALE (MMS) MISSION DRE SYSTEM

This section presents an overview of NASA's magnetospheric multiscale (MMS) mission [40] as a case study to motivate the need for RACE in the context of open DRE systems. We also describe the resource and QoS management challenges involved in developing the MMS mission using QoS-enabled component middleware.

3.1. MMS mission system overview

NASA's MMS mission system is a representative open DRE system consisting of several interacting subsystems (both in-flight and stationary) with a variety of complex QoS requirements. As shown in Figure 3, the MMS mission consists of a constellation of five spacecrafts that maintain a specific formation while orbiting over a region of scientific interest. This constellation collects science data pertaining to the earth's plasma and magnetic activities while in orbit and send it to a ground station for further processing. In the MMS mission spacecrafts, availability of resource such as processing power (CPU), storage, network bandwidth, and power (battery) are

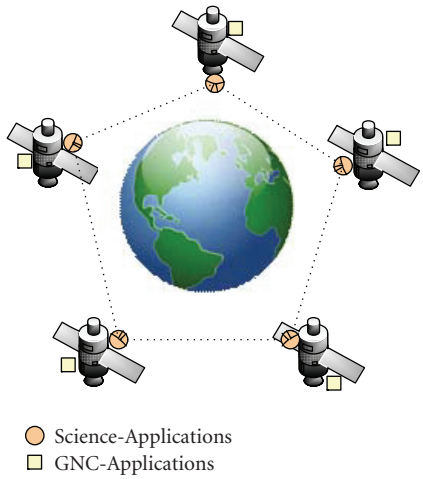


Figure 3: MMS mission system.

limited and subjected to run-time variations. Moreover, resource utilization by, and input workload of, applications that execute in this system cannot be accurately characterized a priori. These properties make the MMS mission system an open DRE system.

Applications executing in this system can be classified as guidance, navigation, and control (GNC) applications and science applications. The GNC applications are responsible for maintaining the spacecraft within the specified orbit. The science applications are responsible for collecting science data, compressing and storing the data, and transmitting the stored data to the ground station for further processing.

As shown in Figure 3, GNC applications are localized to a single spacecraft. Science applications tend to span the entire spacecraft constellation, *that is*, all spacecrafts in the constellation have to coordinate with each other to achieve the goals of the science mission. GNC applications are considered *hard real-time* applications (i.e., the penalty of not meeting QoS requirement(s) of these applications is very high, often fatal to the mission), whereas science applications are considered *soft real-time* applications (i.e., the penalty of not meeting QoS requirement(s) of these applications is high, but not fatal to the mission).

Science applications operate in three modes: *slow survey*, *fast survey*, and *burst* mode. Science applications switch from one mode to another in reaction to one or more *events of interest*. For example, for a science application that monitors the earth's plasma activity, the *slow* survey mode is entered outside the regions of scientific interests and enables only a minimal set of data acquisition (primarily for health monitoring). The *fast* survey mode is entered when the spacecrafts are within one or more regions of interest, which enables data acquisition for all payload sensors at a moderate rate. If plasma activity is detected while in fast survey mode, the application enters *burst* mode, which results in data collection at the highest data rates. Resource utilization by, and importance of, a science application is determined by its mode of operation, which is summarized by Table 1.

Table 1: Characteristics of science application.

Mode	Relative importance	Resource consumption
Slow survey	Low	Low
Fast survey	Medium	Medium
Burst	High	High

Each spacecraft consists of an onboard intelligent mission planner, such as the *spreading activation partial-order planner* (SA-POP) [31] that decomposes overall mission goal(s) into GNC and science applications that can be executed concurrently. SA-POP employs decision-theoretic methods and other AI schemes (such as hierarchical task decomposition) to decompose mission goals into navigation, control, data gathering, and data processing applications. In addition to initial generation of GNC and science applications, SA-POP incrementally generates new applications in response to changing mission goals and/or degraded performance reported by onboard mission monitors.

We have developed a prototype implementation of the MMS mission systems in conjunction with our colleagues at Lockheed Martin Advanced Technology Center, Palo Alto, California. In our prototype implementation, we used the *component-integrated ACE ORB* (CIAO) [41] and *deployment and configuration engine* (DAnCE) [42] as the QoS-enabled component middleware platform. Each spacecraft uses SA-POP as its onboard intelligent mission planner.

3.2. Adaptive resource management requirements of the MMS mission system

As discussed in Section 2.2, the use of QoS-enabled component middleware to develop open DRE systems, such as the NASA MMS mission, can significantly improve the design, development, evolution, and maintenance of these systems. In the absence of an adaptive resource management framework, however, several key requirements remain unresolved when such systems are built in the absence of an adaptive resource management framework. To motivate the need for RACE, the remainder of this section presents the key resource and QoS management requirements that we addressed while building our prototype of the MMS mission DRE system.

3.2.1. Requirement 1: resource allocation to applications

Applications generated by SA-POP are *resource sensitive*, *that is*, QoS is affected significantly if an application does not receive the required CPU time and network bandwidth within bounded delay. Moreover, in open DRE systems like the MMS mission, input workload affects utilization of system resources and QoS of applications. Utilization of system resources and QoS of applications may therefore vary significantly from their estimated values. Due to the operating conditions for open DRE systems, system resource availability, such as available network bandwidth, may also be time variant.

A resource management framework therefore needs to (1) monitor the current utilization of system resources,

(2) allocate resources in a timely fashion to applications such that their resource requirements are met using resource allocation algorithms such as PBFD [43], and (3) support multiple resource allocation strategies since CPU and memory utilization overhead might be associated with implementations of resource allocation algorithms themselves and select the appropriate one(s) depending on properties of the application and the overheads associated with various implementations. Section 4.2.1 describes how RACE performs online resource allocation to application components to address this requirement.

3.2.2. Requirement 2: configuring platform-specific QoS parameters

The QoS experienced by applications depend on various platform-specific real-time QoS configurations including (1) *QoS configuration of the QoS-enabled component middleware*, such as priority model, threading model, and request processing policy; (2) *operating system QoS configuration*, such as real-time priorities of the process(es) and thread(s) that host and execute within the components, respectively; and (3) *networks QoS configurations*, such as *diffserv* code points of the component interconnections. Since these configurations are platform-specific, it is tedious and error-prone for system developers or SA-POP to specify them in isolation.

An adaptive resource management framework therefore needs to provide abstractions that shield developers and/or SA-POP from low-level platform-specific details and define higher-level QoS specification models. System developers and/or intelligent mission planners should be able to specify QoS characteristics of the application such as QoS requirements and relative importance, and the adaptive resource management framework should then configure the platform-specific parameters accordingly. Section 4.2.2 describes how RACE provides a higher level of abstractions and shield system developers and SA-POP from low-level platform-specific details to address this requirement.

3.2.3. Requirement 3: enabling dynamic system adaptation and ensuring QoS requirements are met

When applications are deployed and initialized, resources are allocated to application components based on the *estimated* resource utilization and estimated/current availability of system resources. In open DRE systems, however, *actual* resource utilization of applications might be significantly different than their estimated values, as well as availability of system resources vary dynamically. Moreover, for applications executing in these systems, the relation between input workload, resource utilization, and QoS cannot be characterized a priori.

An adaptive resource management framework therefore needs to provide monitors that track system resource utilization, as well as QoS of applications, at run-time. Although some QoS properties (such as accuracy, precision, and fidelity of the produced output) are application-specific, certain QoS (such as *end-to-end latency* and throughput) can be tracked by the framework transparently to the application.

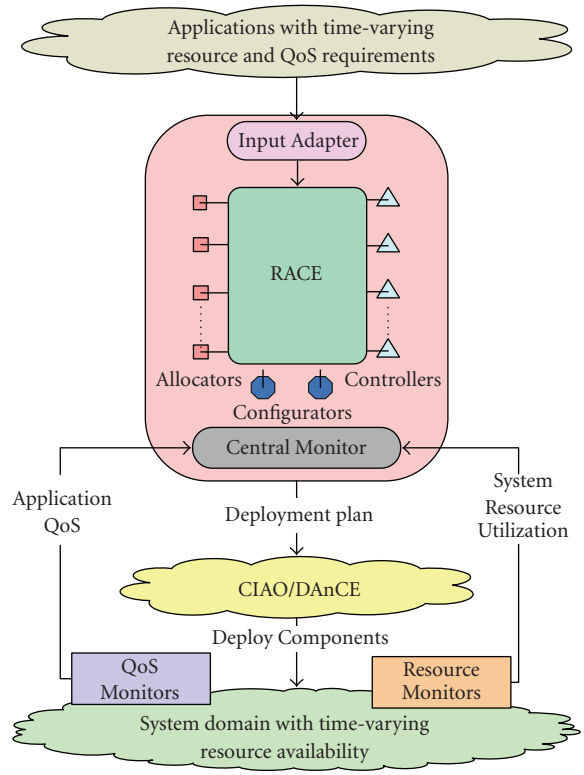


Figure 4: Detailed design of RACE.

However, customization and configuration of the framework with domain-specific monitors (both platform-specific resource monitors and application-specific QoS monitors) should be possible. In addition, the framework needs to enable the system to *adapt* to dynamic changes, such as variations in operational conditions, input workload, and/or resource availability. Section 4.2.3 demonstrates how RACE performs system adaptation and ensures QoS requirements of applications are met to address this requirement.

4. STRUCTURE AND FUNCTIONALITY OF RACE

This section describes the structure and functionality of RACE. RACE supports open DRE systems built atop CIAO, which is an open-source implementation of lightweight CCM. All entities of RACE themselves are designed and implemented as CCM components, so RACE's *Allocators* and *Controllers* can be configured to support a range of resource allocation and control algorithms using model-driven tools, such as PICML.

4.1. Design of RACE

Figure 4 elaborates the earlier architectural overview of RACE in Figure 1 and shows how the detailed design of RACE is composed of the following components: (1) *InputAdapter*, (2) *CentralMonitor*, (3) *Allocators*, (4) *Configurators*, (5) *Controllers*, and (6) *Effectors*. RACE monitors application QoS and system resource usage via its *Resource*

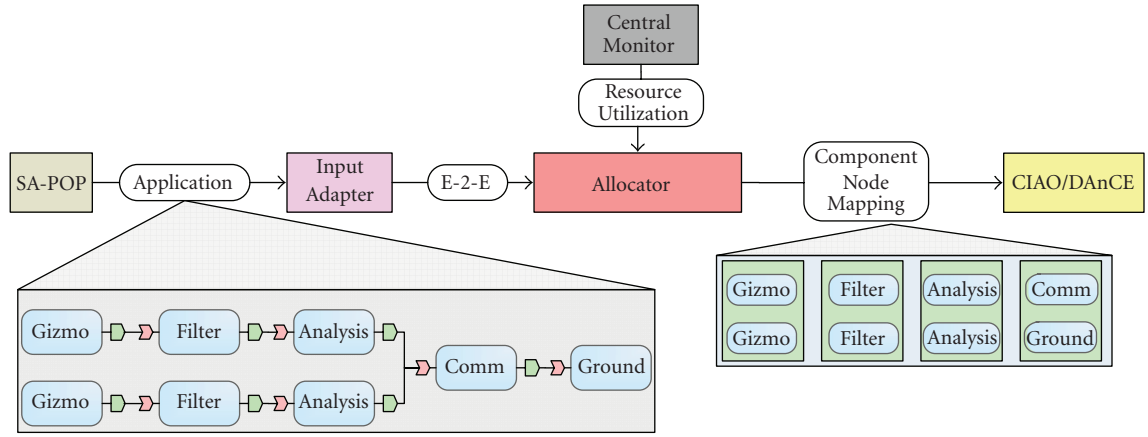


Figure 5: Resource allocation to application components using RACE.

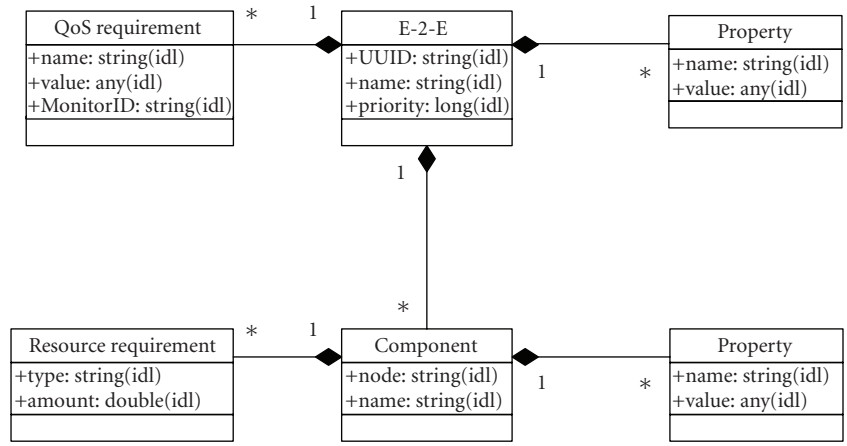


Figure 6: Main entities of RACE's E-2-E IDL structure.

Monitor, *QoS-Monitors*, *Node Monitors*, and *Central Monitor*. Each component in RACE is described below in the context of the overall adaptive resource management challenge it addresses.

4.1.1. *Challenge 1: domain-specific representation of application metadata*

Problem

End-to-end applications can be composed either at design-time or at run-time. At design-time, CCM-based end-to-end applications are composed using model-driven tools, such as PICML; and at run-time, they can be composed by intelligent mission planners like SA-POP. When an application is composed using PICML, metadata describing the application is captured in XML files based on the *PackageConfiguration* schema defined by the object management group's deployment and configuration specification [44]. When applications are generated during run-time by SA-POP, metadata is captured in an in-memory structure defined by the planner.

Solution: domain-specific customization and configuration of RACE's adapters

During design-time, RACE can be configured using PICML and an *InputAdapter* appropriate for the domain/system can be selected. For example, to manage a system in which applications are constructed at design-time using PICML, RACE can be configured with the *PICMLInputAdapter*; and to manage a system in which applications are constructed at run-time using SA-POP, RACE can be configured with the *SAPOPInputAdapter*. As shown in Figure 5, the *InputAdapter* parses the metadata that describes the application into an in-memory end-to-end (*E-2-E*) IDL structure that is internal to RACE. Key entities of the *E-2-E* IDL structure are shown in Figure 6.

The *E-2-E* IDL structure populated by the *InputAdapter* contains information regarding the application, including (1) components that make up the application and their resource requirement(s), (2) interconnections between the components, (3) application QoS properties (such relative priority) and QoS requirement(s) (such as end-to-end delay), and (4) mapping components onto domain nodes. The

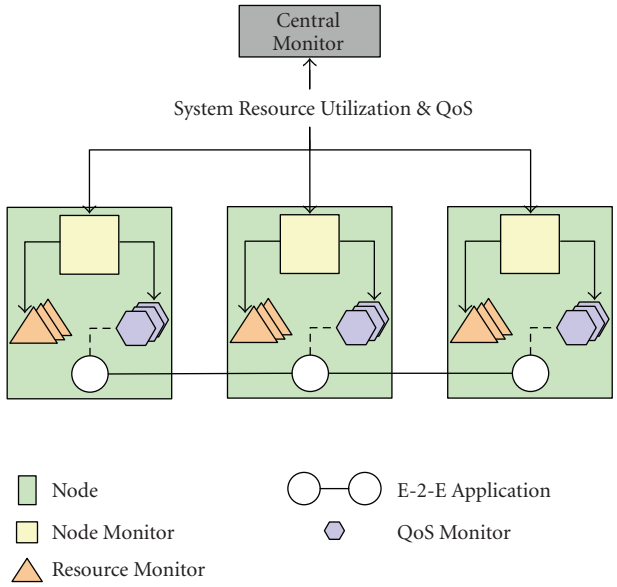


Figure 7: Architecture of monitoring framework.

mapping of components onto nodes need not be specified in the metadata that describes the application which is given to RACE. If a mapping is specified, it is honored by RACE; if not, a mapping is determined at run-time by RACE's *Allocators*.

4.1.2. Challenge 2: efficient monitoring of system resource utilization and application QoS

Problem

In open DRE systems, input workload, application QoS, and utilization and availability of system resource are subject to dynamic variations. In order to ensure application QoS requirements are met, as well as utilization of system resources are within specified bounds, application QoS and utilization/availability of system resources are to be monitored periodically. The key challenge lies in designing and implementing a resource and QoS monitoring architecture that scales as well as the number of applications and nodes in the system increase.

Solution: hierarchical QoS and resource monitoring architecture

RACE's monitoring framework is composed of the *Central Monitor*, *Node Monitors*, *Resource Monitors*, and *QoS Monitors*. These components track resource utilization by, and QoS of, application components. As shown in Figure 7, RACE's *Monitors* are structured in the following hierarchical fashion. A *Resource Monitor* collects resource utilization metrics of a specific resource, such as CPU or memory. A *QoS Monitor* collects specific QoS metrics of an application, such as end-to-end latency or throughput. A *Node Monitor* tracks the QoS of all the applications running on a node as well as the resource utilization of that node. Finally, a *Central Monitor* tracks the QoS of all the applications running the entire system, which captures the system QoS, as well as the resource utilization of the entire system, which captures the system resource utilization.

Resource Monitors use the operating system facilities, such as */proc* file system in *Linux/Unix* operating systems and the *system registry* in *Windows* operating systems, to collect resource utilization metrics of that node. As the resource monitors are implemented as shared libraries that can be loaded at run-time, RACE can be configured with new-/domain-specific resource monitors without making any modifications to other entities of RACE. *QoS-Monitors* are implemented as software modules that collect end-to-end latency and throughput metrics of an application and are dynamically installed into a running system using DyInst [45]. This approach ensure rebuilding, reimplementation, or restarting of already running application components are not required. Moreover, with this approach, *QoS-Monitors* can be turned on or off on demand at run-time.

The primary metric that we use to measure the performance of our monitoring framework is *monitoring delay*, which is defined as the time taken to obtain a snapshot of the entire system in terms of resource utilization and QoS. To minimize the monitoring delay and ensure that RACE's monitoring architecture scales as the number of applications and nodes in the system increase, the RACE's monitoring architecture is structured in a hierarchical fashion. We validate this claim in Section 5.

4.1.3. Challenge 3: resource allocation

Problem

Applications executing in open DRE systems are resource sensitive and require multiple resources such as memory, CPU, and network bandwidth. In open DRE systems, resources allocation cannot be performed during design-time as system resource availability may be time variant. Moreover, input workload affects the utilization of system resources by already executing applications. Therefore, the key challenge lies in allocating various systems resources to application components in a timely fashion.

Solution:online resource allocation

RACE's *Allocators* implement resource allocation algorithms and allocate various domain resources (such as CPU, memory, and network bandwidth) to application components by determining the mapping of components onto nodes in the system domain. For certain applications, *static* mapping between components and nodes may be specified at design-time by system developers. To honor these static mappings, RACE therefore provides a *static allocator* that ensures components are allocated to nodes in accordance with the static mapping specified in the application's metadata. If no static mapping is specified, however, *dynamic allocators* determine the component to node mapping at run-time based on resource requirements of the components and current resource availability on the various nodes in the domain. As shown in

Figure 5, input to *Allocators* include the *E-2-E* IDL structure corresponding to the application and the current utilization of system resources.

The current version of RACE provides the following *Allocators*: (1) a single dimension binpacker [46] that makes allocation decisions based on either CPU, memory, or network bandwidth requirements and availability, (2) a multidimensional binpacker—partitioned breadth first decreasing allocator [43]—that makes allocation decisions based on CPU, memory, and network bandwidth requirements and availability, and (3) a static allocator. Metadata is associated with each allocator and captures its type (i.e., static, single dimension binpacking, or multidimensional binpacker) and associated resource overhead (such as CPU and memory utilization). Since *Allocators* themselves are CCM components, RACE can be configured with new *Allocators* by using PICML.

4.1.4. Challenge 4: accidental complexities in configuring platform-specific QoS parameters

Problem

As described in Section 3.2.2, real-time QoS *configuration* of the underlying component middleware, operating system, and network affects the QoS of applications executing in open DRE systems. Since these configurations are platform-specific, it is tedious and error-prone for system developers or SA-POP to specify them in isolation.

Solution: automate configuration of platform-specific parameters

As shown in Figure 8, RACE's *Configurators* determine values for various low-level platform-specific QoS parameters, such as middleware, operating system, and network settings for an application based on its QoS characteristics and requirements such as relative importance and end-to-end delay. For example, the *MiddleWareConfigurator* configures component lightweight CCM policies, such as threading policy, priority model, and request processing policy based on the class of the application (*important* and *best effort*). The *OperatingSystemConfigurator* configures operating system parameters, such as the priorities of the *component servers* that host the components based on rate monotonic scheduling (RMS) [46] or based on criticality (relative importance) of the application. Likewise, the *NetworkConfigurator* configures network parameters, such as *diffserv* code points of the component interconnections. Like other entities of RACE, *Configurators* are implemented as CCM components, so new configurators can be plugged into RACE by configuring RACE at design-time using PICML.

4.1.5. Challenge 5: computation of system adaptation decisions

Problem

In open DRE systems, resource utilization of applications might be significantly different than their estimated values

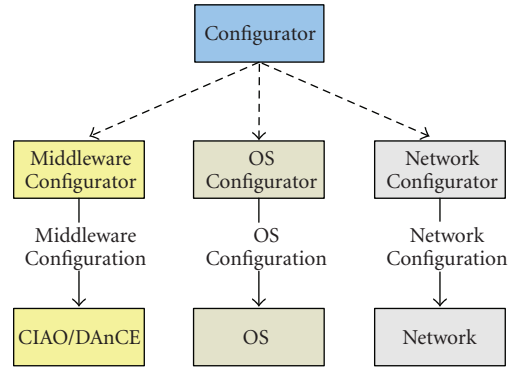


Figure 8: QoS parameter configuration with RACE.

and availability of system resources may be time variant. Moreover, for applications executing in these systems, the relation between input workload, resource utilization, and QoS cannot be characterized a priori. Therefore, in order to ensure that QoS requirements of applications are met, and utilization system resources are within the specified bounds, the system must be able to *adapt* to dynamic changes, such as variations in operational conditions, input workload, and/or resource availability.

Solution: control-theoretic adaptive resource management algorithms

RACE's *Controllers* implement various Control-theoretic adaptive resource management algorithms such as EUCON [9], DEUCON [10], HySUCON [11], and FMUF [12], thereby enabling open DRE systems to adapt to changing operational context and variations in resource availability and/or demand. Based on the control algorithm they implement, *Controllers* modify configurable system parameters, such as execution rates and mode of operation of the application, real-time configuration settings—operating system priorities of *component servers* that host the components—and network *diffserv* code points of the component interconnections. As shown in Figure 9, input to the controllers include current resource utilization and current QoS. Since *Controllers* are implemented as CCM components, RACE can be configured with new *Controllers* by using PICML.

4.1.6. Challenge 6: efficient execution of system adaptation decisions

Problem

Although control theoretic adaptive resource management algorithms compute system adaptation decisions, one of the challenges we faced in building RACE is the design and implementation of *effectors*—entities that modify system parameters in order to achieve the controller recommended system adaptation. The key challenge lies in designing and implementing the effector architecture that scales as well as the number of applications and nodes in the system increases.

Solution: hierarchical effector architecture

Effectors modify system parameters, including resources allocated to components, execution rates of applications, and OS/middleware/network QoS setting for components, to achieve the controller recommended adaptation. As shown in Figure 9, *Effectors* are designed hierarchically. The *Central Effector* first computes the values of various system parameters for all the nodes in the domain to achieve the *Controller*-recommended adaptation. The computed values of system parameters for each node are then propagated to *Effectors* located on each node, which then modify system parameters of its node accordingly.

The primary metric that is used to measure the performance of a monitoring effectors is *actuation delay*, which is defined as the time taken to execute controller-recommended adaptation throughout the system. To minimize the actuation delay and ensure that RACE scales as the number of applications and nodes in the system increases, the RACE's effectors are structured in a hierarchical fashion. We validate this claim in Section 5.

Since the elements of RACE are developed as CCM components, RACE itself can be configured using model-driven tools, such as PICML. Moreover, new- and/or domain-specific entities, such as *InputAdapters*, *Allocators, Controllers, Effectors, Configurators, QoS-Monitors*, and *Resource Monitors*, can be plugged directly into RACE without modifying RACE's existing architecture.

4.2. Addressing MMS mission requirements using RACE

Section 4.1 provides a detailed overview of various adaptive resource management challenges of open DRE systems and how RACE addresses these challenges. We now describe how RACE was applied to our MMS mission case study from Section 3 and show how it addressed key resource allocation, QoS-configuration, and adaptive resource management requirements that we identified in Section 3.

4.2.1. Addressing requirement 1: resource allocation to applications

RACE's *InputAdapter* parses the metadata that describes the application to obtain the resource requirement(s) of components that make up the application and populates the *E-2-E* IDL structure. The *Central Monitor* obtains system resource utilization/availability information for RACE's *Resource Monitors*, and using this information along with the *estimated* resource requirement of application components captured in the *E-2-E* structure, the *Allocators* map components onto nodes in the system domain based on run-time resource availability.

RACE's *InputAdapter, Central Monitor*, and *Allocators* coordinate with one another to allocate resources to applications executing in open DRE systems, thereby addressing the resource allocation requirement for open DRE systems identified in Section 3.2.1.

4.2.2. Addressing requirement 2: configuring platform-specific QoS parameters

RACE shields application developers and SA-POP from low-level platform-specific details and defines a higher-level QoS specification model. System developers and SA-POP specify only QoS characteristics of the application, such as QoS requirements and relative importance, and RACE's *Configurators* automatically configures platform-specific parameters appropriately.

For example, consider two science applications—one executing in fast survey mode and one executing in slow survey mode. For these applications, middleware parameters configured by the *Middleware Configurator* includes (1) CORBA end-to-end priority, which is configured based on execution mode (fast/slow survey) and application period/deadline; (2) CORBA priority propagation model (CLIENT_PROPAGATED/SERVER_DECLARED), which is configured based on the application structure and interconnection; and (3) threading model (single threaded/thread-pool/thread-pool with lanes), which is configured based on number of concurrent peer components connected to a component. The *Middleware Configurator* derives configuration for such low-level platform-specific parameters from application end-to-end structure and QoS requirements.

RACE's *Configurators* provides higher-level abstractions and shield system developers and SA-POP from low-level platform-specific details, thus addressing the requirements associated with configuring platform-specific QoS parameters identified in Section 3.2.2.

4.2.3. Addressing requirement 3: monitoring end-to-end QoS and ensuring QoS requirements are met

When resources are allocated to components at design-time by system designers using PICML, *that is,* mapping of application components to nodes in the domain are specified, these operations are performed based on estimated resource utilization of applications and estimated availability of system resources. Allocation algorithms supported by RACE's *Allocators* allocate resources to components based on current system resource utilization and component's estimated resource requirements. In open DRE systems, however, there is often no accurate a priori knowledge of input workload, the relationship between input workload and resource requirements of an application, and system resource availability.

To address this requirement, RACE's control architecture employs a feedback loop to manage system resource and application QoS and ensures (1) QoS requirements of applications are met at all times and (2) system stability by maintaining utilization of system resources below their specified utilization set-points. RACE's control architecture features a feedback loop that consists of three main components: *Monitors, Controllers*, and *Effectors*, as shown in Figure 9.

Monitors are associated with system resources and QoS of the applications and periodically update the *Controller* with the current resource utilization and QoS of applications currently running in the system. The *Controller* implements a particular control algorithm such as EUCON [9], DEUCON

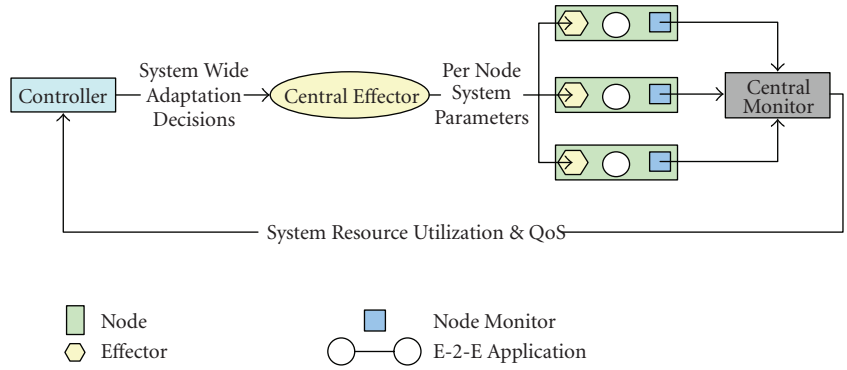


Figure 9: RACE's feedback control loop.

Table 2: Lines of source code for various system elements.

Entity	Total lines of source code
MMS DRE system	19,875
RACE	157,253
CIAO/DAnCE	511,378

[10], HySUCON [11], and FMUF [12], and computes the adaptations decisions for each (or a set of) application(s) to achieve the desired system resource utilization and QoS. *Effectors* modify system parameters, which include resource allocation to components, execution rates of applications, and OS/middleware/network QoS setting of components, to achieve the controller-recommended adaptation.

As shown in Figure 9, RACE's monitoring framework, *Controllers*, and *Effectors* coordinate with one another and the aforementioned entities of RACE to ensure (1) QoS requirements of applications are met and (2) utilization of system resources are maintained within the specified utilization set-point set-point(s), thereby addressing the requirements associated with run-time end-to-end QoS management identified in Section 3.2.3. We empirically validate this in Section 5.

5. EMPIRICAL RESULTS AND ANALYSIS

This section presents the design and results of experiments that evaluate the performance and scalability of RACE in our prototype of the NASA MMS mission system case study described in Section 3. These experiments validate our claims in Sections 4 and 4.2 that RACE is an scalable adaptive resource management framework and can perform effective end-to-end adaptation and yield a predictable and scalable DRE system under varying operating conditions and input workload.

5.1. Hardware and software test-bed

Our experiments were performed on the ISISLab test-bed at Vanderbilt University (www.dre.vanderbilt.edu/ISISlab). The hardware configuration consists of six nodes, five of which acted as spacecrafts and one acted as a ground station. The hardware configuration of all the nodes was a 2.8 GHz Intel Xeon dual processor, 1 GB physical memory, 1 GHz ethernet network interface, and 40 GB hard drive. The Redhat Fedora core release 4 OS with real-time preemption patches [47] was used for all the nodes.

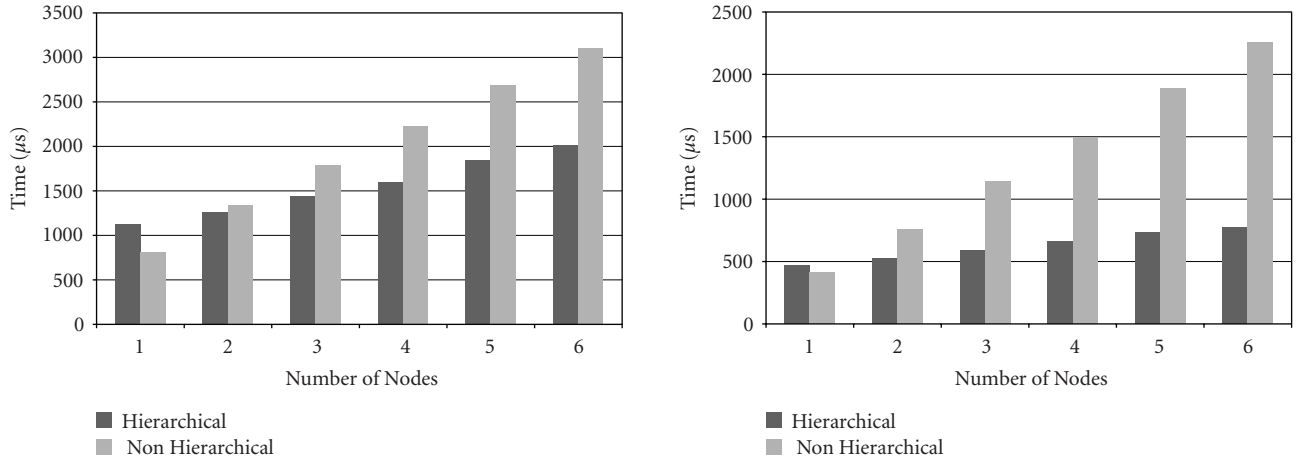
Our experiments also used CIAO/DAnCE 0.5.10, which is our open source QoS-enabled component middleware that implements the OMG lightweight CCM [48] and deployment and configuration [44] specifications. RACE and our DRE system case study are built upon CIAO/DAnCE.

5.2. MMS DRE system implementation

Science applications executing atop our MMS DRE system are composed of the following components:

(i) *plasma sensor component*, which manages and controls the plasma sensor on the spacecraft, collects metrics corresponding to the earth's plasma activity;
(ii) *camera sensor component*, which manages and controls the high-fidelity camera on the spacecraft and captures images of one or more star constellations;
(iii) *filter component*, which processes the data from the sensor components to remove any extraneous noise in the collected data/image;
(iv) *analysis component*, which processes the collected data to determine if the data is of interest or not. If the data is of interest, the data is compressed and transmitted to the ground station;
(v) *compression component*, which uses loss-less compression algorithms to compresses the collected data;
(vi) *communication component*, which transmits the compressed data to the ground station periodically;
(vii) *ground component*, which receives the compressed data from the spacecrafts and stores it for further processing.

All these components—except for the ground component—execute on the spacecrafts. Our experiments used component emulations that have the same resource utilization characteristics as the original components. Table 2 summarizes the number of lines of C++ code of various entities in our middleware, RACE, and our prototype implementation of



(a) Monitoring Delay versus Number of Nodes

(b) Actuation Delay versus Number of Nodes

Figure 10: Impact of increase in number of nodes on monitoring and actuation delay.

the MMS DRE system case study, which were measured using SLOCCount (www.dwheeler.com/sloccount/).

5.3. Evaluation of RACE's scalability

Sections 4.1.2 and 4.1.6 claimed that the hierarchical design of RACE's monitors and effectors enables RACE to scale as the number of applications and nodes in the system grows. We validated this claim by studying the impact of increasing number of nodes and applications on RACE's monitoring delay and actuation delay when RACE's monitors and effectors are configured hierarchically and nonhierarchically. As described in Sections 4.1.2 and 4.1.6, *monitoring delay* is defined as the time taken to obtain a snapshot of the entire system in terms of resource utilization and QoS and *actuation delay* is defined as the time taken to execute controller-recommended adaptation throughout the system.

To measure the monitoring and actuation delays, we instrumented RACE's *Central Monitor* and *Central Effector*, respectively, with high resolution timers—*ACE_High_Res_Timer* [15]. The timer in the *Central Monitor* measured the time duration from when requests were sent to individual *Node Monitors* to the time instant when replies from all *Node Monitors* were received and the data (resource utilization and application QoS) were assembled to obtain a snapshot of the entire system. Similarly, the timer in the *Central Effector* measured the time duration from when system adaptation decisions were received from the *Controller* to the time instant when acknowledgment indicating successful execution of node level adaption from individual *Effectors* (located on each node) were received.

5.3.1. Experiment 1: constant number of application and varying number of nodes

This experiment studied the impact of varying number of nodes in the system domain on RACE's monitoring and actuation delay. We present the results obtained from running the experiment with a constant of five applications, each composed of six components (plasma-sensor/camera-sensor, analysis, filter analysis, compression, communication, and ground), and a varying number of nodes.

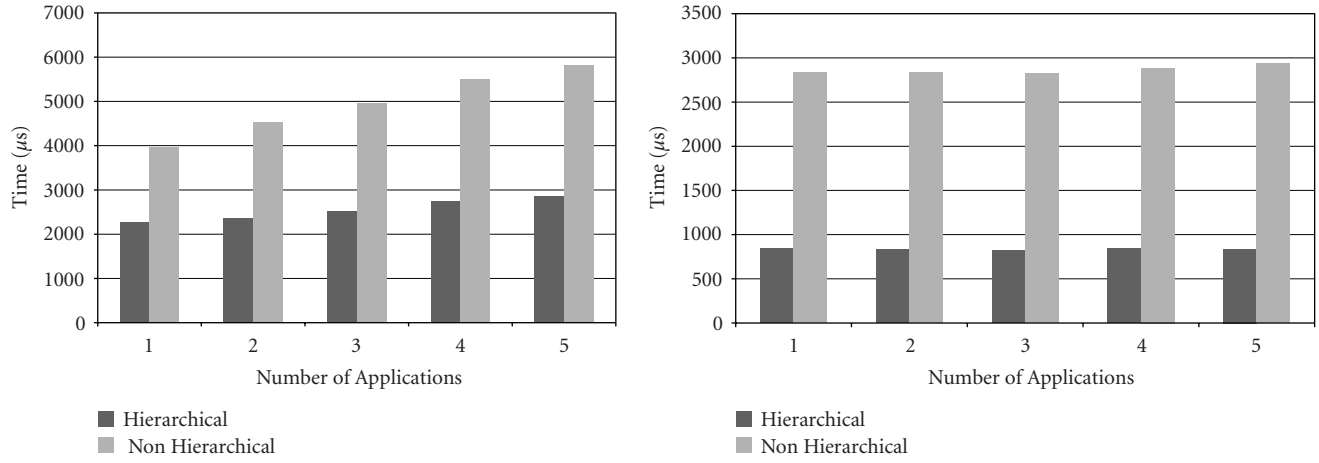
Experiment configuration

We varied the number of nodes in the system from one to six. A total of 30 application components were evenly distributed among the nodes in the system. The experiment was composed of two scenarios: (1) hierarchical and (2) nonhierarchical configuration of RACE's monitors and effectors. Each scenario was comprised of seven runs, and the number of nodes in the system during each run was. During each run, monitoring delay and actuation delay were collected over 50,000 iterations.

Analysis of results

Figures 10(a) and 10(b) compare the impact of increasing the number of nodes in the system on RACE's monitoring and actuation delay, respectively, under the two scenarios. Figures 10(a) and 10(b) show that monitoring and actuation delays are significantly lower in the hierarchical configuration of RACE's monitors and effectors compared to the nonhierarchical configuration. Moreover, as the number of nodes in the system increases, the increases in monitoring and actuation delays are significantly (i.e., 18% and 29%, resp.) lower in the hierarchical configuration compared to the nonhierarchical configuration. This result occurs because individual node monitors and effectors execute in parallel when monitors and effectors are structured hierarchically, thereby significantly reducing monitoring and actuation delay, respectively.

Figures 10(a) and 10(b) show the impact on monitoring and actuation delay when the monitors and effectors are structured hierarchically and the number of nodes in the system increases. Although individual monitors and effectors



(a) Monitoring Delay versus Number of Applications

(b) Actuation Delay versus Number of Applications

Figure 11: Impact of increase in number of application on monitoring and actuation delays.

execute in parallel, resource data aggregation and computation of per-node adaptation decisions are centralized by the *Central Monitor* and *Central Effector*, respectively. The results show that this configuration yields a marginal increase in the monitoring and actuation delay (i.e., 6% and 9%, resp.) as the number of nodes in the system increases.

Figures 10(a) and 10(b) show that when there is only one node in the system, the performance of the hierarchical configuration of RACE's monitors and effectors is worse than the nonhierarchical configuration. This result measures the overhead associated with the hierarchical configuration. As shown in Figures 10(a) and 10(b), however, as the number of nodes in the system increase, the benefit of the hierarchical configuration outweighs this overhead.

5.3.2. Experiment 2: constant number of nodes and varying number of applications

This experiment studied the impact of varying the number of applications on RACE's monitoring and actuation delay. We now present the results obtained from running the experiment with six nodes in the system and varying number of applications (from one to five), each composed of six components (plasma-sensor/camera-sensor, analysis, filter analysis, compression, communication, and ground).

Experiment configuration

We varied the number of applications in the system from one to five. Once again, the application components were evenly distributed among the six nodes in the system. This experiment was composed of two scenarios: (1) hierarchical and (2) nonhierarchical configuration of RACE's monitors and effectors. Each scenario was comprised of five runs, with the number of applications used in each run held constant. As we varied the number of applications from one to five, for each scenario we had a total of five runs. During each run, monitoring delay and actuation delay were collected over 50,000 iterations.

Analysis of results

Figures 11(a) and 11(b) compare the impact on increase in number of applications on RACE's monitoring and actuation delay, respectively, under the two scenarios. Figures 11(a) and 11(b) show that monitoring and actuation delays are significantly lower under the hierarchical configuration of RACE's monitors and effectors compared with the nonhierarchical configuration. These figures also show that under the hierarchical configuration, there is a marginal increase in the monitoring delay and negligible increase in the actuation delay as the number of applications in the system increase.

These results show that RACE scales as well as the number of nodes and applications in the system increase. The results also show that RACE's scalability is primarily due to the hierarchical design of RACE's monitors and effectors, there by validating our claims in Sections 4.1.2 and 4.1.6.

5.4. Evaluation of RACE's adaptive resource management capabilities

We now evaluate the adaptive resource management capabilities of RACE under two scenarios: (1) moderate workload, and (2) heavy workload. Applications executing on our prototype MMS mission DRE system were periodic, with deadline equal to their periods. In both the scenarios, we use the deadline miss ratio of applications as the metric to evaluate system performance. For every sampling period of RACE's *Controller*, deadline miss ratio for each application was computed as the ratio of number of times the application's end-to-end latency was greater than its deadline to the number of times the application was invoked. The end-to-end latency of an application was obtained from RACE's QoS *Monitors*.

5.4.1. Summary of evaluated scheduling algorithms

We studied the performance of the prototype MMS system under various configurations: (1) a baseline configuration without RACE and static priority assigned to application

Table 3: Application configuration under moderate workload.

Application	Component allocation		Ground station	Period (msec)	Mode
	Spacecraft				
	1	2			
1	Communication plasma-sensor	Analysis compression	Ground	1000	Fast survey
2	Analysis camera-sensor Filter	Communication compression	Ground	900	Slow survey
3	Plasma-sensor camera-sensor	Communication compression Filter	Ground	500	Slow survey

components based on rate monotonic scheduling (RMS) [46], (2) a configuration with RACE's maximum urgency first (MUF) *Configurator*, and (3) a configuration with RACE's MUF *Configurator* and flexible MUF (FMUF) [12] *Controller*. The goal of these experiments is not to compare the performance of various adaptive resource management algorithms, such as EUCON [9], DEUCON [10], HySUCON [11], or FMUF. Instead, the goal is to demonstrate how RACE can be used to implement these algorithms.

A disadvantage of RMS scheduling is that it cannot provide performance isolation for higher importance applications [49]. During system overload caused by dynamic increase in the workload, applications of higher importance with a low rate may miss deadlines. Likewise, applications with medium/lower importance but high rates may experience no missed deadlines.

In contrast, MUF provides performance isolation to applications of higher importance by dividing operating system and/or middleware priorities into two classes [49]. All components belonging to applications of higher importance are assigned to the high-priority class, while all components belonging to applications of medium/lower importance are assigned to the low-priority class. Components within a same priority class are assigned operating system and/or middleware priorities based on the RMS policy. Relative to RMS, however, MUF may cause priority inversion when a higher importance application has a lower rate than medium/lower importance applications. As a result, MUF may unnecessarily cause an application of medium/lower importance to miss its deadline, even when all tasks are schedulable under RMS.

To address limitations with MUF, RACE's FMUF *Controller* provides performance isolation for applications of higher importance while reducing the deadline misses of applications of medium/lower importance. While both RMS and MUF assign priorities statically at deployment time, the FMUF *Controller* adjusts the priorities of applications of medium/lower importance dynamically based on performance feedback. The FMUF *Controller* can reassign applications of medium/lower importance to the high-priority class when (1) all the applications currently in the high-priority class meet their deadlines while (2) some applications in the low-priority class miss their deadlines. Since the FMUF *Controller* moves applications of medium/lower importance back to the low-priority class when the high-priority class experiences deadline misses it can effectively deal with workload variations caused by application arrivals and changes in application execution times and invocation rates.

5.4.2. Experiment 1: moderate workload

Experiment configuration

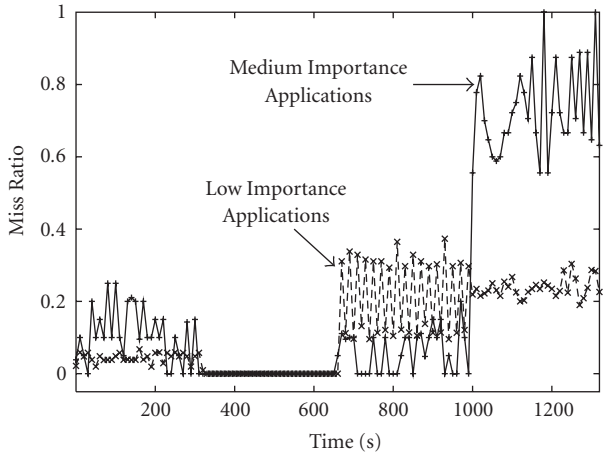
The goal of this experiment configuration was to evaluate RACE's system adaptation capabilities under a moderate workload. This scenario therefore employed two of the five emulated spacecrafts, one emulated ground station, and three periodic applications. One application was initialized to execute in fast survey mode and the remaining two were initialized to execute in slow survey mode. As described in Section 3.1, applications executing in fast survey mode have higher relative importance and resource consumption than applications executing in slow survey mode. Each application is subjected to an end-to-end deadline equal to its period. Table 3 summarizes application periods and the mapping of components/applications onto nodes.

The experiment was conducted over 1,400 seconds, and we emulated variation in operating condition, input workload, and a mode change by performing the following steps. At time $T = 0$ second, we deployed applications one and two. At time $T = 300$ seconds, the input workload for all the application was reduced by ten percent, and at time $T = 700$ seconds we deployed application three. At $T = 1000$ seconds, application three switched mode from slow survey to fast survey. To emulate this mode change, we increased the rate (i.e., reduced the period) of application three by twenty percent. Since each application was subjected to an end-to-end deadline equal to its period, to evaluate the performance of RACE, we monitored the *deadline miss ratio* of all applications that were deployed.

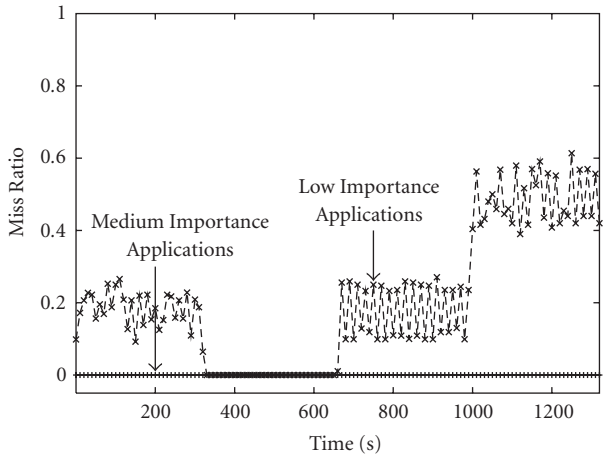
RACE's FMUF *Controller* was used for this experiment since the MMS mission applications described above do not support rate adaptation. RACE is a framework, however, so other adaptation strategies/algorithms, such as HySUCON [11], can be implemented and employed in a similar way. Below, we evaluate the use of FMUF for end-to-end adaptation. Since this paper focuses on RACE—and not the design or evaluation of individual control algorithms—we use FMUF as an example to demonstrate RACE's ability to support the integration of feedback control algorithms for end-to-end adaptation in DRE systems. RACE's FMUF controller was configured with the following parameters: sampling period = 10 seconds, $N = 5$, and *threshold* = 5%.

Analysis of results

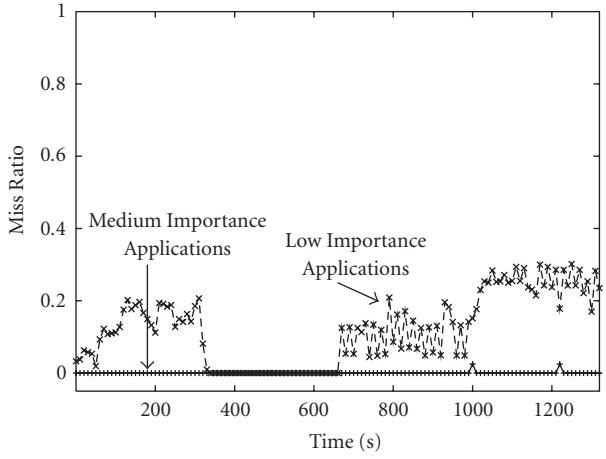
Figures 12(a), 12(b), and 12(c) show the deadline miss ratio of applications when the system was operated under



(a) Baseline (RMS)



(b) MUF Configurator



(c) MUF Configurator + FMUF Controller

Figure 12: Deadline miss ratio under moderate workload.

baseline configuration, with RACE's MUF *Configurator*, and with RACE's MUF *Configurator* along with FMUF *Controller*, respectively. These figures show that under all the three configurations, deadline miss ratio of applications (1) reduced at $T = 300$ seconds due to the decrease in the input work load, (2) increased at $T = 700$ seconds due to the introduction of new application, and (3) further increased at $T = 1,000$ seconds due to the mode change from slow survey mode to fast survey mode. These results demonstrate the impact of fluctuation in input workload and operating conditions on system performance.

Figure 12(a) shows that when the system was operated under the baseline configuration, deadline miss ratio of medium-importance applications (applications executing in fast survey mode) were higher than that of low-importance applications (applications executing in slow survey mode) due to reasons explained in Section 5.4.1. Figures 12(b) and 12(c) show that when RACE's MUF *Configurator* is used (both individually and along with FMUF *Controller*), deadline miss ratio of medium importance applications were nearly zero throughout the course of the experiment. Figures 12(a) and 12(b) demonstrate that the RACE improves QoS of our DRE system significantly by configuring platform-specific parameters appropriately.

As described in [12], the FMUF *Controller* responds to variations in input workload and operating conditions (indicated by deadline misses) by dynamically adjusting the priorities of the low-importance applications (i.e., moving low-importance applications into or out of the high-priority class). Figures 12(a) and 12(c) demonstrate the impact of the RACE's *Controller* on system performance.

5.4.3. Experiment 2: heavy workload

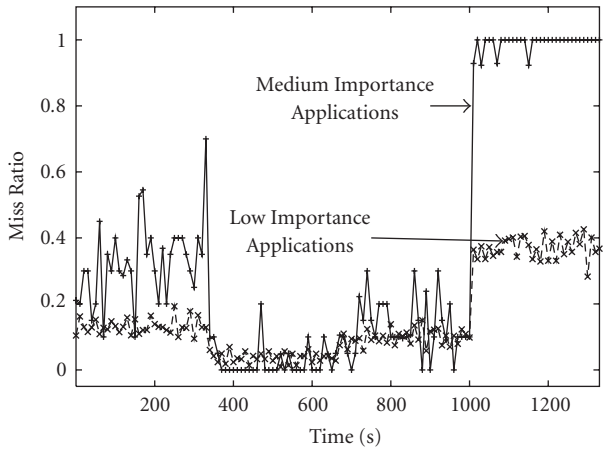
Experiment configuration

The goal of this experiment configuration was to evaluate RACE's system adaptation capabilities under a heavy workload. This scenario, therefore, employed all five emulated spacecrafts, one emulated ground station, and ten periodic applications. Four of these applications were initialized to execute in fast survey mode and the remaining six were initialized to execute in slow survey mode. Table 4 summarizes the application periods and the mapping of components/applications onto nodes.

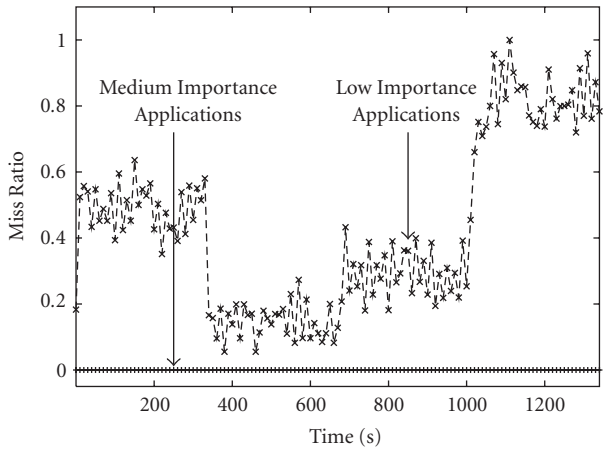
The experiment was conducted over 1,400 seconds, and we emulated the variation in operating condition, input workload, and a mode change by performing the following steps. At time $T = 0$ second, we deployed applications one through six. At time $T = 300$ seconds, the input workload for all the application was reduced by ten percent, and at time $T = 700$ seconds, we deployed applications seven through ten. At $T = 1,000$ seconds, applications two through five switched modes from slow survey to fast survey. To emulate this mode change, we increased the rate of applications two through five by twenty percent. RACE's FMUF controller was configured with the following parameters: sampling period = 10 seconds, $N = 5$, and *threshold* = 5%.

Analysis of results

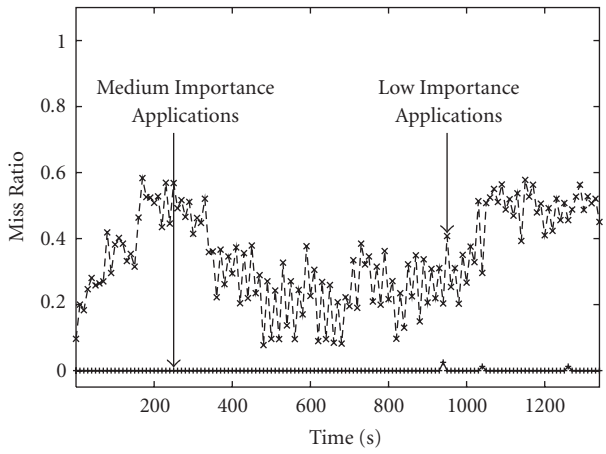
Figure 13(a) shows that when the system was operated under the baseline configuration, the deadline miss ratio of the medium importance applications were again higher than that of the low-importance applications. Figures 13(b) and



(a) Baseline (RMS)



(b) MUF Configurator



(c) MUF Configurator + FMUF Controller

Figure 13: Deadline Miss Ratio under Heavy Workload.

13(c) show that when RACE's MUF *Configurator* is used (both individually and along with FMUF *Controller*), deadline miss ratio of medium importance applications were nearly zero throughout the course of the experiment. Figures 13(a) and 13(b) demonstrate how RACE improves the QoS of our DRE system significantly by configuring platform-specific parameters appropriately. Figures 12(a) and 12(c) demonstrate that RACE improves system performance (deadline miss ratio) even under heavy workload.

These results show that RACE improves system performance by performing adaptive management of system resources there by validating our claim in Section 4.2.3.

5.5. Summary of experimental analysis

This section evaluated the performance and scalability of the RACE framework by studying the impact of increase in number of nodes and applications in the system on RACE's monitoring delay and actuation delay. We also studied the performance of our prototype MMS DRE system with and without RACE under varying operating condition and input workload. Our results show that RACE is a scalable adaptive resource management framework and performs effective end-to-end adaptation and yields a predictable and high-performance DRE system.

From analyzing the results in Section 5.3, we observe that RACE scales as well as the number of nodes and applications in the system increases. This scalability stems from RACE's the hierarchical design of monitors and effectors, which validates our claims in Sections 4.1.2 and 4.1.6. From analyzing the results presented in Section 5.4, we observe that RACE significantly improves the performance of our prototype MMS DRE system even under varying input workload and operating conditions, thereby meeting the requirements of building component-based DRE systems identified in Section 3.2. These benefits result from configuring platform-specific QoS parameters appropriately and performing effective end-to-end adaptation, which were performed by RACE's *Configurators* and *Controllers*, respectively.

6. CONCLUDING REMARKS

Open DRE systems require end-to-end QoS enforcement from their underlying operating platforms to operate correctly. These systems often run in environments where resource availability is subject to dynamic changes. To meet end-to-end QoS in these dynamic environments, open DRE systems can benefit from adaptive resource management frameworks that monitors system resources, performs efficient application workload management, and enables efficient resource provisioning for executing applications. Resource management algorithms based on control-theoretic techniques are emerging as a promising solution to handle the challenges of applications with stringent end-to-end QoS executing in open DRE systems. These algorithms enable adaptive resource management capabilities in open DRE systems and adapt gracefully to fluctuation in resource availability and application resource requirement at run-time.

This paper described the *resource allocation and control engine* (RACE), which is our adaptive resource management framework that provides end-to-end adaptation and resource management for open DRE systems built atop QoS-enabled component middleware. Open DRE systems built using RACE benefit from the advantages of component-based middleware, as well as QoS assurances provided

Table 4: Application configuration under heavy workload.

Application	Component allocation						Period (msec)	Mode
	Spacecraft					Ground station		
	1	2	3	4	5			
1	Communication		Analysis Plasma-sensor	Filter	Compression	Ground	1000	Fast Survey
2	Camera-sensor Compression	Filter Analysis			Communication	Ground	900	Slow Survey
3	Camera-sensor	Plasma-sensor	Communication Compression	Analysis	Filter	Ground	500	Slow Survey
4		Communication	Filter Analysis	Plasma-sensor	Compression	Ground	800	Slow Survey
5	Communication Filter		Camera-sensor	Analysis	Compression	Ground	1200	Slow Survey
6	Analysis	Filter	Communication	Compression	Plasma-sensor	Ground	700	Slow Survey
7	Plasma-sensor	Plasma-sensor	Communication Compression	Analysis	Filter	Ground	600	Fast Survey
8		Communication Filter	Analysis	Plasma-sensor	Compression	Ground	700	Slow Survey
9	Communication Filter		Camera-sensor Plasma-sensor	Analysis	Compression	Ground	400	Fast Survey
10	Compression Filter		Communication Analysis		Plasma-sensor	Ground	700	Fast Survey

by adaptive resource management algorithms. We demonstrated how RACE helped resolve key resource and QoS management challenges associated with a prototype of the NASA MMS mission system. We also analyzed results from performance in the context of our MMS mission system prototype.

Since the elements of the RACE framework are CCM components, RACE itself can be configured using model-driven tools, such as PICML [38]. Moreover, new *InputAdapters, Allocators, Configurators*, and *Controllers* can be plugged into RACE using PICML without modifying its architecture. RACE can also be used to deploy, allocate resources to, and manage performance of, applications that are composed at design-time and run-time.

The lessons learned in building RACE and applying to our MMS mission system prototype thus far include the following.

(i) *Challenges involved in developing open DRE systems.* Achieving end-to-end QoS in open DRE systems requires adaptive resource management of system resources, as well as integration of a range of real-time capabilities. QoS-enabled middleware, such as CIAO/DAnCE, along with the support of DSMLs and tools, such as PICML, provide an integrated platform for building such systems and are emerging as an operating platform for these systems. Although CIAO/DAnCE and PICML alleviate many challenges in building DRE systems, they do not address the adaptive resource management challenges and requirements of open DRE systems. Adaptive resource management solutions are therefore needed to ensure QoS requirements of applications executing atop these systems are met.

(ii) *Decoupling middleware and resource management algorithms.* Implementing adaptive resource management algorithms within the middleware tightly couples the resource management algorithms within particular middleware platforms. This coupling makes it hard to enhance the algorithms without redeveloping significant portions of the middleware. Adaptive resource management frameworks, such as RACE, alleviate the tight coupling between resource management algorithms and middleware platforms and improve flexibility.

(iii) *Design of a framework determines its performance and applicability.* The design of key modules and entities of the resource management framework determines the scalability, and therefore the applicability, of the framework. To apply a framework like RACE to a wide range of open DRE system, it must scale as the number of nodes and application in the system grows. Our empirical studies on the scalability of RACE showed that structuring and designing key modules of RACE (e.g., monitors and effectors) in a hierarchical fashion not only significantly improves the performance of RACE, but also improves its scalability.

(iv) *Need for configuring/customizing the adaptive resource management framework with domain specific monitors.* Utilization of system resources, such as CPU, memory, and network bandwidth, and system performance, such as latency and throughput, can be measured in a generic fashion across various system domains. In open DRE systems, however, the need to measure utilization of domain-specific resources, such as battery utilization, and application-specific QoS metrics, such as the fidelity of the collected plasma data,

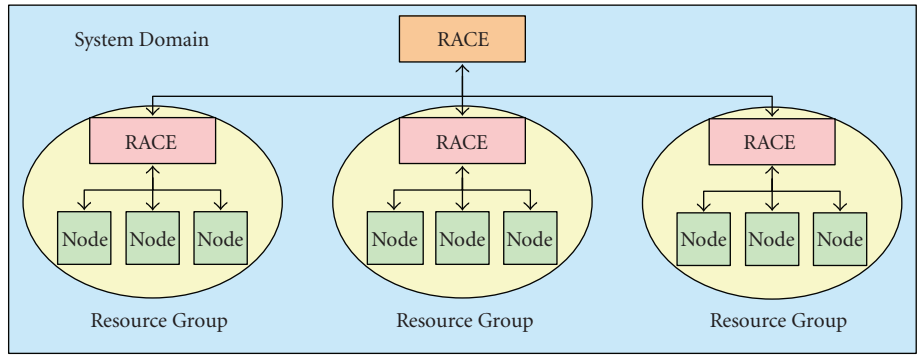


Figure 14: Hierarchical Composition of RACE.

might occur. Domain-specific customization and configuration of an adaptive resource management framework, such as RACE, should therefore be possible. RACE supports domain-specific customization of its *Monitors*. In future work, we will empirically evaluate the ease of integration of these domain-specific resource entities.

(v) *Need for selecting an appropriate control algorithm to manage system performance.* The control algorithm that a *Controller* implements relies on certain system parameters that can be fine-tuned/modified at run-time to achieve effective system adaptation. For example, FMUF relies on fine-tuning operating system priorities of processes hosting application components to achieve desired system adaptation; EUCON relies on fine-tuning execution rates of end-to-end applications to achieve the same. The applicability of a control algorithm to a specific domain/scenario is therefore determined by the availability of these run-time configurable system parameters. Moreover, the responsiveness of a control algorithm and the *Controller* in restoring the system performance metrics to their desired values determines the applicability of a *Controller* to a specific domain/scenario. During system design-time, a *Controller* should be selected that is appropriate for the system domain/scenario.

(vi) *Need for distributed/decentralized adaptive resource management.* It is easier to design, analyze, and implement *centralized* adaptive resource management algorithms that manage an entire system than it is to design, analyze, and implement *decentralized* adaptive resource management algorithms. As a the size of a system grows, however, centralized algorithms can become bottlenecks since the computation time of these algorithms can scale exponentially as the number of end-to-end applications increases. One way to alleviate these bottlenecks is to partition system resources into *resource groups* and employ hierarchical adaptive resource management, as shown in Figure 14. In our future work, we plan to enhance RACE so that a *local* instance of the framework can manage resource allocation, QoS configuration, and run-time adaption within a resource group, whereas a *global* instance can be used to manage the resources and performance of the entire system.

RACE, CIAO, DAnCE, and PICML are available in open source form for download at http://deuce.doc.wustl.edu/.

REFERENCES

[1] S. A. Brandt, S. Banachowski, C. Lin, and T. Bissom, "Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes," in *Proceedings of the 24th IEEE International Real-Time Systems Symposium (RTSS '03)*, pp. 396–407, Cancun, Mexico, December 2003.

[2] P. Martí, C. Lin, S. A. Brandt, M. Velasco, and J. M. Fuertes, "Optimal state feedback based resource allocation for resource-constrained control tasks," in *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS '04)*, pp. 161–172, Lisbon, Portugal, December 2004.

[3] Object Management Group, "Real-time CORBA Specification," OMG Document formal/05-01-04 ed, August 2002.

[4] G. Bollella, J. Gosling, B. Brosgol, et al., *The Real-Time Specification for Java*, Addison-Wesley, Reading, Mass, USA, 2000.

[5] D C. Sharp and W. C. Roll, "Model-based integration of reusable component-based avionics systems," in *Proceedings of the IEEE Real-time and Embedded Technology and Applications Symposium*, Cancun, Mexico, December 2003.

[6] D. C. Sharp and W. C. Roll, "Model-Based integration of reusable component-based avionics system," in *Proceedings of the Workshop on Model-Driven Embedded Systems in RTAS 2003*, Washington, DC, USA, May 2003.

[7] D. Suri, A. Howell, N. Shankaran, et al., "Onboard processing using the adaptive network architecture," in *Proceedings of the 6th Annual NASA Earth Science Technology Conference*, College Park, Md, USA, June 2006.

[8] P. K. Sharma, J. P. Loyall, G. T. Heineman, R. E. Schantz, R. Shapiro, and G. Duzan, "Component-based dynamic QoS adaptations in distributed real-time and embedded systems," in *Proceedings of the International Symposium on Distributed Objects and Applications (DOA '04)*, vol. 3291, pp. 1208–1224, Agia Napa, Cyprus, October 2004.

[9] C. Lu, X. Wang, and X. D. Koutsoukos, "Feedback utilization control in distributed real-time systems with end-to-end tasks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 6, pp. 550–561, 2005.

[10] X. Wang, D. Jia, C. Lu, and X. D. Koutsoukos, "Decentralized utilization control in distributed real-time systems," in *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS '05)*, pp. 133–142, Miami, Fla, USA, December 2005.

[11] X. D. Koutsoukos, R. Tekumalla, B. Natarajan, and C. Lu, "Hybrid supervisory control of real-time systems," in *Proceedings*

of the 11th IEEE Real-time and Embedded Technology and Applications Symposium, San Francisco, Calif, USA, March 2005.

[12] Y. Chen and C. Lu, "Flexible maximum urgency first scheduling for distributed real-time systems," *Tech. Rep. WUCSE-2006-55*, Washington University, St. Louis, Mo, USA, October 2006.

[13] N. Shankaran, X. D. Koutsoukos, D. C. Schmidt, Y. Xue, and C. Lu, "Hierarchical control of multiple resources in distributed real-time and embedded systems," in *Proceedings of the 18th Euromicro Conference on Real-Time Systems (ECRTS '06)*, pp. 151–160, Dresden, Germany, July 2006.

[14] X. Wang, C. Lu, and X. D. Koutsoukos, "Enhancing the robustness of distributed real-time middleware via end-to-end utilization control," in *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS '05)*, pp. 189–199, Miami, Fla, USA, December 2005.

[15] D. C. Schmidt and S. D. Huston, *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*, Addison-Wesley, Reading, Mass, USA, 2002.

[16] Object Management Group, "Common Object Request Broker Architecture Version 1.3," OMG Document formal/2004-03-12 ed, March 2004.

[17] E. Pitt and K. McNiff, *Java RMI : The Remote Method Invocation Guide*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[18] B. Ravindran, L. Welch, and B. Shirazi, "Resource management middleware for dynamic, dependable real-time systems," *Real-Time Systems*, vol. 20, no. 2, pp. 183–196, 2001.

[19] L. R. Welch, B. A. Shirazi, B. Ravindran, and C. Bruggeman, "DeSiDeRaTa: QoS management technology for dynamic, scalable, dependable real-time systems," in *Proceedings of the IFACs 15th Workshop on Distributed Computer Control Systems (DCCS '98)*, IFAC, Como, Italy, September 1998.

[20] D. Fleeman, M. Gillen, A. Lenharth, et al., "Quality-based adaptive resource management architecture (QARMA): a CORBA resource management service," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS '04)*, vol. 18, pp. 1623–1630, Santa Fe, NM, USA, April 2004.

[21] C. D. Gill, *Flexible scheduling in middleware for distributed rate-based real-time applications, Ph.D. dissertation*, St. Louis, Mo, USA, 2002.

[22] K. Bryan, L. C. DiPippo, V. Fay-Wolfe, et al., "Integrated CORBA scheduling and resource management for distributed real-time embedded systems," in *Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium (RTAS '05)*, pp. 375–384, San Francisco, Calif, USA, March 2005.

[23] V. F. Wolfe, L. C. DiPippo, R. Bethmagalkar, et al., "RapidSched: static scheduling and analysis for real-time CORBA," in *Proceedings of the 4th International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS '99)*, pp. 34–39, Santa Barbara, Calif, USA, January 1999.

[24] J. W. Liu, J. Redondo, Z. Deng, et al., "PERTS: a prototyping environment for real-time systems," *Tech. Rep. UIUCDCS-R-93-1802*, University of Illinois at Urbana-Champaign, Champaign, Ill, USA, 1993.

[25] Object Management Group, "CORBA Components," OMG Document formal/2002-06-65 ed, June 2002.

[26] Sun Microsystems, "Enterprise JavaBeans specification," August 2001, http://java.sun.com/products/ejb/docs.html.

[27] A. Thomas, "Enterprise JavaBeans technology," prepared for Sun Microsystems, December 1998, http://java.sun.com/products/ejb/white_paper.html.

[28] J. A. Zinky, D. E. Bakken, and R. E. Schantz, "Architectural support for quality of service for CORBA objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, pp. 55–73, 1997.

[29] R. Schantz, J. Loyall, M. Atighetchi, and P. Pal, "Packaging quality of service control behaviors for reuse," in *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC '02)*, pp. 375–385, Washington, DC, USA, April-May 2002.

[30] P. Manghwani, J. Loyall, P. Sharma, M. Gillen, and J. Ye, "End-to-end quality of service management for distributed real-time embedded applications," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS '05)*, p. 138a, Denver, Colo, USA, April 2005.

[31] J. S. Kinnebrew, N. Shankaran, G. Biswas, and D. C. Schmidt, "A decision-theoretic planner with dynamic component reconfiguration for distributed real-time applications," in *Proceedings of the 21th National Conference on Artificial Intelligence*, Boston, Mass, USA, July 2006.

[32] J. Hatcliff, X. Deng, M. B. Dwyer, G. Jung, and V. P. Ranganath, "Cadena: an integrated development, analysis, and verification environment for component-based systems," in *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, pp. 160–172, Portland, Ore, USA, May 2003.

[33] J. A. Stankovic, R. Zhu, R. Poornalingam, et al., "VEST: an aspect-based composition tool for real-time systems," in *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '03)*, pp. 58–69, Toronto, Canada, May 2003.

[34] A. Lédeczi, A. Bakay, M. Maróti, et al., "Composing domain-specific design environments," *Computer*, vol. 34, no. 11, pp. 44–51, 2001.

[35] S. Kodase, S. Wang, Z. Gu, and K. G. Shin, "Improving scalability of task allocation and scheduling in large distributed real-time systems using shared buffers," in *Proceedings of the 9th Real-time/Embedded Technology and Applications Symposium (RTAS '03)*, pp. 181–188, San Francisco, Calif, USA, May 2003.

[36] P. López, J. L. Medina, and J. M. Drake, "Real-time modelling of distributed component-based applications," in *Proceedings of the 32nd Euromicro Conference on Software Engineering and Advanced Applications (Euromicro-SEAA '06)*, pp. 92–99, Dubrovnik, Croatia, August 2006.

[37] M. G. Harbour, J. J. G. García, J. C. P. Gutiérrez, and J. M. D. Moyano, "MAST: modeling and analysis suite for real time applications," in *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS '01)*, pp. 125–134, Delft, The Netherlands, June 2001.

[38] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, and D. C. Schmidt, "A platform-independent component modeling language for distributed real-time and embedded systems," in *Proceedings of the11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS '05)*, pp. 190–199, San Francisco, Calif, USA, March 2005.

[39] S. Bagchi, G. Biswas, and K. Kawamura, "Task planning under uncertainty using a spreading activation network," *IEEE Transactions on Systems, Man, and Cybernetics Part A*, vol. 30, no. 6, pp. 639–650, 2000.

[40] S. Curtis, "The magnetospheric multiscale mission—resolving fundamental processes in space plasmas," *Tech. Rep. NASA/TM-2000-209883*, NASA STI/Recon, Greenbelt, Md, USA, December 1999.

[41] N. Wang, D. C. Schmidt, A. Gokhale, et al., "QoS-enabled middleware," in *Middleware for Communications*,

Q. Mahmoud, Ed., pp. 131–162, John Wiley & Sons, New York, NY, USA, 2004.

[42] G. Deng, J. Balasubramanian, W. Otte, D. C. Schmidt, and A. Gokhale, "DAnCE: a QoS-enabled component deployment and configuration engine," in *Proceedings of the 3rd International Working Conference on Component Deployment (CD '05)*, vol. 3798 of *Lecture Notes in Computer Science*, pp. 67–82, Grenoble, France, November 2005.

[43] D. de Niz and R. Rajkumar, "Partitioning bin-packing algorithms for distributed real-time systems," *International Journal of Embedded Systems*, vol. 2, no. 3-4, pp. 196–208, 2006.

[44] Object Management Group, "Deployment and Configuration Adopted Submission," OMG Document mars/03-05-08 ed, July 2003.

[45] B. Buck and J. K. Hollingsworth, "API for runtime code patching," *International Journal of High Performance Computing Applications*, vol. 14, no. 4, pp. 317–329, 2000.

[46] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: exact characterization and average case behavior," in *Proceedings of the Real-Time Systems Symposium (RTSS '89)*, pp. 166–171, Santa Monica, Calif, USA, December 1989.

[47] I. Molnar, "Linux with real-time pre-emption patches," September 2006, http://www.kernel.org/pub/linux/kernel/projects/rt/.

[48] Object Management Group, "Light Weight CORBA Component Model Revised Submission," OMG Document realtime/03-05-05 ed, May 2003.

[49] D. B. Stewart and P. K. Khosla, "Real-time scheduling of sensor-based control systems," in *Real-time Programming*, W. Halang and K. Ramamritham, Eds., pp. 139–144, Pergamon Press, Tarrytown, NY, USA, 1992.

Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 285160, 10 pages
doi:10.1155/2008/285160

Research Article

Reconfiguration Management in the Context of RTOS-Based HW/SW Embedded Systems

Yvan Eustache and Jean-Philippe Diguet

LESTER Lab, CNRS/UBS, 56100 Lorient, France

Correspondence should be addressed to Yvan Eustache, yvan.eustache@univ-ubs.fr

Received 1 April 2007; Revised 24 August 2007; Accepted 19 October 2007

Recommended by Alfons Crespo

This paper presents a safe and efficient solution to manage asynchronous configurations of dynamically reconfigurable systems-on-chip. We first define our unified RTOS-based framework for HW/SW task communication and configuration management. Then three issues are discussed and solutions are given: the formalization of configuration space modeling including its different dimensions, the synchronization of configuration that mainly addresses the question of task configuration ordering, and the configuration coherency that solves the way a task accepts a new configuration. Finally, we present the global method and give some implementation figures from a smart camera case study.

1. INTRODUCTION

1.1. Self-adaptive RTOS-controlled SoC

Upcoming embedded systems will implement complex multistandard multimode applications competing for resources within a heterogeneous architecture. One of the most important challenges in this context is the tradeoff between flexibility needed for fast design of mass products, performance, power management, and quality of service (QoS). This tradeoff can only be partially obtained at design time. CPU and communication loads vary with tasks' changeable execution times. This can be due to network access conditions, data values, architecture hazards (e.g., cache miss), or user choices.

One of the most promising directions to deal with such constraints is the reconfigurable architecture that can be tuned accordingly to resource requirements.

Embedded systems, in the domain of wireless networks and multimedia applications, manage concurrent applications with fluctuating loads. It means that reconfiguration requests are not usually synchronized with running tasks and can happen at any time independently of data and control flows of concurrent applications. In such a context, one of the main tedious problems is to guarantee a safe reconfiguration synchronization that takes care of data dependencies and algorithm coherency. The resulting complexity of the system control requires RTOS services for synchronization, communication, and concurrency management. In addition to that, extensions of RTOS services are needed for configuration management in order to provide two kinds of capabilities: (i) transparent hardware and software communication services and (ii) configuration management in order to facilitate the design of complex heterogeneous systems. Another important RTOS advantage is the abstraction to increase portability and to facilitate reuse of code and repartitioning.

The objective of this work is to formalize and implement an abstraction layer suitable for usual RTOS in order to handle hardware and software communications and configuration management. In our experiments, we use μCOSII which is implemented on different targets such as NIOS, PowerPC, and MicroBlaze cores on Altera and Xilinx devices, respectively.

The background of this paper is our solution for implementing self-adaptive systems. The foreground is the tedious question of configuration switching in the context of hardware and software implementations. This point is generally simplified, whereas it raises in practice complex questions about configuration granularity and synchronization.

The rest of the paper is organized as follows. Section 2 provides a description of our adaptive system model including the HW/SW unified interface and the configuration model. In Section 3, we present the main issues and solutions

involved with the reconfiguration and propose the new concept of configuration granularity to manage a coherent configuration. Finally, a smart camera case study is introduced in Section 4 to validate our approach.

1.2. Related works

In this section, we survey a selection of related works in the area of RTOS-based reconfigurable SoC. Firstly a lot of work has been produced in the domain of adaptive architectures. Different techniques have been introduced for clock and voltage scaling [1], cache resources [2], and functional units [3] allocations. These approaches can be classified in the category of local configurations based on specific aspects. Our aim is to add global configuration management including both algorithmic and architectural aspects.

Secondly RTOS for HW management has been recently introduced. Proofs of concepts are exhibited in [4, 5]. These experiments show that RTOS level management of reconfigurable architectures can be considered as being available from a research perspective. In [5], the RTOS is mainly dedicated to the management (placement/communication) of HW tasks. Run-time scheduling and 1D and 2D placement of HW tasks' algorithms are described in [6]. In [4], the OS4RS layer is an OS extension that abstracts the task implementation in hardware or software; the main contribution of this work is the communication API based on message passing where communication between HW and SW tasks is handled with a hardware abstraction layer. Moreover, the heterogeneous context switch issue is solved by defining *switching points* in the data flow graph, but no computation details are given. In [7], abstraction of the CPU/FPGA component boundary is supported by HW thread interface concept, and specific RTOS services are implemented in HW. In [8], more details are given about a network-on-chip communication scheme.

The current state of the art shows that adaptive, reconfigurable, and programmable architectures are already available, but there is no real complete solution proposed to guarantee safe configuration management. Some concepts are presented but no systematic solutions are proposed.

2. ADAPTIVE SYSTEM MODEL

2.1. System model

The targeted system is a multitask heterogeneous system composed by an embedded processor for the SW tasks and accelerators implementing HW tasks; both are masters on a communication medium, which is a shared bus or a network-on-chip. Such a system improves performances since several data-dependent or independent tasks can run concurrently. Communication between data-dependent tasks is based on shared memories for data and RTOS message passing services for events and address notification (e.g., mailbox and message queue). Data-dependent tasks are gathered in a cluster; each cluster can have one or more source tasks and sink tasks; a minimum cluster is reduced to a single task. At the highest level, applications can be composed by several clusters.

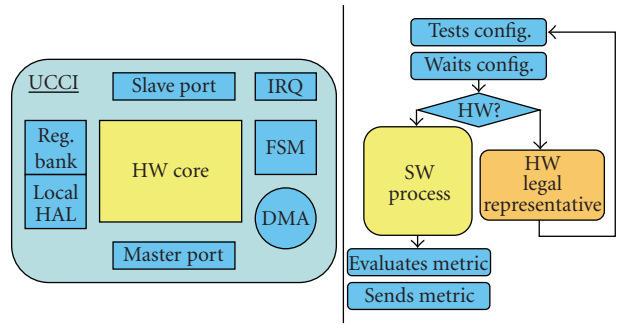


Figure 1: Unified communication and configuration interface.

We define the model of the system as a directed acyclic graph where a task is represented by a node and the shared memory by an edge connecting two nodes (details are given in Section 3.3.3).

2.2. UCCI concept

2.2.1. OS services access for HW tasks

Abstract HW/SW interface for an embedded system-on-chip is an important topic. It allows communications between tasks independently of their implementations. In our adaptive strategy, communications are not limited to data since local data (task context: recursive data, counters), configuration (pointers, configuration ID), and metrics are also exchanged. Thus, task communications are encapsulated within a unified interface called UCCI (see Figure 1) that handles configuration, data, and control communications in such a way that usual SW and HW implementations can be directly instantiated in the new framework.

Each HW task has a legal representative (LR) which is a lightweight SW task that maintains communications between RTOS and tasks implemented in hardware. Local data can be read (resp., written) by the LR in case of HW to SW (resp., SW to HW) migration or by the HW UCCI in case of HW to HW migration. Basically, a context transfer is equivalent to a data transfer performed between reconfigurations in case of HW to SW or HW to HW migrations.

The UCCI implementation imposes some constraints regarding I/O format. For instance, local data are based on identical stacks for all HW and SW versions. Moreover, some SW routines have been added for data format adaptation; the more used one is the bit/byte conversion routine required when SW tasks accede to binary images where pixel is coded as bit for bandwidth and area optimization.

2.2.2. Tasks communication

Communications with an SW task are involved to call the intertask communication RTOS services. Two kinds of operation can be used: post and pend communications. For SW → SW communications, traditional communication services are used, whereas in HW ↔ SW communications, the LR task

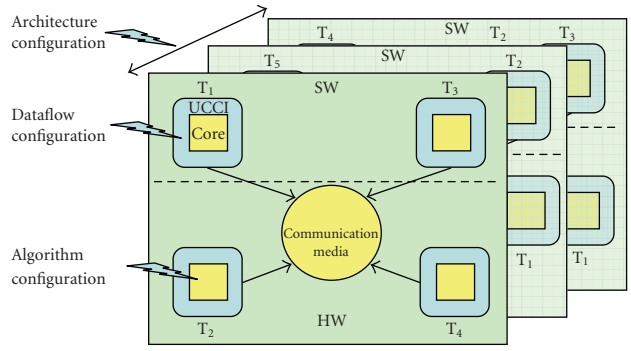


Figure 2: Configuration space model.

is mainly used to handle the HW pend request. Postservices are supported directly by the interrupt service routine (ISR). If using RTOS communication services appears to be essential to abstract the communication between tasks, the overhead due to the HW to LR communication via the ISR and the involved context switches can be important for an application using mainly message passing. Communication overhead between two HW tasks via RTOS communication service has been drastically reduced while delegating post and pend communication management to HW UCCI which provides message passing services. Thus, a first task can write directly the message in a specific register of a second HW task. "Pend" is represented by a wait state until a message is written in the register. Other services such as mutex, semaphore, and flag services are provided by the RTOS and not implemented in UCCI since no distributed implementation was justified. However, independently of our work, some specific HW implementations of mutex, for instance, can be added to improve performances as shown in [9].

UCCI concept brings efficient communication between HW tasks. However, control is added to the HW task to manage direct or OS message passing according to the implementation of the other tasks (more details are given in [10]).

2.3. Configuration space modeling

Based on our analysis, we propose to partition the configuration space into three levels (see Figure 2).

Algorithm reconfiguration targets the functionality of a task. A new configuration may occur to improve performance, QoS, or energy. In this paper, we consider that each configurable task implements different algorithms with various complexity/performance tradeoffs; it can be, for instance, different lowpass filters based on an average of a variable number of images (2, 3, 4), image processing based on various mask sizes, or image thresholding based on static or dynamic thresholds.

Data flow reconfiguration represents the intertask communication scheme at the application level. The data flow configuration mainly impacts task UCCI and principally communication parameters (read/write addresses) located in the HAL. Such a configuration is used, for instance, when a task is inserted in or removed from the application. Since data flow and algorithm modifications are very close in terms of frequency and management, we gather these two kinds of configurations under the term of algorithm configuration in the rest of the paper.

Architecture configuration is the arrangement of HW and SW task implementations for a set of tasks with associated selected algorithms. Architecture reconfiguration may occur to migrate an SW task to an HW implementation. In opposition, the system may switch a task from HW to SW. However, this possibility of task migration leads to important time and resource overheads (HW ↔ SW context store/load, bitstream generation, and download). In consequence, the modification of an architecture cannot occur at the application frequency. A tradeoff has to be solved between the rate of architecture reconfiguration and the gain it brings. We assume that HW dynamic configuration is available, but we do not address this point which is very technology-dependent (e.g., Xilinx bitstream loading). Besides, we have implemented our smart camera prototype on an Altera StratixII FPGA, which does not allow for dynamic reconfiguration; so all tasks are implemented at design time. Thus, HW task activation is implemented with a clock gating control.

Finally, reconfiguration raises the questions of algorithm/architecture selection, task activation control, communication scheme setup, and HW clock management. So, a configuration can be modeled by a unique configuration identifier (CID), which represents a point in a three-dimensional space, namely, the combination of a set of algorithms, a data flow, and an architecture.

2.4. Configuration management

2.4.1. Of/online configuration management

Embedded systems are characterized by scarce resources in terms of energy, memory, and processing to be used efficiently. Our low-cost configuration management reduces the time and memory overheads separating offline selection and adaptation at run time. The first step of the management is the offline selection of potential configurations. The space of configuration (algorithm, data flow, and architecture configurations) is reduced to a unit of best configurations, in terms of performance, consumption, and QoS. An incertitude due to environment fluctuations or internal risks (cache misses, bus collisions, etc.) is also taken into account. In addition, the system adapts itself to its configuration by selecting online the best configuration within the preselected configuration finite space. Thus, no generation of configuration is processed and adaptation time is minimized.

2.4.2. Local/global separation of concerns

Online adaptation issue encompasses different aspects that drive the implementation of the reconfiguration decision control. The first relevant point is locality. A reconfiguration can be decided at the application level or system level. Based on application-specific data, a local decision provides a short reaction delay and metrics to compute the QoS. However, some decisions must be considered globally when a tradeoff

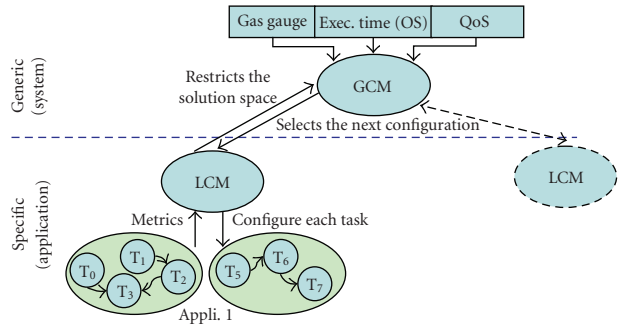


Figure 3: Application versus system configuration management.

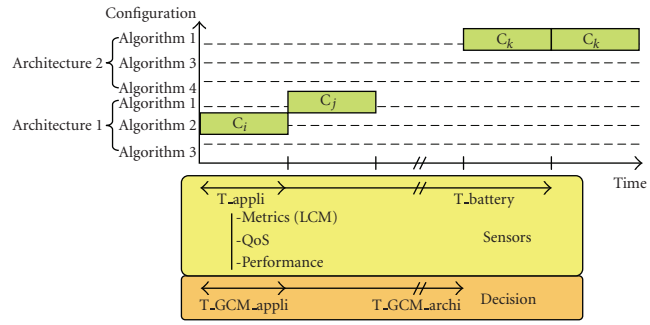


Figure 4: Configuration management timing.

has to be found over the complete system between power consumption and computation efficiency. Another pertinent point is the complexity of the decision implementation. In the context of embedded systems, only low-cost solutions must be considered. To cope with these issues, we implement a two-step configuration management (see Figure 3). The local configuration manager (LCM) is the first level; it is application-specific. The second one is the global configuration manager (GCM); it is generic and so independent of embedded applications.

LCM is in charge of the algorithm configuration for all the tasks of the application it controls. Given results from application tasks, it first transforms application-specific metrics into normalized QoS metrics for the GCM. This is a kind of "application sensors" for the global manager. Secondly, it selects the minimum algorithm configuration to be considered by the GCM. Finally, the LCM controls the application configuration while applying the GCM decisions.

GCM is in charge of global system parameters (e.g., I/O data rates) and HW/SW implementation decisions. It collects information for configuration decisions from sensors such as CPU load from the OS, application-specific QoS from LCMs, and battery level and power from the gas gauge controller or from estimators when no measures are available. The GCM decides upon the new system configuration according to user requirements (system references) and configuration solutions issued from the local managers' design space restrictions.

LCM and GCM as new RTOS services

LCM is application-specific; it can be interpreted as a kind of driver or abstraction layer to get algorithmic configuration wishes issued from application tasks and to send new configurations to application tasks. A GCM is a new RTOS service comparable to scheduling or resource allocation.

In practice, LCM and GCM can be implemented as high-priority tasks within an existing RTOS (e.g., μCos). In such a scheme, each LCM pends on a message queue where application metrics are loaded and reacts according to its configuration management policy. The GCM wakes up each time a configuration algorithm is notified and reacts according to its configuration management policy. A simple policy consists in waiting all notifications from application tasks or LCM, and it decides upon the new configuration compliant with user requirements.

2.4.3. Configuration management timing

Configuration frequency is also an important issue regarding configuration overhead. As shown in Figure 4, application execution time is defined as the base period; after each execution, the LCM receives metrics from tasks and collects QoS and execution time. Sensors acquisition (e.g., battery level) is managed with lower frequencies and can use linear estimators [11] to alleviate the measure overhead. In opposition, battery sensor, due to its important overhead of resources (control, computation time, consumption), is monitored with a lower frequency. At the application period, the GCM selects suitable algorithm configuration according to the actual architecture. The GCM is not synchronized with LCMs. In case of a single application, its period is the application one with an important restriction due to the configuration overhead of current reconfigurable SoC (FPGA). Thus, when architecture reconfiguration occurs, a *provision* period is computed; during this period, the GCM can only allow algorithmic reconfigurations. This two-step configuration management is very flexible and allows a short reaction delay for algorithm adaptation compared to the architectural modification.

3. SYNCHRONIZATION AND COHERENCY

Adaptation process involves three main steps. First, the system has to be able to monitor its environment. For example, in our smart camera case study, the LCM locally uses the number of objects and the image resulting noise (number of isolated white pixels) to configure/select algorithms. Moreover, the LCM provides GCM with the application QoS being computed as the difference between object position based on image processing and estimations based on linear extrapolations. Other global information used is the battery level and task execution times.

Based on this set of information, the system (GCM) takes a decision (e.g., selection of a more robust 2D low-pass filter). Finally, it must guarantee a safe transition from an old to a new configuration. We propose to partition it into three subissues: local data management, communication

dependencies, and functional dependencies that guarantee application coherency.

3.1. Issues

Local data management: architectural reconfiguration raises the question of management of local data. Close to the SW context switch, HW task preemption involves saving processing data. It allows the task to resume its process with another implementation (*HW/SW* switches or HW move) with identical conditions. The main difficulty is to define a context model and store/load management.

Communication dependency: the communication topology can be modified according to data flow reconfiguration. So, the reconfiguration may be performed in a particular order compliant with communication dependencies to avoid memory or mail queue address mismatches. It means that a sender task will be configured before the task waiting for its data.

Functional dependency: configuration may also involve functional coherency management. However, if communication dependency is a run-time issue, management of functional coherencies can be processed offline. Such a modification may affect three process characteristics of a task: the functionality (e.g., compression/decompression algorithm), the amount of data (e.g., image size), and the data type (e.g., bit or byte processing). Thus, it may involve configuration dependencies between communicating tasks. Algorithm modification of configuration-dependent tasks involves defining configuration management to insure coherent data exchanges. On the other side, some algorithm configurations involve no configuration dependencies between tasks (e.g., the coefficient modification in a filtering task). In this case, the modification may be accepted at any time.

Finally, the problem of the coherency management is"how to be sure that a task receives sufficient homogeneous data from other tasks to complete its process and to provide sufficient homogeneous data to the downstream tasks before accepting a reconfiguration." This issue has to be solved regarding the complete system from source to sink tasks of all clusters and for each configuration.

3.2. Solutions

To solve the issues described above, we mainly target two points: synchronization which corresponds to the timing order to reconfigure the tasks, and algorithm coherency which corresponds to the integrity of data exchanged between tasks being reconfigured.

3.2.1. Local data management

Local data management is a key point in the reconfigurable HW system research topics and mainly for the preemption of HW tasks. Usually, it is solved by considering that HW tasks are not preemptible, and so no context needs to be saved. Walder et al. target this issue in [12] and indicate that such services must be included, but no more details are given. This issue is strongly technology-dependent. In our application case study, we define task local data context (LDC) as a task-specific stack containing resident computation data and data pointers, which are unique for both HW and SW tasks. For SW to HW and HW to SW reconfigurations, contexts can be communicated directly via the LR. Regarding HW to HW reconfigurations, we use the same scheme as that implemented for data communications: shared memories and message (addresses) passing. Thus, contexts are saved in a context memory reloaded after each configuration. LDC is automatically loaded after a reconfiguration as the initialization step before running.

3.2.2. Synchronization

Online synchronization issue can be solved with configuration diffusion mechanisms. Our concept is based on the utilization of existing communication channel (direct HW to HW configuration or with RTOS communication services). Thus, we define a two-step strategy. Firstly, the configuration manager sends the CID to all source tasks of each cluster through a multicast diffusion. Secondly, the CID is propagated gradually from the source to the sink tasks over data channels. More details are given in Section 3.4

With such diffusion principles, we guarantee that all tasks will be configured starting with the source tasks. Note that propagation principle increases the HW control. Indeed, HW tasks receive and send the configuration ID via OS communication services or directly according to the implementation of other tasks. Moreover, it has also to inform its LR when it receives directly the CID from an HW task.

3.2.3. Algorithm configuration coherencies

To solve configuration coherency, we introduce the new concept of configuration granularity that guarantees for each task consistent production of data. For each configuration and for each task, it can be computed offline, and it corresponds to a task static parameter.

3.3. Configuration coherency

3.3.1. Granularity concept

By providing algorithm configuration capability, adaptive system has the objective to handle safe transitions between such configurations. In the case of two configurations involving coherencies, the adaptation manager has to be assured that tasks reach a synchronization point before they are reconfigured; namely, enough data must be available or produced before a configuration can be accepted. To solve this issue, we introduce the concept of configuration granularity, *Gc*.

Definition 1. *Gc* corresponds to the data quantity a task has to produce for a given output before it accepts a new configuration. It guarantees that sink tasks will receive enough data to complete their cycle before switching to another configuration.

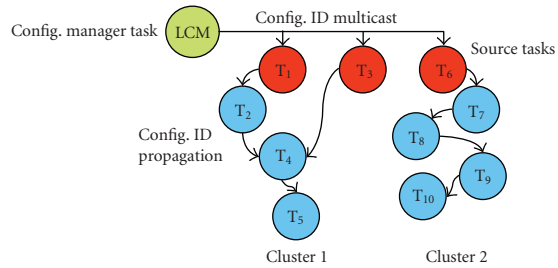


Figure 5: Synchronization with multicasting and propagation.

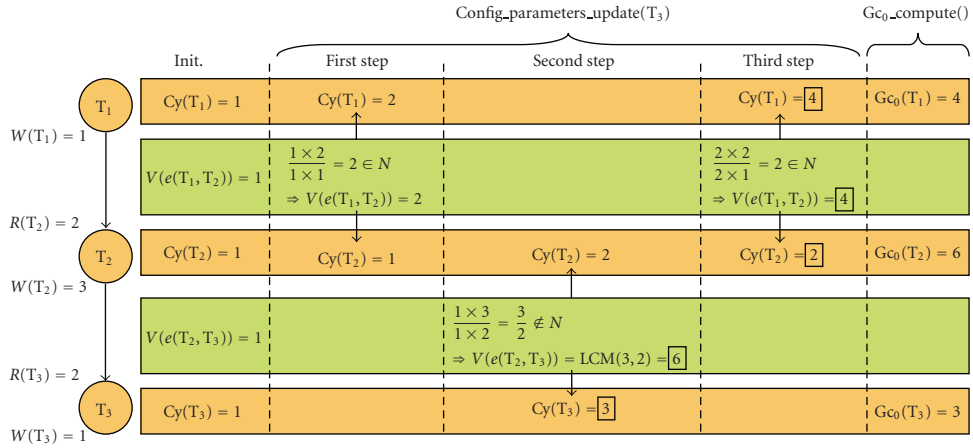


Figure 6: Gc_0 computation example.

3.3.2. Reconfiguration process

In our model, data communication is based on shared memories; it means that local data only deals with counters, residual, or recursive data. These are two distinct issues necessary for reconfiguration success, but they are driven by configuration granularity constraints. Thus, the configuration process is as follows:

(1) CID reception,
(2) no change until Gc is not reached,
(3) save local data context (if necessary),
(4) propagate CID (if successors),
(5) task configuration:

(i) HW↔ SW or HW→ HW migration, LDC loading,
(ii) datapath modification: HAL loading,
(iii) HW task suppression: bitstream removing or clock stopping, corresponding to SW task in SW state,
(iv) HW task insertion: bitstream loading, clock starting, or algorithm configuration, corresponding to SW task in LR state.

3.3.3. Graph model for a given configuration

Let $G_C(N, E)$ be the system configured with the configuration C, where a node $n \in N$ and a directed weighted edge $e \in E$ connecting two nodes represent a task and a shared memory between two tasks, respectively. The weight of the edge corresponds to the size of the minimum shared memory.

The configuration model is presented as follows.

(1) $nb(\text{pred}(n))$: predecessor number of node n.
(2) $\text{pred}(n)[i]$: predecessor node connected to input i of n.
(3) $\text{succ}(n)[i]$: successor node connected to output i of n.
(4) $R(n, o)$: amount of data produced by the predecessor node o; a node n reads per task cycle.
(5) $W(m, n)$: amount of data the precedent node m produces per task cycle.
(6) $\text{Cy}(n)$: number of task cycles needed by the node n to produce and consume data.
(7) $e(\text{pred}(n)[i], n)$: edge connecting the node n and its predecessor $\text{pred}(n)$ with the ith output of the node n. It represents the memory between $\text{pred}(n)$ and n.
(8) $V(e(\text{pred}(n)[i], e))$: value of the edge connecting the node n and its predecessor node $\text{pred}(n)$. The weight of the edge corresponds to the shared memory size.
(9) $\text{Gc}_0^n[j]$: configuration granularity of the node n on the edge connected to the jth output.

For a given configuration, $R(\cdots)$ and $W(\cdots)$ are constant characteristics of a node, whereas $\text{Cy}(n)$, $V(e(n, \text{pred}(n)[i]))$, and $\text{Gc}_0^n[j]$ must be homogenized.

Definition 2. Gc_0 can be deduced from the number of process cycles that a task needs to produce sufficient data

according to the given output writing constraints (the constant quantity of data produced per cycle). For a multioutput task, the configuration granularity of each output respects the following equation:

$$\forall i, \quad \frac{\mathrm{Gc}_0^n[i]}{W(n, \mathrm{succ}(n)[i])} = \mathrm{Cy}(n). \tag{1}$$

3.3.4. SDF analogy

Our system can be modelized by a synchronous data flow (SDF) [13], where one SDF model corresponds to our graph model for a given configuration. On one side, the SDF graph traveling goal is to find a periodic admissible schedule, meaning that tasks will be scheduled to run only when data are available and finite amount of data is required. On the other side, our goal is to find a homogeneous data computation configuration, meaning the cycle number of tasks to produce sufficient data before accepting a new configuration. The number of occurrences of a node then corresponds to the cycle number of a task in our graph model. Our contribution is to find the minimum value of configuration granularity $\mathrm{Gc}_0^n[i]$ for each output i of tasks n. Equation (1) gives the relation between $\mathrm{Gc}_0^n[i]$ and $\mathrm{Cy}(n)$.

3.3.5. Granularity computation

For a safe transition (no data famine and no blocking behavior), configuration granularity is computed for each coherency-dependent couple of producer-consumer tasks and for all configurations. Its statical characteristics allow us to do it offline. Granularity computation is realized for each path from all sink tasks. The result depends not only on consumer task characteristics but also on characteristics of all tasks composing the cluster.

We have developed an algorithm to compute the configuration granularity in three steps. First, we assume that the system is consistent in the sense of SDF formalism where for a connected SDF graph with s nodes and topology matrix Γ, $\mathrm{rank}(\Gamma) = s - 1$ is a necessary condition for a periodic admissible sequential schedule to exist. Real HW/SW applications are consistent and our solution can be applied.

We first initialize the cycle number, the configuration granularity, and the weight of the edges to one (at least one cycle for computation, one dataset produced, and one cycle for reconfiguration). The algorithm then homogenizes the edge weight and the cycle number of each node according to their read and write constraints. Finally, it computes the configuration granularity for each node output.

The Config_Parameter_Update() function travels each branch of the graph from the sink to the source tasks. The computation core checks if data quantity produced by the precedent task $\mathrm{pred}(n)$ is sufficient and corresponds to an integer value compared with the consumption of the task n. Otherwise, the least common multiplier between these two values is computed. The cycle number of the tasks $\mathrm{pred}(n)$ and n is then updated according to the read and write constraints, $R(n, \mathrm{pred}(n)[i])$ and $W(\mathrm{pred}(n)[i], n)$. Finally, modifications of the number of cycles are propagated to the upstream and downstream tasks by the recursive characteristic of the algorithm. Applying the Gc_0_Compute function for each task output, we guarantee the minimum exchange of data between tasks before a new configuration can be taken into account without data loss and blocking states.

Algorithm 1 is launched for all sink tasks and repeated until no modifications are observed.

It is applied on a simple example in Figure 6. In a cluster, three data-dependent tasks communicate in a single path via shared memories. The tasks are also configuration-dependent. In consequence, the configuration granularity is computed. T_1, T_2, and T_3 have to produce, respectively, 4, 6, and 3 data before accepting a reconfiguration.

With its recursive characteristic, Algorithm 1 targets also multipath graph. Each path is traveled from the sink to the source tasks.

An additional termination condition is added due to hardware limitation. Thus, intertask memory size is bounded by the resource limitations involved by embedded systems characteristics.

3.3.6. Application rescaling

While the configuration granularity Gc_0 represents the minimum protection against famine and blocking behavior, application constraints may impose rescaling of the configuration granularity. Rescaling is processed task by task according to application and user requirements:

$$\forall n, \quad \mathrm{Gc}^n = K^n \times \mathrm{Gc}_0^n \text{ with } K^n \in \mathbb{N}^*. \tag{2}$$

The scaling factor K^i of the task Ti can be computed according to the minimal configuration granularity Gc_0 and the application-constrained configuration granularity Gc_{appli}:

$$\forall n, \quad K^n = \frac{\mathrm{LCM}(\mathrm{Gc}_0^n, \mathrm{Gc}_{\mathrm{appli}}^n)}{\mathrm{Gc}_0^n}. \tag{3}$$

For instance, in a 20-row image application, the task $T2$, after system homogenization by granularity functions (see Figure 6), has a minimal configuration granularity equal to six rows, $\mathrm{Gc}_0^n = 6$ (the task accepts a reconfiguration after the computation of six rows). However, a designer can impose that the task cannot be reconfigured before the complete image process for QoS reasons (image homogeneity), $\mathrm{Gc}_{\mathrm{appli}}^n = 20$. The scaling factor K^n is then equal to $10(\mathrm{LCM}(6, 20)/6 = 10)$. Finally, the application-constrained configuration granularity imposes that the task produces 60 lines (3 images) before accepting a reconfiguration.

3.3.7. K-setup rules

Two add-on rules ensure a safe and coherent configuration transition. We consider $K^{\mathrm{pred}(n)}$ and K^n the scaling factors applied to the minimal granularity $\mathrm{Gc}_0^{\mathrm{pred}(n)}$ and Gc_0^n of tasks $T^{\mathrm{pred}(n)}$ and T^n (the producer task and the consumer task, resp.)

```
(1)  Initialization();
(2)
(3)  for all sink Task S do
(4)     Config_Parameter_Update(S);
(5)  end for
(6)
(7)  for all n do
(8)     Gc0_Compute(n);
(9)  end for
(10)
(11)  Config_Parameter_Update(n)
(12)
(13)  NP = nb(pred(n))
(14)  i = 0
(15)  while NP > 0 do
(16)     Config_Parameter_Update(pred(n)[i])
(17)     Modif(Cy(n)) = 0
(18)     Modif(Cy(pred(n)[i])) = 0
(19)     A = Cy(n) × R(n, pred(n)[i])
(20)     B = Cy(pred(n)[i], n)×W(pred(n)[i], n)
(21)     if A/B ∈ ℕ then
(22)        V(e(n, pred(n)[i])) = A
(23)     else
(24)        V(e(n, pred(n)[i])) = LCM(A, B)
(25)        C = V(e(n, pred(n)[i])) / R(n, pred(n)[i])
(26)        if Cy(n)≠C then
(27)           Cy(n) = C
(28)           Modif(Cy(n)) = 1
(29)        end if
(30)     end if
(31)     D = V(e(n, pred(n)[i])) / W(pred(n)[i], n)
(32)     if Cy(pred(n)[i])≠D then
(33)        Cy(pred(n)[i]) = D
(34)     Modif(Cy(pred(n)[i])) = 1
(35)     end if
(36)     if Modif(Cy(n)) = 1 then
(37)        i = 0
(38)        NP = nb(pred(n))
(39)     else if Modif Cy(pred(n)[i])) = 1 then
(40)        i = i
(41)        NP = NP
(42)     else
(43)        i = i + 1
(44)        NP = NP − 1
(45)     end if
(46)  end while
(47)
(48)
(49)  Gc0_Compute(n)
(50)
(51)  for all j do
(52)     Gc0^n[j] = Cy(n) × W(n, succ(n)[j]);
(53)  end for
(54)
```

Algorithm 1: Configuration_granularity().

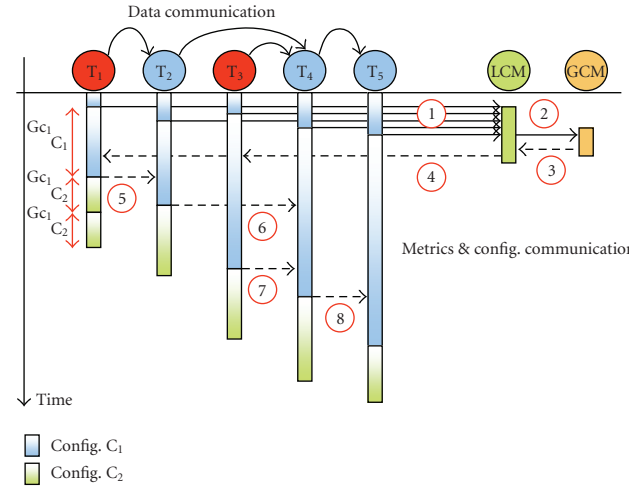


Figure 7: Global configuration management.

Table 1: Communication performance results.

	SW↔ SW	SW↔ HW	HW↔ HW
MB post	517 cy.	2035 cy.	15 cy.
MB pend	425 cy.	3087 cy.	11 cy.

Rule 1. "Data homogeneity": all data computed by the consumer task with a configuration have been produced with the same configuration:

$$\forall n, \quad K^{\mathrm{pred}(n)} \leq K^{n}. \tag{4}$$

Scaling factor K^n from source to sink tasks follows a monotonically decreasing function.

Rule 2. "Computation homogeneity": all data produced with a configuration have to be computed by the consumer task with the same configuration:

$$\forall n, \quad K^{\mathrm{pred}(n)} \geq K^{n}. \tag{5}$$

Scaling factor K^n from source to sink tasks follows a monotonically increasing function.

Rule 3. "Safe reconfiguration rule": applying these two precedent rules means that each task has to respect the following rule:

$$\forall n, \quad K^{\mathrm{pred}(n)} = K^{n} = \mathrm{Max}(K^{n}). \tag{6}$$

In consequence, when all tasks of a cluster use the same scaling factor, data and process homogeneities are ensured and a safe reconfiguration is managed.

3.3.8. Summary

Four cases can be considered. When tasks have no data dependency and there are no application constraints, each task belongs to separate clusters and so has a configuration granularity equal to the initial value of one. Thus, each task can be reconfigured after producing one dataset.

Table 2: Implementation results with a 50 MHz system clock.

(a)

T_1	T_2	T_3	T_4	T_5
Average & Background suppr.	Erosion & Dilation	Reconstruction	Labeling	Display

(b)

T_1-T_2-T_3-T_4-T_5	SW-SW-SW-SW-SW	SW-HW-HW-SW-HW	HW-HW-HW-HW-HW
Exec. Time	245.650.000 cy. ±0,01%	80.800.000 cy. ±0,04%	1.820.000 cy. ±0,2%
Frames/sec.	0,20	0,62	26
Area	19%	59%	92%
StratixII S60 Power	137 mW	228 mW	285 mW

Table 3: Fluctuating execution time due to data.

Task	Variation	Exec. time
Erosion	1 pixel	14.240.816 cy.
	320∗240 pixels	146.197.242 cy.
Reconstruction	1 iteration	15.641.863 cy.
	2 iterations	162.476.520 cy.
	3 iterations	172.675.410 cy.

When tasks have data dependency (they are gathered into a cluster) and no configuration dependency, and there are no application constraints, each task has a configuration granularity equal to the initial value of one. Thus, each task can be configured after one cycle.

When tasks have data dependency and configuration dependency (e.g., the type of data), and there are no application constraints, we compute Gc_0 for each task. Thus, each task can be configured after producing Gc_0 data.

When tasks have data dependency and configuration dependency and there are application constraints (e.g., complete image processing), we compute Gc_0 for each task and rescale it according to constraints. Thus, each task can be configured after producing $K \times Gc_0$ data. K of all tasks is set up according to the third K-setup rule to guarantee data and computation homogeneities.

3.4. Global configuration management

3.4.1. Online configuration process

Figure 7 summarizes the online configuration procedure; upload of metrics (1) from the tasks to the LCM and (2) between local and global configuration managers allows the GCM to select the next configuration. The CID is then downloaded to the LCM (3) which diffuses it by multicasting (4) to the source tasks (T_1 and T_3). Both propagate the message to their downstream tasks as soon as they leave their configuration granularity, and then they take into account the new configuration. The propagation follows the same process gradually (leave the configuration granularity, propagate to its neighbor, and reconfigure it) from the source to the sink tasks (5) and (8). Task T_4 waits the configuration signal from T_2 and T_3 before accepting the new configurations (6) and (7). Otherwise, without a new configuration, tasks continue their process with the current configuration.

3.4.2. Case of nondeterministic data production

In the previous development, the amount of data produced by each task in a given configuration is considered as being fixed; it means that the graph is synchronous in the sense of the SDF formalism. This assumption is no more valid if data production is data-dependent.

However, configuration managers offer the flexibility to solve this issue. In such a case, the amount of produced data is a metric to be sent to the LCM, that can react by sending specific configuration messages to tasks where new configuration granularity is specified. This decision is safe since the LCM can send granularity specification before any reconfiguration order. In practice, the configuration can be recomputed online according to Algorithm 1 or decided according to pre-computed modes.

4. CASE STUDY

Our application is an embedded smart camera for object tracking, as presented in [11], implemented on an Altera Stratix II. It is composed of several tasks which can be implemented in hardware or software. There are two types of tasks. The first one is the set of application tasks for image processing (e.g., averaging, background suppression, erosion, dilation, labeling, etc.). The second one is composed of configuration management tasks (GCM, LCM), sensor and peripheral tasks, which include camera, VGA, and gas gauge controllers. All message passing uses mailbox or message queues

(e.g., configuration or metric message). Image data are transferred through shared memories.

Table 1 shows the communication performances. Overhead of HW ↔ SW communication is due to context switch and control. Otherwise, we reduce drastically the HW to HW communication time. Some architecture configuration may involve an important time overhead (HW and SW tasks alternation). However, in our case study, message passing represents a low percentage of the whole communication.

With different algorithm and architectural configurations, we obtain various tracking system performances as shown in Table 2. We obtain for different architectural and algorithmic configurations a tracking system from 0,2 to 26 frames per second. Execution time results correspond to a tracking process with a standard input frame; so in that case, execution time variations are due to system architecture (e.g., cache miss, bus collision, etc.). However, Table 3 shows some causes of variation at task level due to data characteristics. The reconstruction task is a recursive task depending on object complexity. During each iteration, execution time depends on the number of white pixels. In the same way, erosion and labeling execution times depend on number of white pixels and number of objects, and number of white pixels and object complexity, respectively.

5. CONCLUSION AND PERSPECTIVES

In this paper, we formalize our concept of task configuration management in the context of self-adaptive systems. We propose three contributions implemented as new RTOS services compliant with asynchronous reconfiguration requests. We first define a unified framework for HW/SW task communication and configuration management. We then formalize and provide a systematic solution to compute configuration granularity that guarantees configuration coherency. Finally, we propose a solution to safely control reconfigurations within reconfigurable SOC through propagation principle. Our experience with the image processing application demonstrates the interest of adaptive systems in fluctuating resources requirements according to the environment. We now intend to combine this project with some auto and dynamic reconfiguration experiences already performed on Xilinx FPGA in order to include bitstream dynamic management.

REFERENCES

[1] J. L. Wong, G. Qu, and M. Potkonjak, "An on-line approach for power minimization in qos sensitive systems," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '03)*, pp. 59–64, Kitakyushu, Japan, January 2003.

[2] D. H. Albonesi, "Selective cache ways: on-demand cache resource allocation," in *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pp. 248–259, Haifa, Israel, November 1999.

[3] R. Maro, Y. Bai, and R. I. Bahar, "Dynamically reconfiguring processor resources to reduce power consumption in high-performance processors," in *Proceedings of the 1st International Workshop on Power-Aware Computer Systems (PACS '00)*, p. 97, Cambridge, Mass, USA, November 2001.

[4] J.-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '03)*, vol. 1, pp. 986–991, Munich, Germany, March 2003.

[5] H. Walder and M. Platzner, "Reconfigurable hardware operating systems: from design concepts to realizations," in *Proceedings of International Conference on Engineering of Reconfigurable Systems & Algorithms (ERSA '03)*, pp. 284–287, Las Vegas, Nev, USA, June 2003.

[6] C. Steiger, H. Walder, and M. Platzner, "Operating systems for reconfigurable embedded platforms: online scheduling of real-time tasks," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1393–1407, 2004.

[7] D. Andrews, R. Sass, E. Anderson, et al., "The case for high level programming models for reconfigurable computers," in *Proceedings of International Conference on Engineering of Reconfigurable Systems & Algorithms (ERSA '06)*, pp. 21–32, Las Vegas, Nev, USA, June 2006.

[8] T. Marescaux, V. Nollet, J.-Y. Mignolet, et al., "Run-time support for heterogeneous multitasking on reconfigurable SoCs," *Integration, the VLSI Journal*, vol. 38, no. 1, pp. 107–130, 2004.

[9] V. J. Mooney and D. Blough, "A hardware-software realtime operating system framework for socs," *IEEE Design & Test of Computers*, vol. 19, no. 6, pp. 44–51, 2002.

[10] Y. Eustache, J.-P. Diguet, and M. Khodary, "RTOS-based hardware software communications and configuration management in the context of a smart camera," in *Proceedings of International Conference on Engineering of Reconfigurable Systems & Algorithms (ERSA '06)*, pp. 84–92, Las Vegas, Nev, USA, June 2006.

[11] J.-P. Diguet, Y. Eustache, and M. Kodhary, "Feedback control learning model for qos, power and performance management of reconfigurable embedded systems," in *Proceedings of the 8th International Symposium on DSP and Communication Systems (DSPCS '05)*, Noosa Heads, Australia, December 2005.

[12] H. Walder and M. Platzner, "Reconfigurable hardware OS prototype," April 2003, ftp://ftp.tik.ee.ethz.ch/pub/people/walder/TIKR168.pdf.

[13] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, vol. 36, no. 1, pp. 24–35, 1987.

Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 518904, 15 pages
doi:10.1155/2008/518904

Research Article

Assessing Task Migration Impact on Embedded Soft Real-Time Streaming Multimedia Applications

Andrea Acquaviva,[1] Andrea Alimonda,[2] Salvatore Carta,[2] and Michele Pittau[2]

[1] *Institute of Information Science and Technology, Electronics and Systems Design Group, University of Verona, 37100 Verona, Italy*
[2] *Department of Mathematics and Computer Science, University of Cagliari, 09100 Cagliari, Italy*

Correspondence should be addressed to Andrea Alimonda, alimonda@sc.unica.it

Received 4 April 2007; Revised 10 August 2007; Accepted 19 October 2007

Recommended by Alfons Crespo

Multiprocessor systems on chips (MPSoCs) are envisioned as the future of embedded platforms such as game-engines, smartphones and palmtop computers. One of the main challenge preventing the widespread diffusion of these systems is the efficient mapping of multitask multimedia applications on processing elements. Dynamic solutions based on task migration has been recently explored to perform run-time reallocation of task to maximize performance and optimize energy consumption. Even if task migration can provide high flexibility, its overhead must be carefully evaluated when applied to soft real-time applications. In fact, these applications impose deadlines that may be missed during the migration process. In this paper we first present a middleware infrastructure supporting dynamic task allocation for NUMA architectures. Then we perform an extensive characterization of its impact on multimedia soft real-time applications using a software FM Radio benchmark.

1. INTRODUCTION

Multiprocessor systems on chip are of increasing and widespread use in many embedded applications ranging from multimedia to biological processing [1]. To satisfy application requirements by ensuring flexibility, many proposals were rising for architectural integration of system components and their interconnection. Currently, two main directions can be identified that lead to commercial products: master-slave architectures [2, 3], homogeneous architectures [4]. Due to the advances in integration and the increasing chip size, computational tiles are going to be resemble to elements of a cluster, with a nonuniform type of memory access (NUMA) and connected by networks on chip (NoC). This architecture addresses the scalability issues proper of symmetric multiprocessors that limit the number of integrable cores. It follows the structure envisioned for non-cache-coherent MPSoCs [5, 6]. These systems run tasks in private memories, like in distributed systems but similarly to symmetric multiprocessor SoCs, they exploit shared memories for communication and synchronization between processing nodes.

Even if MPSoCs are becoming the preferred target for multimedia embedded systems because they allow to fit a huge number of different applications, there are still fundamental challenges concerning the mapping of task into such a complex systems. The target is to balance the workload among processing units to minimize some metric such as overall execution time, power, or even temperature. Nevertheless, the software support implementing these strategies, that can be defined as the *resource manager*, must provide to the programmer a clean and standard interface for developing and running multimedia applications while hiding low-level details such as to which processor each task of the application is mapped. Given the large variety of possible use cases that these platforms must support and the resulting workload variability, offline approaches are no longer sufficient as application mapping paradigms, because they are limited in handling workload modifications due to variable quality of service (QoS) requirements. The extensive offline characterization on which they are based becomes unpractical in this context. Moreover, even when workload is constant, next generation processor will be characterized by variable performance, requiring online adaptation [7].

For this reason, dynamic mapping strategy have been recently proposed [8]. Run-time task allocation has been shown to be a promising technique to achieve load balancing, power minimization, temperature balancing, and reliability improvement [9]. To enable dynamic allocation, a task migration support must be provided. In distributed systems such as high-end multiprocessors and clusters of workstations (CoW) [10], task migration has been implemented and supported at the middleware level. Mosix is a well-known load-balancing support for CoW implemented at the kernel level that moves tasks in a fully transparent way. In the embedded multimedia system context however, task migration support has to trade off transparency with efficiency and lightweight implementation. User-level migration support may help in reducing its impact on performance [11]. However, it requires a nonnegligible effort by the programmer that has to explicitly define the context to be saved and restored in the new process life.

Multimedia applications are characterized by soft real-time requirements. As a consequence, migration overheads must be carefully evaluated to prevent deadline misses when moving processes between cores. These overheads depend on migration implementation which in turn depends on system architecture and communication paradigm. In distributed memory MPSoCs task migration involves task memory content transfer. As such, migration overhead depends both on the migration mechanism and on the task memory footprint. For this reason, there is the need of developing an efficient task migration strategy suitable for embedded multimedia MPSoC systems.

In this work, we address this need by presenting a middleware layer that implements task migration in MPSoCs and we assess its effectiveness when applied in the context of soft real-time multimedia applications. To achieve this target, we designed and implemented the proposed middleware on top of uClinux operating system running on a prototype multicore emulation platform [12]. We characterized its performance and energy overhead through extensive benchmarking and we finally assessed its effectiveness when applied to a multitask software FM Radio multitasking application. To test migration effectiveness, we impose in our experiments a variable frame rate to the FM Radio. Tasks are moved among processors as the frame rate changes to achieve the best configuration in terms of energy efficiency that provides the required performance level. Each configuration consists in a mapping of task to processor and the corresponding frequencies. Configurations are precomputed and stored for each frame rate.

Our experiments demonstrate that (i) migration at the middleware/OS level is feasible and improves energy efficiency of multimedia applications mapped on multiprocessor systems; (ii) migration overhead in terms of QoS can be hidden by the data reservoir present in interprocessor communication queues.

The rest of the paper is organized in the following way. Background work is covered in Section 2. Section 3 describes the architectural template we considered. Section 4 covers the organization of the software abstraction layer. Multimedia application mapping is discussed in Section 5 while results are discussed in Section 6.

2. BACKGROUND WORK

Due to the increasing complexity of these processing platforms, there is a large quantity and variety of resources that the software running on top of them has to manage. This may become a critical issue for embedded application developers, because resource allocation may strongly affect performance, energy efficiency, and reliability [13]. As a consequence, from one side there is need of efficiently exploit system resources, on the other side, being in an embedded market, fast and easy development of applications is a critical issue. For example, since multimedia applications are often made of several tasks, their mapping into processing elements has to be performed in a efficient way to exploit the available computational power and reducing energy consumption of the platform.

The problem of resource management in MPSoCs can be tackled from either a static or dynamic perspective. Static resource managers are based on the a priori knowledge of application workload. For instance, in [14] a static scheduling and allocation policy is presented for real-time applications, aimed at minimizing overall chip power consumption taking also into account interprocessor communication costs. Both worst case execution time and communication needs of each tasks are used as input of the minimization problem solved using integer linear programming (ILP) techniques. In this approach, authors first perform allocation of tasks to processors and memory requirement to storage devices, trying to minimize the communication cost. Then scheduling problem is solved, using the minimization of execution time as design objective.

Static resource allocation can have a large cost, especially when considering that each possible set of applications may lead to a different use case. The cost is due to run-time analysis of all use cases in isolation. In [15] a composition method is proposed to reduce the complexity of this analysis. An interesting semistatic approach that deals with scheduling in multiprocessor SoC environments for real-time systems is presented in [16]. The authors present a task decomposition/clustering method to design a scalable scheduling strategy. Both static and semistatic approaches have limitations in handling varying workload conditions due to data dependency or to changing application scenarios. As a consequence, dynamic resource management came into play.

Even if scheduling can be considered a dynamic resource allocation mechanism, in this paper we assume that a main feature of a dynamic resource manager in a multiprocessor system is the capability of moving tasks from processing elements at run time. This is referred to as task migration.

In the field of multiprocessor systems-on-chip, process migration can be effectively exploited to facilitate thermal chip management by moving tasks away from hot processing elements, to balance the workload of parallel processing elements and reduce power consumption by coupling dynamic voltage and frequency scaling [17–19]. However, the implementation of task migration, traditionally developed

for computer clusters or symmetric multiprocessor, cache-coherent machines, poses new challenges [11]. This is specially true for non-cache-coherent MPSoCs, where each core runs its own local copy of the operating system in private memory. A migration paradigm similar to the one implemented in computer clusters should be considered, with the addition of a shared memory support for interprocessor communication.

For instance, many embedded system architectures do not even provide support for virtual memory; therefore, many task migration optimization techniques applied to systems with remote paging support cannot be directly deployed, such as the eager dirty [10] or the copy-on-reference [20] strategies.

In general, migrating a task in a fully distributed system involves the transfer of processor state (registers), user level and kernel level context, and address space. A process address space usually accounts for a large fraction of the process state; therefore, process migration performance largely depends on the transfer efficiency of the address space. Although a number of techniques have been devised to alleviate this migration cost (e.g., lazy state transfer, precopying, residual dependencies [21]), a frequent number of migrations might seriously degrade application performance in an MPSoC scenario. As a consequence, assessing the impact of migration overhead is critical.

In the context of MPSoCs, in [8] a selective code/data migration strategy is proposed. Here authors use a compilation-level code profiling technique to evaluate the communication energy cost of transferring each function and procedure over the on-chip network. This information is used to decide whether it is worth migrating tasks on the same processor to reduce communication overhead or transferring data between them.

In [11], a feasibility study for the implementation of a lightweight migration mechanism is proposed. The user-managed migration scheme is based on code checkpointing and user-level middleware support. The user is responsible for determining the context to be migrated. To evaluate the practical viability of this scheme, authors propose a characterization methodology for task migration overhead, which is the minimum execution time following a task migration event during which the system configuration should be frozen to make up for the migration cost.

In this work, task migration for embedded systems is evaluated when applied to a real-world soft real-time application. Compared to [11], our migration strategy is implemented at the operating system and middleware level. Checkpoints are only used to determine migration points, while the context is automatically determined by the operating system.

3. TARGET ARCHITECTURE ORGANIZATION

The software infrastructure we present in this work is targeted to a wide range of multicore platforms having a number of homogeneous cores that can execute the same set of tasks, otherwise task migration is unfeasible. A typical target homogeneous architecture is the one shown in Figure 1(a). The architectural template we consider is based on a configurable number of 32-bit RISC processors without memory management unit (MMU) accessing cacheable private memories and a single noncacheable shared memory.

The template we are targeting is compliant with state-of-the-art MPSoC architectures. For instance, because of its synergistic processing elements with local storage and shared memory used as support for message-based stream processing is closely related to CELL [22].

In our target platform, each core runs in the logical private memory a single operating system instance. This is compliant with the state-of-the-art homogeneous multicore architectures such as the MP211 multicore by NEC [23].

As regards as task migration, the main architectural impact is related to the interconnect model. As we will describe in Section 6, our target platform uses a shared bus as interconnect. This is the worst case for task migration since it implies data transfers from the private memory of one core to the one of another core.

As far as this MPSoC model is concerned, processor cores execute tasks from their private memory and explicitly communicate with each others by means of the shared memory [24]. Synchronization and communication are supported by hardware semaphores and interrupt facilities: (i) each core can send interrupts to others using a memory-mapped interprocessor interrupt module; (ii) cores can synchronize between each other using a hardware *test-and-set* semaphore module that implements test-and-set operations. Additional dedicated hardware modules can be used to enhance interprocessor communication [25, 26]. In this paper, we consider a basic support to save the portability of the approach.

4. SOFTWARE INFRASTRUCTURE

Following the distributed NUMA architecture, each core runs its own instance of the uClinux operating system [27] in the private memory. The uClinux OS is a derivative of Linux 2.4 kernel intended for microcontrollers without MMU. Each task is represented using the process abstraction, having its own private address space. As a consequence, communication has to be explicitly carried on using a dedicated shared memory area on the same on-chip bus. The OS running on each core sees the shared area as an external memory space.

The software abstraction layer is described in Figure 1(b). Since uClinux is natively designed to run in a single-processor environment, we added the support for interprocessor communication at the middleware level. This organization is a natural choice for a loosely coupled distributed systems with no cache coherency, to enhance efficiency of parallel application without the need of a global synchronization, that would be required by a centralized OS. On top of local OSes we developed a layered software infrastructure to provide an efficient parallel programming model for MPSoC software developers enabled by an efficient task migration support layer.

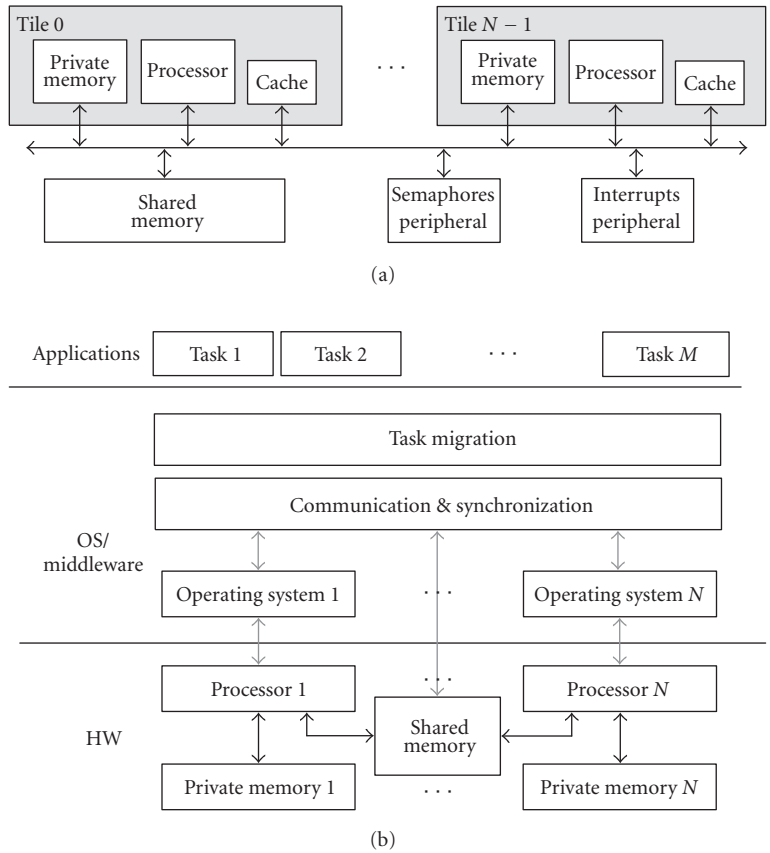


Figure 1: Hardware and software organizations: (a) target hardware architecture; (b) scheme of the software abstraction layer.

4.1. Communication and synchronization support

The communication library supports message passing through mailboxes. They are located either in the shared memory space or in smaller private scratch-pad memories, depending on their size and depending if the task owner of the queue is defined as migratable or not. The concept of migratable task will be explained later in this section. For each process a message queue is allocated in shared memory.

To use shared memory paradigm, two or more tasks are enabled to access a memory segment through a *shared malloc* function that returns a pointer to the shared memory area. The implementation of this additional system call is needed because by default the OS is not aware of the external shared memory. When one task writes into a shared memory location, all the other tasks update their internal data structure to account for this modification. Allocation in shared memory is implemented using a parallel version of the Kingsley allocator, commonly used in Linux kernels.

Task and OS synchronization is supported providing basic primitives like binary and counting semaphores. Both spinlock and blocking versions of semaphores are provided. Spinlock semaphores are based on hardware test-and-set memory-mapped peripherals, while nonblocking semaphores also exploit hardware interprocessor interrupts to signal waiting tasks.

4.2. Task migration support

To handle dynamic workload conditions and variable task and workload scenarios that are likely to arise in MPSoCs targeted to multimedia applications, we implemented a task migration strategy enabled by the middleware support. Migration policies can exploit this mechanism to achieve load balancing for performance and power reasons. In this section, we describe the middleware-level task migration support. In Section 6, we will show how task migration can be used to improve the efficiency of the system.

In our implementation, migration is allowed only at predefined checkpoints, that are provided to the user through a library of functions together with message passing primitives. A so-called *master daemon* runs in one of the cores and takes care of dispatching tasks on the processors. We implemented two kinds of migration mechanisms that differs in the way the memory is managed. A first version, based on a so-called "task-recreation" strategy, kills the process on the original processor and recreate it from scratch on the target processor. This support works only in operating systems supporting dynamic loading, such as uClinux. Task recreation is based on the execution of fork-exec system calls that take care of allocating the memory space required for the incoming task. To support task recreation on an architecture without MMU performing hardware address

translation, a position-independent type of code (called PIC) is required to prevent the generation of wrong references of pointers, since the starting address of the process memory space may change upon migration.

Unfortunately, PIC is not supported by the target processor we are using in our platform (microblazes) [28]. For this reason, we implemented an alternative migration strategy where a replica of each task is present in each local OS, called "task-replication." Only one processor at a time can run one replica of the task. While here the task is executed normally, in the other processors it is in a queue of suspended tasks. As such, a memory area is reserved for each replica in the local memory, while kernel-level task-related information is allocated by each OS in the process control block (PCB) (i.e., an array of pointers to the resources of the task). A second valid reason to implement this alternative technique is because deeply embedded operating systems are often not capable of dynamic loading and the application code is linked together with the OS code. Task replication is suitable for an operating system without dynamic loading because the absolute memory position of the process address space does not change upon migration, since it can be statically allocated at compile time. This is the case of deeply embedded operating systems such as RTEMS or eCos. This is compliant also with heterogeneous architectures, slave processors run a minimalist OS, that is, a library statically linked with the tasks to be run, that are known a priori. The master processor typically runs a general purpose OS such as Linux. Even if this technique leads to a waste of memory for migratable tasks, it has also the advantage of being faster, since it cuts down on memory allocation time with respect to a task recreation.

To further limit waste of memory, we defined both *migratable* and *nonmigratable* types of tasks. A migratable task is launched using a special system call, that enables the replication mechanism. Nonmigratable tasks are launched normally. As such, in the current implementation the user is responsible for distinguishing between the two types of tasks. However, in future implementation the middleware itself could be responsible of selecting migratable tasks depending on task characteristics.

The difference in terms of migration costs for the two strategies is shown in Figure 2. Cost is shown in terms of processor cycles needed to perform migrations as a function of the task size. In both cases, there is a contribution to migration overhead due to the amount of data transferred through the shared memory. Moreover, for task recreation technique, there is another overhead due to the additional time required to reload the program code from the file system. This explains the offset between the two curves. Moreover, the task recreation curve as a larger slope. This is due to the fact that a large amount of memory transfers also lead to an increasing contention on the bus, so that the contribution on the execution time increases more as the file size increases with respect to the task replication case.

In our system, the migration process is managed using two kinds of kernel daemons (part of the middleware layer), a master daemon running in a single processor, and slave daemons running in all the processors. The communication between master and slave daemons is implemented using dedicated, interrupt-based messages in shared memory. The master daemon takes care of implementing the run-time task allocation policy. Tasks can be migrated only corresponding to user-defined checkpoints. The code of the checkpoints is provided as a library to the programmer. When a new task or an application (i.e., a set of tasks) is launched by the user, the master daemon sends a message to each slave, that forks an instance of each task in the local processor. Depending on master's decision, tasks that have not to be executed on the local processor are placed in the suspended tasks queue, while the others are placed in the ready queue.

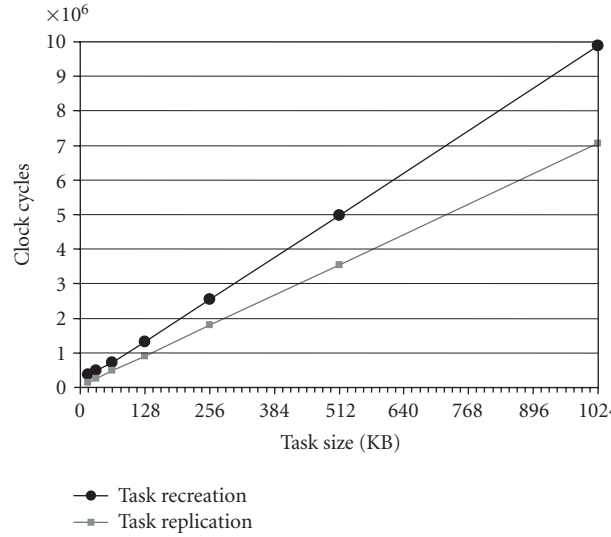


Figure 2: Migration cost as a function of task size for task replication and task recreation.

During execution, when a task reaches a user-defined checkpoint, it checks for migration requests performed by the master daemon. If the migration is taken, they suspend their execution waiting to be deallocated and restore to another processor from the migration middleware. When the master daemon wants to migrate a task, it signals to the slave daemons of the source processor that a task has to be migrated. A dedicated shared memory space is used as a buffer for task context transfer. To assist migration decision, each slave daemon writes in a shared data structure the statistics related to local task execution (e.g., processor utilization and memory occupation of each task) that are periodically read by the master daemon.

Migration mechanisms are outlined in Figure 3. Both execution and memory views are shown. With task replication (Figures 3(a) and 3(b)), the address space of all the tasks is present in all the private memories of processor 0,1, and 2. However, only a single instance of a task is running on processor 0, while others are sleeping on processors 1 and 2. It must be noted that master daemon (*M_daemon* in Figure 3) runs on processor 0 while slave daemons (*S_daemon* in Figure 3) run on all of the processors. However, any processor can run the master daemon.

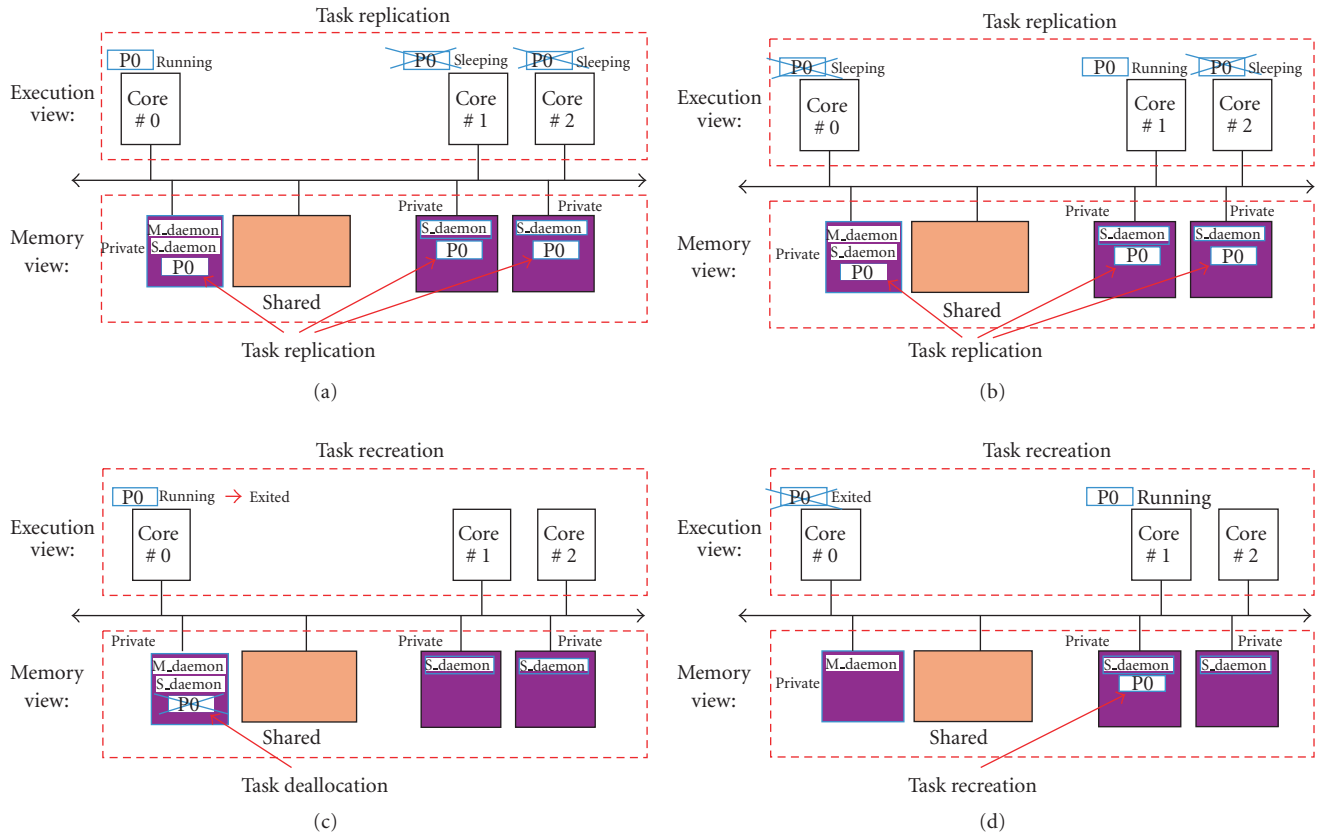


FIGURE 3: Migration mechanism: (a) task replication phase 1; (b) task replication phase 2; (c) task recreation phase 1; (d) task recreation phase 2.

Figures 3(c) and 3(d) shows task recreation mechanism. Before migration, process P0 runs on processor 0 and occupies memory space on the private memory of the same processor. Upon migration, P0 performs an exit system call and thus its memory space is deallocated. After migration (Figure 3(d)), memory space of P0 is reallocated on processor 1, where P0 runs.

Being based on a middleware-level implementation running on top of local operating systems, the proposed mechanism is suitable for heterogeneous architectures and its scalability is only limited by the centralized nature of the master-slave daemon implementation.

It must be noted that we have implemented a particular policy, where the master daemon keeps track of statistics and triggers the migration; however, based on the proposed infrastructure, a distributed load balancing policy can be implemented with slave daemons coordinating the migration without the need of a master daemon. Indeed, the distinction between master and slaves is not structural, but only related to the fact that the master is the one triggering the migration decision, because it keeps track of task allocation and loads. However, using an alternative scalable distributed policy (such as the Mosix algorithm used in computer clusters) this distinction is no longer needed and slave daemons can trigger migrations without the need of a centralized coordination.

5. MULTIMEDIA SOFT REAL-TIME APPLICATION MAPPING

Multimedia applications are typically composed by multiple tasks. In this paper, we consider each task as a process with its own private address space. This allows us to easily map these applications on a distributed memory platform like the one we are targeting in this work. The underlying framework supports task migration so that dynamic resource management policies can take care of run-time mapping of task to processors, to improve performance, power dissipation, thermal management, reliability. The programmer is not exposed to mapping issues, it is only responsible for the communication and synchronization as well as code checkpointing for migration. Indeed, task migration can occur only corresponding to checkpoints manually inserted in the code by the programmer.

In our migration framework, all the data structures describing the task in memory are replicated. Upon migration, the only kernel structure that is moved is the stack. As such, if a process has opened a local resource, this information is lost after migration. The programmer is responsible for carefully selecting migration points or eventually reopening resources left open in the previous task life.

An alternative, completely transparent approach, is the one implemented in Mosix for computer clusters [10], where

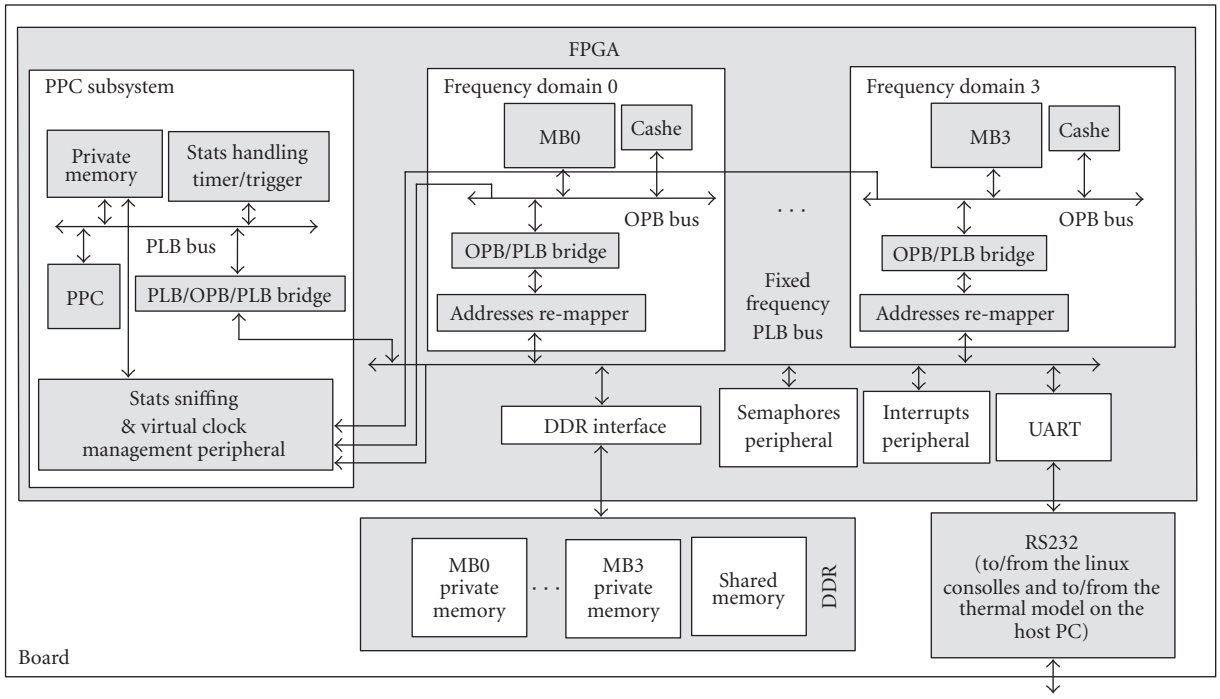


Figure 4: Overview HW architecture of emulated MPSoC platform.

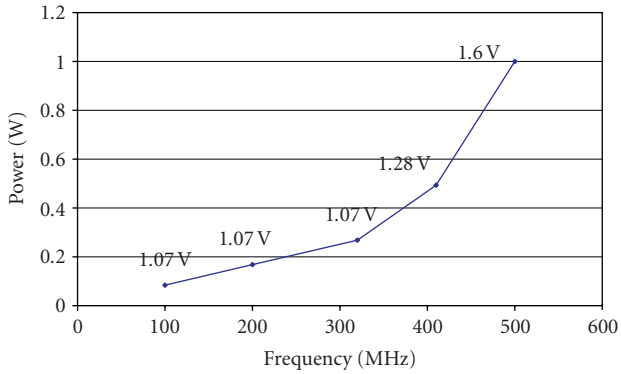


Figure 5: Power model: power as a function of the frequency.

a home processor defined for each process requires the implementation of a forwarding layer that takes care of forwarding system calls to the home node. In our system, we do not have the notion of home node. A more complex programming paradigm is thus traded off with efficiency and predictability of migration process. This approach is much more suitable to an embedded context, where controllability and predictability are key issues.

6. EXPERIMENTAL RESULTS

In this section, results of a deep experimental analysis of the multiprocessing middleware performed using an FPGA-based multiprocessor emulation platform are described. A first run of tests, based on synthetic applications, have been used to characterize the overhead of migration in terms of code checkpointing and run-time support. A second set of tests were performed on a soft real-time streaming application to highlight the effect of migration on a real-life multimedia application. Results show that a temporary QoS degradation due to migration can be hidden by exploiting data reservoir in interprocessor communication buffers and provide design guidelines concerning buffer size and migration times.

6.1. Emulation platform description and setup

For the simulation and performance evaluation of the proposed middleware, we used an FPGA-based, cycle accurate, MPSoC hardware emulator [12] built on top of the Xilinx XUP FPGA board [28], and described in Figure 4. Time and energy data are run-time stored using a nonintrusive, statistics subsystem, based on hardware sniffers which store frequencies, bus, and memory accesses. A PowerPC processor manages data and control communication with the host PC using a dedicated UART-based serial protocol. Run-time frequency scaling is also supported and power models running on the host PC allow to emulate voltage scaling mechanism. Frequency scaling is based on memory-mapped frequency dividers, which can be programmed both by microblazes or by PowerPC. The power associated to each processor is 1 Watt for the maximum frequency, and scales down almost cubically to 84 mW as voltage and frequency decrease. Power data refers to a commercial embedded RISC processor and is provided by an industrial partner. Power/frequency relationship is detailed in Figure 5. The emulation platform runs at 1/10 of the emulated frequency, enabling the experimentation of complex applications which may not be

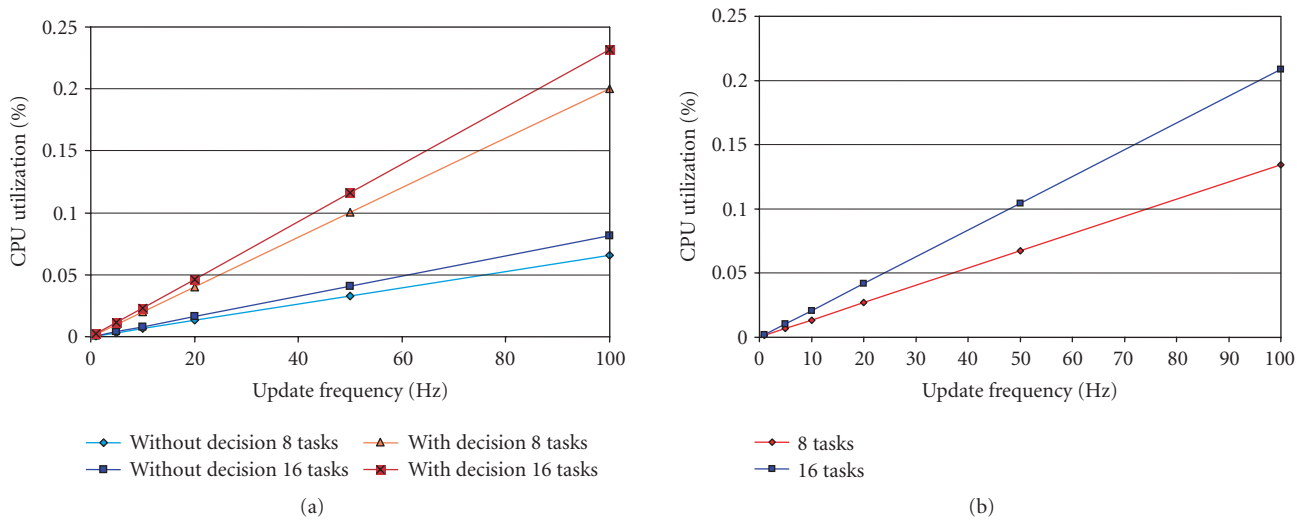


Figure 6: Migration daemon overhead in terms of processor utilization: (a) master daemon CPU load; (b) slave daemon CPU load.

experimented using software simulators with comparable accuracy.

In our platform, private memories are physically mapped into an external DDR memory. This is because the image of uClinux does not fit into the on-chip SRAM. This architectural solution is similar to NEC MP211 multicore platform [23].

Even in this particular implementation, we use an external DDR to map shared and private memories, the proposed approach (i.e., task replication version) can be also implemented in deeply embedded operating systems whose memory footprint would fit into a typical on-chip SRAM.

6.2. Migration support characterization

We present the results of experiments carried out to assess the overhead of the task migration infrastructure in terms of execution time. We exploited a controllable synthetic benchmark to this purpose, thanks to which we evaluated three components of the migration overhead: (i) the cost of the slave daemons running in background to check tasks runtime statistics; (ii) the cost of the master daemon running in background to trigger task shutoff and resume in the various processor cores; (iii) the number of cycles required to complete a migration, that is to shut off a task in the source core, transfer its context, and resume it on the destination core.

We evaluated the overhead of the master daemon by measuring its CPU utilization. Its job is to read the statistics about all the tasks in the system from the shared memory and look in a prestored lookup table whether or not a task should be migrated and where. As such, this overhead does not include the overhead due to (i) the pure cost of migration support and (ii) to the cycles spent to look in the table. We measured these two contributions separately to evaluate migration overhead independently from the particular migration policy. Figure 6(a) shows the CPU load for the two considered cases as a function of the frequency of daemon invocation and for two different quantities of task present in the system. We observed a negligible overhead, highlighting that there is a room for implementing more complex task allocation and migration policies in the master daemon without impacting system performance.

We evaluated the CPU utilization imposed by slave daemons which periodically write task information (currently the task load) in a dedicated data structure allocated in shared memory. This information is then exploited for migration decisions. In Figure 6(b), the CPU utilization of the daemon is shown as a function daemon period in a range of 1 to 100 Hz, two curves are plotted, for 8 and 16 tasks, referred to a core speed of 100 MHZ. It is worth to note that, although the processor speed is moderate, the overhead is negligible being lower than 0.25% in the worst case (update frequency of 100 Hz).

Finally, to quantitatively evaluate the cost of a single task migration, we exploited the capability of our platform to monitor the amount of processor cycles needed to perform a migration, from the checkpoint execution to the complete resume of the task on the new processor. Figure 7 shows migration cost as a function of the task size. Migration overhead is dominated by data transfer. Two main contributions affect the size of the task context to be transferred. The first is the kernel memory space reserved by uClinux to each process to allocate process data structures and the user memory space associated to the task to be migrated. The first contribution consists in our current implementation only of the kernel stack of the processor. This is because the other data structures are already replicated in all the kernels in the other cores (because of the replication strategy). Data, code sections, and heap belong to the second contribution. It must be taken into account that the minimum allocation of user space memory for each processor is 64 KB, even if the process is smaller. In our current implementation, we copy the whole memory areas because we exploit the presence of the memory pointers in the process control block (PCB). With the help of some additional information, optimized techniques can be implemented to reduce the amount of data to be copied, that will

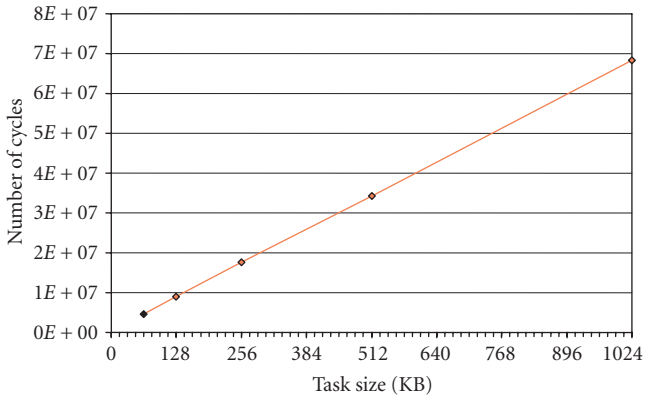


Figure 7: Task migration cost.

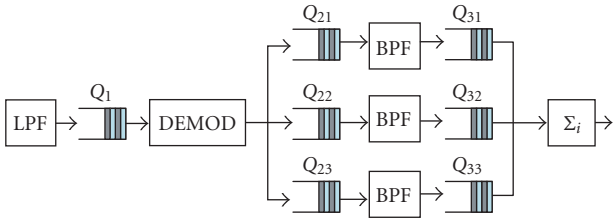


Figure 8: FM radio.

be an object of future work. The linear behavior of the cycle count shows that the only overhead added by the migration support is a constant amount of cycles that are needed to copy the task kernel context and to perform the other migration control operations as described in Section 4. This is visible as the intercept with the Y-axis is in the plot and its value is 104986 cycles.

Finally, it must be noted that a hardware support for data transfer, such as a DMA controller, that is currently not present in our emulation platform, could greatly improve migration efficiency.

6.3. Impact of migration on soft real-time streaming applications

To evaluate the effectiveness of the proposed support on a real test bed, we ported to our system a software FM radio benchmark, that is a representative of a large class of streaming multimedia applications following the split-join model [29] with soft real-time requirements. It allows to evaluate the tradeoff between the long-term performance improvement given by migration-enabled run-time task remapping and the short-term overhead and performance degradation associated to migration.

As shown in Figure 8, the application is composed by various tasks, graphically represented as blocks. Input data represent samples of the digitalized PCM radio signal which has to be processed in order to produce an equalized baseband audio signal. In the first step, the radio signal passes through a *lowpass filter* (LPF) to cut frequencies over the radio bandwidth. Then, it is demodulated by the *demodulator* (DEMOD) to shift the signal at the baseband and produce the audio signal. The audio signal is then equalized with a number of *bandpass filters* (BPF) implemented with a parallel split-join structure. Finally, the *consumer* (Σ) collects the data provided by each BPF and makes the sum with different weights (gains) in order to produce the final output.

To implement the communication between tasks, we adopted the message passing paradigm discussed in Section 4. Synchronization among the tasks is realized through message queues, so that each task reads data from its input queue and sends the processed results to the output queue to be read by the next task in the software pipeline. Since each BPF of the equalizer stage acts on the same data flow, the demodulator has to replicate its output data flow writing the same packet on every output queue.

Thanks to our platform, we could measure the workload profile of the various tasks. We verified that the most computational intensive stage is the demodulator, imposing a CPU utilization of 45% of the total, while for the other tasks we observed, respectively, 5% for the LPF, 13% for each BPF and 5% for the consumer. This information will be used by the implemented migration support to decide which task has to be migrated.

To assess migration advantages and costs, we refer to migration represented in Figure 9. We start from a single processor configuration providing a low frame rate. For each frame rate we stored the best configuration in terms of energy efficiency that provides the required performance level. Thus, as the frame rate imposed by the application increases, we move first to a two processors configuration and then to a three processors configuration. For each processor we also predetermined the appropriate frequency level.

In the rest of this subsection, we first show how the best configurations and the frame rate represent the crossing points between configurations. Second, to show what is the cost of transition from a configuration to another, we detail migration costs in terms of performance (frame misses) and energy.

Optimal task configurations

We use migration to perform run-time task remapping, driving the system from an unbalanced situation to a more balanced one. The purpose of the migration policy we used in this experiment is to split computational load between processing cores. Each processor automatically scales its speed depending on the value stored in a lookup table as a function of the frame rate.

It must be noted that minimum allowed core frequency is 100 MHz. As a consequence, there is no convenience in moving tasks if the cumulative processor utilization leads to a frequency lower than the minimum. Task load depends on the frame rate specified by the user, which in turn depends on the desired sound quality. After initial placement, migration is needed to balance workload variations generated by run-time user requests. The migration is automatically triggered by the master daemon integrated in our layer, following an offline computed lookup table which gives the best task/frequency mapping as a function of these run-time events. It is worth noting that in this paper, we are not

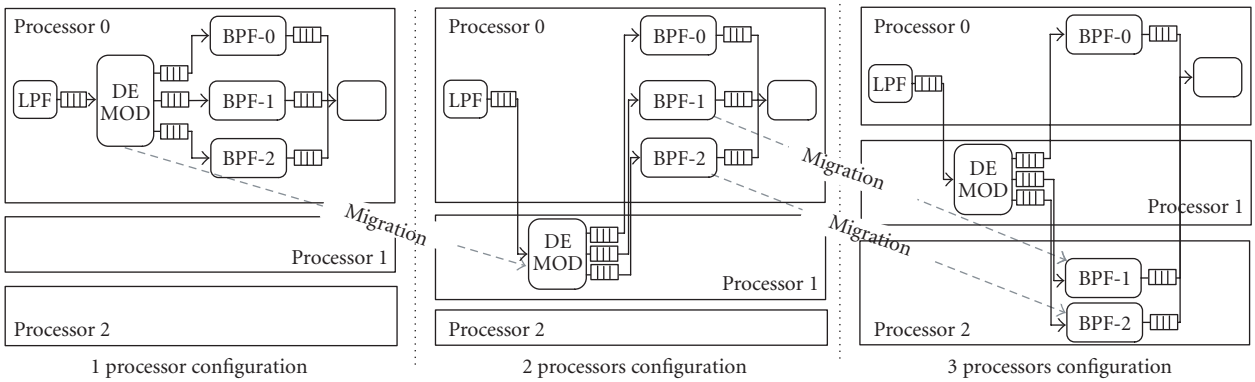


Figure 9: Task migrations performed to support the required frame rate.

interested in evaluating task/frequency allocation policies but the efficiency of the infrastructure. Rather, we are interested in demonstrating how migration cost is compensated by the execution time saved after migration when the system runs a more balanced configuration.

Task reallocation allows to improve the energy efficiency of the system at the various frame rates requested by the user. We show the impact of task reallocation on energy efficiency and we outline energetic cost of task migration. Energy-efficient configurations are computed starting from energy versus frame-rate curves. Each curve represents all the configurations that are obtained by keeping the same number of processors while increasing the frequency to match the desired frame rate. Then we take the Pareto configurations that minimize energy consumption for a given frame rate.

We considered two different energy behaviors of our system. In the first configuration, called *always on*, we do not use any built-in power-saving scheme, that is, the cores are always running and consume power depending on their frequency and voltage. In the second configuration, we emulate a *shutdown-when-idle* power-saving scheme in which the cores are provided with an ideal low-power state in which their power consumption is negligible.

In the *always on* case, the power consumed by the cores as a function of the frame rate is shown in Figure 10. On the left side of the figure, the plot shows three curves obtained by computing the power consumed when mapping the FM Radio tasks in one, two, or three cores. Following one of the curves, it can be noted that, due to the frequency discretization, power consumption is almost constant in frame rate intervals where the cores run at the same frequency/voltage. In each one of these intervals, the amount of idleness is maximum at the beginning and decreases until a frequency step is needed to support the next frame rate value. On the right side of the figure, the corresponding configurations are highlighted, as well as the task mapping and frequencies for each core. Being the demodulator task, the more computational intensive, it is the task that runs at the higher frequency than others.

In order to determine the best power configuration for each frame rate, we computed the Pareto points, as shown in Figure 11. Intuitively, when the frame rate increases, a higher number of cores are needed to get the desired QoS in an energy-efficient way. However, due to the frequency discretization, this is not true. There are cases in which energy efficiency is achieved using a lower number of cores for a higher frame rate. In practice, the "Pareto path" shown on the right side of Figure 11 is not monotonic on the number of cores. Migrations needed to go from one configuration to another are also highlighted as right-oriented arrows on the left side of the figure.

Conversely, in the *shutdown-when-idle* case, power consumption is no longer constant for a given frequency/voltage, because the cores are more active as the frame rate increases for a given frequency. This is shown in Figure 12, where the same curves presented before for the *always on* case are shown, representing static task mappings in one, two, and three cores.

Frequency discretization has less impact in this case. Indeed, by observing the Pareto curve shown in Figure 13, it is evident that the number of cores of the Pareto configurations is monotonically increasing as a function of the frame rate. In Figure 13 migration costs are also detailed. Following the Pareto curve from left to right, the various task mappings and corresponding core frequencies are shown. In particular, core configurations across migration points are shown as well as migration costs in terms of energy. The cost of migration has been evaluated in terms of cycles as described before. These cycles lead to an energy cost, depending on the frequency and voltage at which the core is executing the migration. In Figure 13, this cost is computed considering that migration is performed at the maximum frame rate sustainable at a given configuration, that is, 3000 fps for the 1-processor configuration and 4200 fps for the 2-processor configuration. In general, however, migration cost for transitioning from one configuration to another depends on the actual frame rate which determines the processor frequency. To detail the cost of migration at the various frequencies, we reported energy spent on transitioning from each configuration to another in Figure 14.

Migration energy cost depends on the frequency and voltage. The first set of bars shows the cost of migration from one-core configuration to two-cores configuration, which implies moving only the demodulator task. The second set

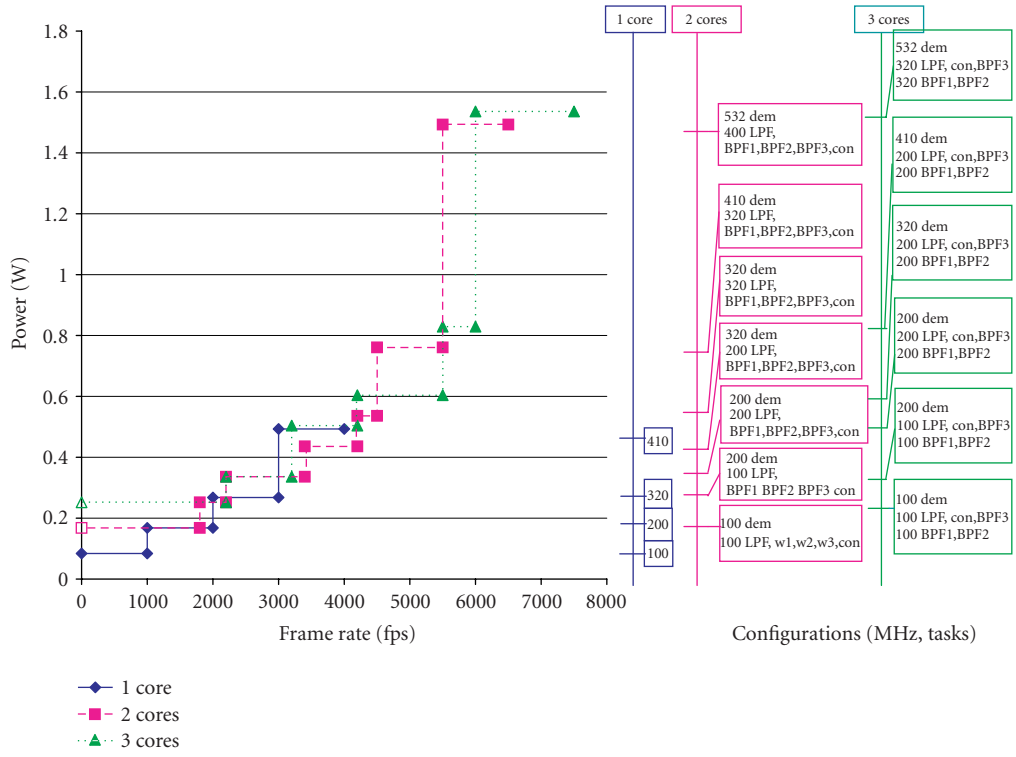


FIGURE 10: Energy cost of static task mappings in the *always on* case.

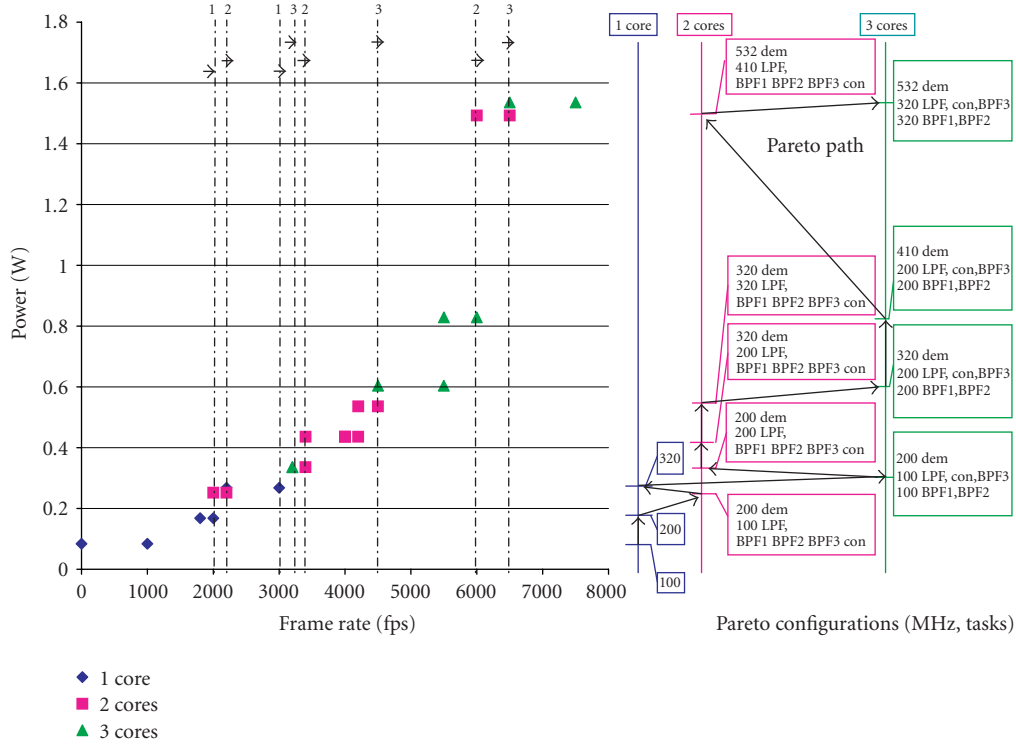


FIGURE 11: Pareto optimal configurations in the *always on* case.

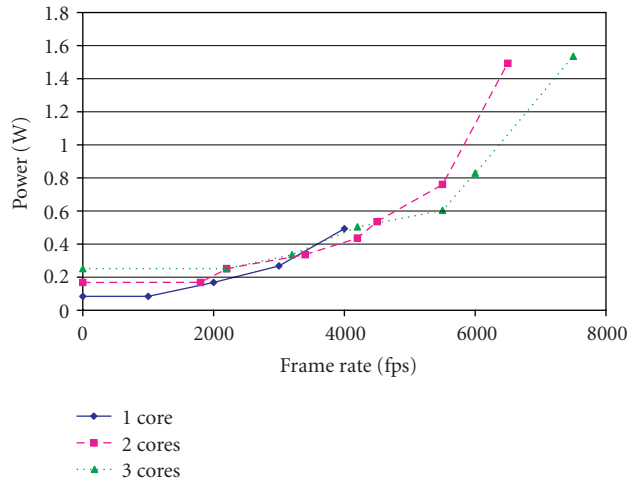


Figure 12: Energy cost of static task mappings in the *shutdown-when-idle* case.

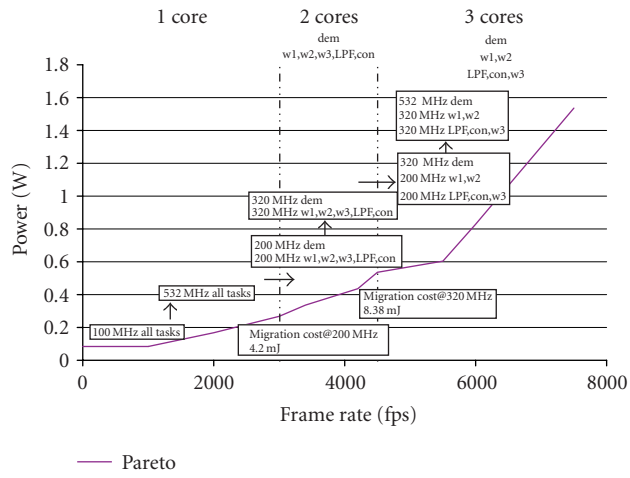


Figure 13: Pareto optimal configurations in the *shutdown-when-idle* case.

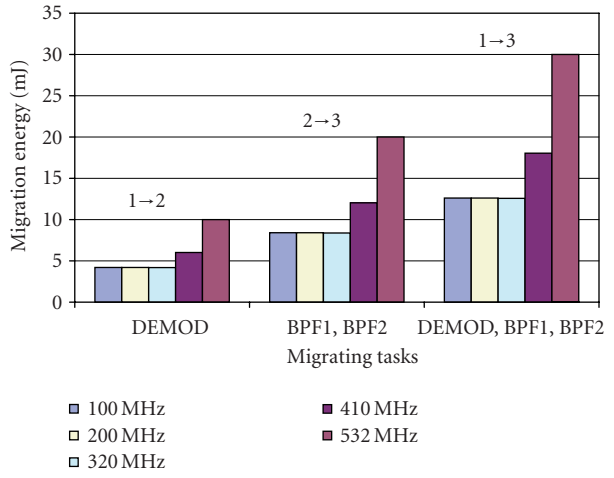


Figure 14: Task migration cost as a function of the frequency.

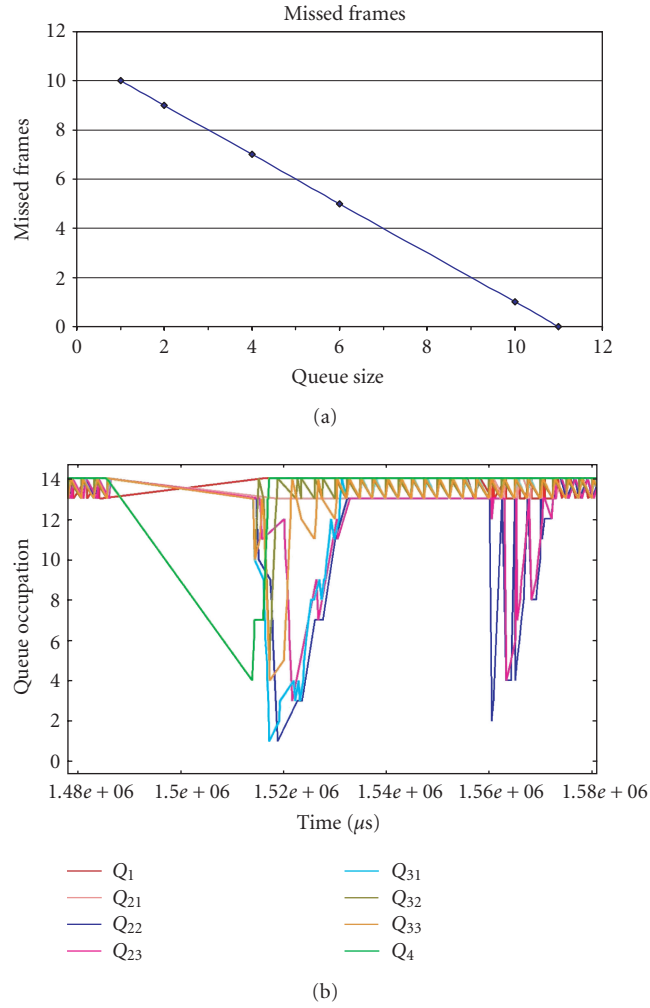


Figure 15: Impact of migration of communicating tasks: (a) Queue size versus deadline misses; (b) Queue occupancy during migration.

of bars shows the cost of migration from two cores to three cores, which implies moving the two BPF tasks from one core to another. The third set of bars shows the cost of migration from one core to three cores directly, which involves moving 4 tasks, the two BPFs, and the demodulator. It is worth noting that energy cost is almost equal from 100 MHz to 320 MHz. This is because in the power models we considered that voltage does not scale below 320 MHz, as reported in Figure 5. As such, dynamic power consumption of the processor linearly depends on the frequency and thus energy consumption is almost constant.

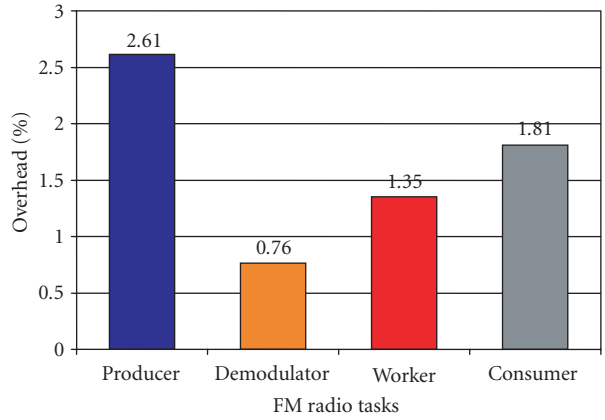
In the *shutdown-when-idle* case, migration implies an increase of the workload and thus less time in low-power state. In the *always on* case, migration cost is hidden if there is enough idleness. However, this depends on the relationship between frame rate (required workload), discrete core frequency and migration cost. In our experiment, migration overhead is completely hidden by the idleness due to frequency discretization. That is why migration costs are not shown in Figure 11.

```
while(1) {

   checkpoint();

   infifo.get (&elem);
   for(q=0; q < FM_FRAME_SIZE; q++)   {
       // demodulation
       temp = (elem.d[q] * lastElem);
      elem.d[q] = mGain *
              arctan(elem.d[q], 1.0/lastElem);
      lastElem = elem.d[q];
   }
   for(int w=0; w<n_workers; w++)   {
      outfifo[w]->put(&elem);
   }
}
```

(a)



(b)

Figure 16: Impact of migration of communicating tasks: (a) code of demodulator with checkpoints; (b) checkpointing overhead.

Migration overhead

Once energy-efficient configurations have been determined, these can be used to achieve the wanted frame rate in an energy-efficient way. However, when transitioning from a configuration to another, a migration cost must be paid. To assess the impact of this cost, in the following experiment we imposed a run-time variation of the required frame rate, leading to a variation in the optimal task/frequency mapping of the application. We considered to have an offline calculated lookup table giving the optimal frequencies/tasks mapping as a function of the frame rate of the application. The system starts with a frame rate of 450 fps. In this case, the system satisfies frame-rate requirements mapping all the tasks on a single processor running at the minimum frequency. As mentioned, scattering task on different processors is not profitable from an energy viewpoint in this case.

A frame rate of 900 fps is imposed at run time that almost doubles the overall workload. The corresponding optimal mapping consists of two processors each one running at the minimum frequency. Due to the workload distribution among tasks, the demodulator will be moved in another core where it can run alone while the other tasks will stay to ensure a balanced condition. Also in this case, further splitting tasks does not pay off because we are already running at the minimum frequency. As a consequence, the system reacts to the frame-rate variation by powering on a new processor and triggering the migration of the demodulator task from the first processor to the newcomer. During migration, the demodulator task is suspended, potentially causing a certain number of deadline misses, and hence, potentially leading to a quality of service degradation. This will be discussed later in this section. Further increasing the frame rate requires the migration of two workers on a third processor.

Whether or not the migration impacts QoS depends on interprocessor queue size. If the system is designed properly, queues should contain a data reservoir to handle sporadic workload variations. In fact, during normal operations, queue empty condition must be avoided and queue level must be maintained to a certain set point. In a real-life system this set point is hard to stabilize because of the discretization and the variability of producer and consumer rates, thus a practical condition is working with full queues (i.e., producer rate larger than consumer rate). When the demodulator task is suspended to be migrated, the queues between the demodulator and the workers start to deplete. Depending on the queue size, this may lead or not to an empty queue condition. This empty condition will propagate to the queue between the workers and the final consumer. If this queue becomes empty, deadline misses may occur.

In Figure 15(a), we show the results of the experiment we performed to evaluate the tradeoff between the queue size and the number of deadline misses due to task migration. The experiment has been carried on by measuring the deadline misses during the whole benchmark execution. The same measurement was performed by changing the size of the queues between the demodulator and the workers and the queue between the worker and the consumer (all having the same size). It can be noted that a queue size of 14 frames is sufficient to avoid deadline misses. Queue occupation during migration is illustrated in Figure 15(b). In this plot, Q_1 is the occupancy level of output queue of the first stage of the pipe(LPF), Q_{21} is the output queue of worker 1, and so on. It can be noted that the temporary depletion of intermediate queues does not cause frame misses. Indeed, the last queue of the pipeline (Q_4), responsible of the frame misses, never deplete.

In order to assess the overhead of migration support, we performed an additional test to quantify the cost of pure code checkpointing without migration. Since this overhead is paid when the system is in a stable, well-balanced configuration, its quantification is critical. It must be noted that checkpoint overhead depends on the granularity of checkpoints that in turn depends on the application code organization. For a software FM Radio application, we inserted a single checkpoint in each task, placed at the end of the processing of each frame. The results are shown in Figure 16(b). In Figure 16(a), we show as a reference the code of the part of the demodulator task with checkpoints. From the plot, we can see that the CPU overhead on the CPU utilization due to checkpoints in

case where no migrations are triggered is negligible, that is less than 1% also when for tasks that are checkpointed with a fine grain as for the producer or the consumer.

7. CONCLUSION

In this paper, we presented the assessment of the impact of task migration in embedded soft real-time multimedia applications. A software middleware/OS infrastructure was implemented to this purpose, allowing run-time allocation of tasks to face the dynamic variations of QoS requirements. Extensive characterization of migration costs have been performed both in terms of energy and deadline misses. Results show that the overhead of the migration infrastructure in terms of CPU utilization of master and slave daemons is negligible. Finally, by means of a software FM Radio streaming multimedia application, we demonstrated that the impact of migration overhead on the QoS can be hidden by properly selecting the size of interprocess communication buffers. This is because migration can be considered as a sporadic event performed to recover from an unbalanced task allocation due to the arrival of new tasks in the system or to the variation of workload and throughput requirements.

As future work, we plan to implement and test more complex task allocation policies, such as thermal and leakage aware. Moreover, we will test them with other multimedia and interactive benchmarks we are currently porting such as an H.264 encoder-decoder and an interactive game. Finally, from a research perspective we are interested in evaluating the scalability of the proposed approach by extending the current FPGA platform to allow the emulation of more processors and finally to implement a heterogeneous architecture exploiting the on-board PowerPC as master.

REFERENCES

[1] K. Eshraghian, "SoC emerging technologies," *Proceedings of the IEEE*, vol. 94, no. 6, pp. 1197–1212, 2006.

[2] Cradle Technologies, "Multi-core DSPs for IP network surveillance," www.cradle.com.

[3] "STMicroelectronics Multimedia Processors," www.st.com/nomadik.

[4] ARM Ltd, "ARM11 MPCore," www.arm.com/products/CPUs/ARM11MPCoreMultiprocessor.html.

[5] L. Friebe, H.-J. Stolberg, M. Berekovic, et al., "HiBRID-SoC: a system-on-chip architecture with two multimedia DSPs and a RISC core," in *Proceedings of the IEEE International Systems-on-Chip (SOC '03)*, pp. 85–88, September 2003.

[6] P. D. Van Wolf, E. De Kock, T. Henriksson, W. Kruijtzer, and G. Essink, "Design and programming of embedded multiprocessors: An interface-centric approach," in *Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and Systems Synthesis, CODES+ISSS*, pp. 206–217, Stockholm, Sweden, September 2004.

[7] C. Sanz, M. Prieto, A. Papanikolaou, M. Miranda, and F. Cathoor, "System-level process variability compensation on memory organizations of dynamic applications: a case study," in *Proceedings of the 7th International Symposium on Quality Electronic Design (ISQED '06)*, p. 7, March 2006.

[8] O. Ozturk, M. Kandemir, S. W. Son, and M. Karakoy, "Selective code/data migration for reducing communication energy in embedded MpSoC architectures," in *Proceedings of the ACM Great Lakes Symposium on VLSI, GLSVLSI*, vol. 2006, pp. 386–391, 2006.

[9] F. Bellosa, S. Kellner, M. Waitz, and A. Weissel, "Event-driven energy accounting for dynamic thermal management," in *Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP '03)*, New Orleans, La, USA, September 2003.

[10] A. Barak, O. La'adan, and A. Shiloh, "Scalable cluster computing with MOSIX for linux," in *Proceedings of the 5th Annual Linux (Expo '99)*, pp. 95–100, Raleigh, NC, USA, May 1999.

[11] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. Poggiali, "Supporting task migration in multi-processor systems-on-chip: a feasibility study," in *Proceedings of the Design, Automation and Test in Europe (DATE '06)*, vol. 1, Munich, DeutSchland, 2006.

[12] S. Carta, M. Acquaviva, P. G. Del Valle, et al., "Multi-processor operating system emulation framework with thermal feedback for systems-on-chip," in *Proceedings of the 17th Great Lakes Symposium on Great Lakes Symposium on VLSI (GLSVLSI '07)*, pp. 311–316, Stresa-Lago Maggiore, Italy, 2007.

[13] A. A. Jerraya, H. Tenhunen, and W. Wolf, "Guest Editors' introduction: multiprocessor systems-on-chips," *IEEE Computer*, vol. 38, no. 7, pp. 36–40, 2005.

[14] M. Ruggiero, A. Acquaviva, D. Bertozzi, and L. Benini, "Application-specific power-aware workload allocation for voltage scalable MPSoC platforms," in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, vol. 2005, pp. 87–93, San Jose, Calif, USA, October 2005.

[15] A. Kumar, B. Mesman, H. Corporaal, J. Van Meerbergen, and Y. Ha, "Global analysis of resource arbitration for MPSoC," in *Proceedings of the 9th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools (DSD '06)*, pp. 71–78, 2006.

[16] Z. Ma and F. Catthoor, "Scalable performance-energy tradeoff exploration of embedded real-time systems on multiprocessor platforms," in *Proceedings of the Design, Automation and Test in Europe (DATE '06)*, vol. 1, Munich, DeutSchland, March 2006.

[17] W.-L. Hung, Y. Xie, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "Thermal-aware allocation and scheduling for systems-on-a-chip design," in *Proceedings of the Design, Automation and Test in Europe (DATE '05)*, pp. 898–899, March 2005.

[18] F. Li and M. Kandemir, "Locality-conscious workload assignment for array-based computations in MPSOC architectures," in *Proceedings of the 42nd Annual Conference on Design Automation*, pp. 95–100, 2005.

[19] M. Kandemir and G. Chen, "Locality-aware process scheduling for embedded MPSoCs," in *Proceedings of the Design, Automation and Test in Europe (DATE '05)*, vol. II, pp. 870–875, 2005.

[20] E. Zayas, "Attacking the process migration bottleneck," in *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pp. 13–24, 1987.

[21] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process Migration Survey," *ACM Computing Surveys*, vol. 32, no. 3, pp. 241–299, 2000.

[22] D. Pham, "The design and implementation of a first generation CELL processor," in *IEEE/ACM ISSCC*, pp. 184–186, July 2003.

[23] J. Sakai, INOUE, and H. M. Edahiro, "Towards scalable and secure execution platform for embedded systems," in *Proceedings of the Design Automation Conference (DAC '07)*, pp. 350–354, Yokohama, Japan, January 2007.

[24] P. Francesco, P. Antonio, and P. Marchal, "Flexible hardware/software support for message passing on a distributed shared memory architecture," in *Proceedings of the Design, Automation and Test in Europe (DATE '05)*, vol. II, pp. 736–741, 2005.

[25] S.-I. Han, A. Baghdadi, M. Bonaciu, S.-I. Chae, and A. A. Jerraya, "An efficient scalable and flexible data transfer architecture for multiprocessor SoC with massive distributed memory," in *Proceedings of the Design Automation Conference (DAC '04)*, pp. 250–255, San Diego, Calif, USA, June 2004.

[26] M. Monchiero, G. Palermo, C. Silvano, and O. Villa, "Power/performance hardware optimization for synchronization intensive applications in MPSoCs," in *Proceedings of the Design, Automation and Test in Europe (DATE '06)*, vol. 1, Munich, DeutSchland, March 2006.

[27] uClinux, "Embedded Linux Microcontroller Project," 2007, www.uclinux.org/.

[28] Xilinx Inc, "Xilinx XUP Virtex II Pro Development System," http://www.xilinx.com/univ/xupv2p.html.

[29] W. Thies, M. I. Gordon, M. Karczmarek, et al., "Language and compiler design for streaming applications," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS '04)*, vol. 18, pp. 2815–2822, 2004.

Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 390106, 13 pages
doi:10.1155/2008/390106

Research Article

A Real-Time Embedded Kernel for Nonvisual Robotic Sensors

Enzo Mumolo,[1] Massimiliano Nolich,[1] and Kristijan Lenac[1,2]

[1] *DEEI, Università degli Studi di Trieste, 34127 Trieste, Italy*
[2] *AIBS-Lab S.r.l., Via del Follatoio 12, 34148 Trieste, Italy*

Correspondence should be addressed to Enzo Mumolo, mumolo@units.it

Received 5 April 2007; Revised 4 December 2007; Accepted 11 January 2008

Recommended by Alfons Crespo

We describe a novel and flexible real-time kernel, called Yartek, with low overhead and low footprint suitable for embedded systems. The motivation of this development was due to the difficulty to find a free and stable real-time kernel suitable for our necessities. Yartek has been developed on a Coldfire microcontroller. The real-time periodic tasks are scheduled using nonpreemptive EDF, while the non-real-time tasks are scheduled in background. It uses a deferred interrupt mechanism, and memory is managed using contiguous allocation. Also, a design methodology was devised for the nonpreemptive EDF scheduling, based on the computation of bounds on the periodic task durations. Finally, we describe a case study, namely, an embedded system developed with Yartek for the implementation of nonvisual perception for mobile robots. This application has been designed using the proposed design methodology.

1. INTRODUCTION

Embedded systems are hardware and software devices designed to perform a dedicated function, therefore, generally not allowing users to build and execute their own programs. Moreover, since embedded devices goal is to control a physical system, such as a robot, time factor is important, as are the size and cost. As a matter of fact, generally embedded systems require to be operated under the control of a real time operating system running on processors or controllers with limited computational power.

In this paper, we present a novel, flexible, real time kernel with low overhead and low footprint for embedded applications and describe an embedded application developed with it. The reason behind this development is the unavailability of an open source real-time operating system with the requested features. In fact, our objective was to realize small autonomous embedded systems for implementing real-time algorithms for nonvisual robotic sensors, such as infrared, tactile/force, inertial devices, or ultrasonic proximity sensors, as described for example in [1, 2]. Our requirements are: real-time operation with nonpreemptive scheduling, deferred interrupt mechanism, low footprint, and low overhead. Nonpreemptive scheduling is suitable to process nonvisual sensors, because most of these sensors use time of flight measurements which are cheaper to perform using polling rather than interrupt management. Furthermore, nonpreemption leads to a lower overhead, as requested by low performance microcontrollers. In addition to nonpreemption, also nonreal-time preemptive tasks are needed, especially for the communication with external devices. As a matter of fact, we interact with external devices through a serial port which is managed using interrupts and served by non-real-time tasks.

The kernel is called Yartek (yet another real time embedded kernel) and its source code is freely available online [3]. Yartek has been developed by modifying the scheduling module of another tiny operating system described in [4], and it is suitable for running on microcontrollers, since it uses a small amount of resources. In general, preemptive scheduling should be preferred over nonpreemptive policies in term of utilization factor. However, there are many applications where properties of hardware devices and software configurations make preemption impossible or expensive. In the application described in this paper, in fact, nonpreemptive management of the sensors leads to a cheaper utilization of computing resources. Moreover, the advantages of nonpreemptive scheduling are: an accurate response analysis, ease of implementation, no synchronization overhead, and reduced stack memory requirements. Finally, in general

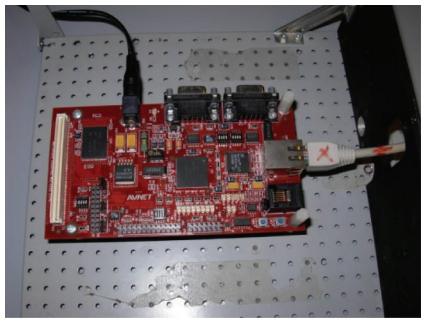


Figure 1: The Coldfire microcontroller.

nonpreemptive scheduling comes with lower-context switch overhead, since there are less interruptions.

In order to operate in unstructured environments, autonomous robots are equipped with a wide range of sensors—visual, like video cameras, and nonvisual, like ultrasonic or inertial sensors—and actuators. The raw sensorial data gathered from the sensors are processed in order to obtain a representation of the perceived environment. However, the robot is controlled by a processor with limited computational power due to the limited power supply of the mobile system. The motivation behind the development of Yartek was the need to build small, autonomous embedded systems which provide the processing requested by nonvisual sensors without imposing a computation burden on the main processor of the robot. In particular, the embedded system described in this paper provides the robot with the environmental map acquired with the ultrasonic sensors.

Yartek allows the creation and running of threads for fast context switch and it is based on a contiguous memory; moreover, it offers a dynamic memory management using a first-fit criterion. The threads can be real-time periodic scheduled with nonpreemptive EDF [5], or nonreal-time. In order to improve the usability of the system, a RAM-disk is included: it is actually an array defined in the main memory and managed using pointers, therefore its operation is very fast. The RAM-disk offers a file system structure for storing temporary data and executable code to enrich the amount of real-time applications which the kernel can run.

Yartek has been developed on a Coldfire microcontroller, in particular on the board having a MCF5282 microcontroller shown in Figure 1.

The main contributions of this paper are the following. The Yartek embedded kernel is introduced, and its performances and comparisons to a different real-time operating system are reported. A simple design methodology for nonpreemptive EDF scheduling is also described, based on bounds on the duration of nonpreemptive tasks. Finally, a real-time application is described, in the field of nonvisual sensor perception in robotics.

This paper is structured as follows. Section 2 summarizes the scheduling policies used in our kernel and proposes a design methodology useful for nonpreemptive tasks. Section 3 describes some technical aspects in the Yartek architecture. Section 4 deals with the performances of this implementation. Section 5 reports a case study where the nonpreemptive design methodology has been applied. Final remarks are discussed in Section 6. Some pieces of source code are reported in Appendices A, B, and C.

2. NONPREEMPTIVE REAL-TIME SCHEDULING

Real-time scheduling of a set of tasks with deadlines means that each task is executed within its deadline. This is a typical requirement of real time kernels for embedded systems, where missing a deadline may lead to an actuator misfunction or missing data during acquisition. We consider a periodic task as a set of instructions periodically invoked: the duration of the ith task is denoted C_i, and its period is denoted p_i.

Remark 1. We assume that time is discrete, and it is indexed by natural numbers because it is measured in clock ticks.

When possible, preemptive EDF is preferred over other strategies because it allows scheduling with high-utilization factors. However, there are many practical situations where nonpreemption of tasks is highly desirable. For example, there are cases where I/O devices make preemption impossible or expensive. Also, nonpreemptive real-time scheduling requires less overhead than preemptive because both synchronization primitives and deadline sorting at each task release are not necessary.

Remark 2. A fundamental parameter in real-time scheduling of n tasks is the utilization factor U: $U = \sum_{i=1}^{n} (C_i/p_i)$.

There are several authors who have presented some results on nonpreemptive scheduling [6]. The main difficulty with nonpreemptive scheduling is that it is, in general, a NP-complete problem [7] for every processor load [8]. In certain constrained cases, the NP-completeness can be broken, as shown by Jeffay et al. in [5] and Georges in [8] for EDF scheduling. In particular, Jeffay et al. show that necessary and sufficient scheduling conditions for a set of n nondecreasing periodic tasks, that is, $p_1 \le p_2 \le \cdots \le p_n$, are the following:

$$\sum_{i=1}^{n} \frac{C_i}{p_i} \le 1, \tag{1}$$

$$t \ge C_i + \sum_{j=1}^{i-1} \left\lfloor \frac{t-1}{p_j} \right\rfloor C_j \quad \forall 1 < i \le n, \ \forall t, \ p_1 < t < p_i. \tag{2}$$

In other words, informally, the EDF scheduling of the set of n periodic tasks is feasible if, according to the first condition, there is enough computational capacity to execute all tasks while, according to the second condition, the total computational demand in a temporal interval t is lower than the length of the interval itself.

2.1. Nonpreemptive design methodology

This section describes a design methodology which is based on the assignment of the computation times C_i of the

nonpreemptive tasks according to the physical requirements and subjected to suitable bounds. In other words, we seek the values of the bounds B_i so that if $C_i < B_i$, for all $i = 1, \ldots, n$, the set of periodic tasks is schedulable.

Starting from the Jeffay conditions (1) and (2), we now derive the bounds for the periodic tasks executions which bring the task set to be schedulable using the EDF nonpreemptive policy.

To this purpose, we can easily prove the following proposition which states a sufficient condition for a set of tasks to be schedulable.

Proposition 1. *If the computation times of a set of n nonpreemptive periodic tasks are bounded by B_i:*

$$B_i = p_1\left(1 - \sum_{j=1}^{i-1}\frac{C_j}{p_j}\right), \tag{3}$$

for $i = 1, \ldots, n$, then the set of tasks is schedulable using nonpreemptive EDF.

Proof. The bounds are a direct consequence of the condition reported in (2), which can be put in the following form:

$$C_i \le t - \sum_{j=1}^{i-1}\left\lfloor\frac{t-1}{p_j}\right\rfloor C_j, \quad \forall i = 2, \ldots, n, \ \forall t : p_1 < t < p_i. \tag{4}$$

If there is only one periodic task, then we can set $B_1 = p_1$. On the other hand, if there are two tasks ($n = 2$), then the condition reported in (4) becomes $C_2 \le t - \lfloor (t-1)/p_1 \rfloor C_1$, for all $t : p_1 < t < p_2$ or, in other words, $B_2 = \min_{p_1<t<p_2}(t - \lfloor (t-1)/p_1 \rfloor C_1)$. Since the possible values for t are: $p_1 + 1, \ldots, p_2 - 1$, we have $B_2 = p_1 + 1 - C_1$. Consider now the situation with i tasks. As before, we have

$$U_i = \min_{p_1<t<p_i}\left(t - \sum_{j=1}^{i-1}\left\lfloor\frac{t-1}{p_j}\right\rfloor C_j\right). \tag{5}$$

By considering the quantity

$$\overline{U_i} = \min_{p_1<t<p_i}\left(t - \sum_{j=1}^{i-1}\frac{t-1}{p_j}C_j\right), \tag{6}$$

which is the same as (5) without the floor operator, then $\overline{U_i} \le U_i$. This means that if $C_i \le \overline{U_i}$, the condition expressed in (2) surely holds. Now, we can easily find out that

$$\begin{aligned}\overline{U_i} &= \min_{p_1<t<p_i}\left(t - \sum_{j=1}^{i-1}\frac{t-1}{p_j}C_j\right)\\ &= \min_{p_1<t<p_i}\left(t\left(1 - \sum_{j=1}^{i-1}\frac{C_j}{p_j}\right) + \sum_{j=1}^{i-1}\frac{C_j}{p_j}\right)\\ &= (p_1+1)\left(1 - \sum_{j=1}^{i-1}\frac{C_j}{p_j}\right) + \sum_{j=1}^{i-1}\frac{C_j}{p_j}\\ &= p_1\left(1 - \sum_{j=1}^{i-1}\frac{C_j}{p_j}\right) + 1.\end{aligned} \tag{7}$$

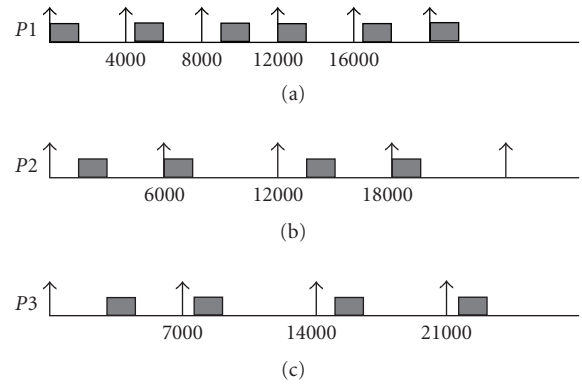


Figure 2: Scheduling example.

On the other hand, the first Jeffay condition, expressed in (1), can be stated as

$$C_i \le V_i = \left(1 - \sum_{j=1}^{i-1}\frac{C_j}{p_j}\right)p_i. \tag{8}$$

In conclusion, we can state that if $C_i \le B_i$, where $B_i = \min(\overline{U_i}, V_i) = \overline{U_i}$ as the p_i's are in nondecreasing order and so $\overline{U_i} \le V_i$, both the Jeffay conditions are satisfied, and the task set is EDF schedulable. It is worth noting that the opposite is not true, namely the condition is only sufficient.

This derivation completes the proof. □

From the proposition, we can immediately plan a design methodology for nonpreemptive real-time scheduling, consisting in finding the bounds of each tasks which guarantee scheduling of the task set, and setting the duration of the tasks within the bounds and according to the physical constraints. In other words, the physical system to be controlled through the real time kernel must have time constants less than the computed bounds. Otherwise, the architecture of the real time solution must be formulated in a different way.

To show how the above conditions can be used in practice, we have worked out the following example.

Example 1. Let us consider three tasks, with $p_1 = 4000$, $p_2 = 6000$, and $p_3 = 7000$. Then, $B_1 = 4000$, and assume that $C_1 = 1500$. Then, the bounds, which guarantee the tasks to be schedulable, are: $B_2 = 4000(1 - C_1/p_1) = 2500$. Assume then that $C_2 = 1500$. In the same way, $B_3 = 4000(1 - C_1/p_1 - C_2/p_2) = 1500$. Assume then that $C_3 = 1500$.

This scheduling is outlined in Figure 2.

Remark 3. The algorithm has a complexity of $O(n^2)$ divisions, where n is the number of tasks.

In fact, for $i = 2$, we have to compute one division, for $i = 3$, we have two divisions, and for the generic $i = n$, we have one product and $n - 1$ divisions. It is worth noting that complexity is not a critical problem, because the scheduling is statically designed.

```
MainLoop()   {
      while(true){
            if (InterruptTable is not empty)
                  ServiceInterruptTable();
            else
                  ServicetaskQueue();
      }
}
```

Pseudocode 1

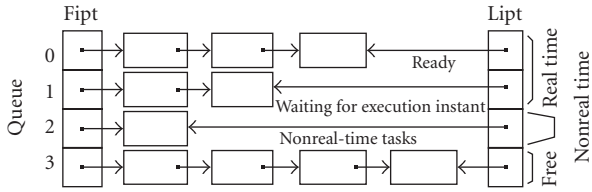


Figure 3: Yartek queues used for scheduling.

3. YARTEK ARCHITECTURE

Yartek has been designed according to the following characteristics:

(i) running on the Freescale MCF5282 Coldfire microcontroller [9];
(ii) nonpreemptive EDF scheduling of periodic real-time threads;
(iii) background scheduling of nonreal-time threads;
(iv) sensor data acquisition with a polling mechanism;
(v) deferred interrupt mechanism;
(vi) contiguous stack and data memory management using first-fit policy;
(vii) RAM-disk management;
(viii) system call primitives for thread, memory, and file management;
(ix) general purpose I/O management;
(x) communication with the external world via serial port.

3.1. Task scheduling

Task scheduling is one of the main activities of the operating system. All the scheduling operations are performed on the basis of a real-time clock, called RTClock, which is generated by an internal timer. Each task is represented using a data structure called thread control block (TCB), which is reported in Appendix A. TCB contains the name, type, and priority of the process, its allocated memory, and fields used to store the processor's state during task execution. For real-time processes, the TCB also contains *Start*, *Dline*, and *Period* fields to store the time when the process starts, its deadline, and its period. Scheduling is managed with a linked list of TCBs with 3 priority levels, as shown in Figure 3. There is one more queue used for storing free TCBs.

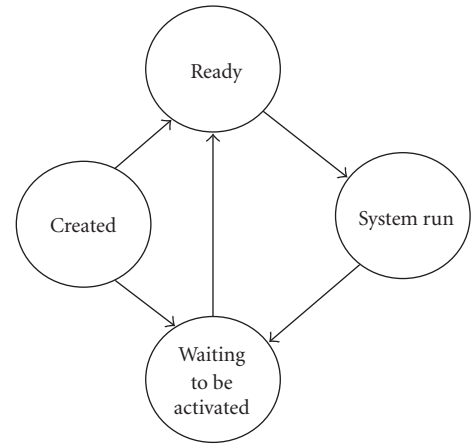


Figure 4: State diagram of a real-time thread in Yartek.

The periodic real-time threads are managed using nonpreemptive EDF scheduling. The TCB queues 0 and 1 are used as follows: queue 0 contains the TCBs of active threads, that is, ready to be scheduled and is ordered by nondecreasing deadline; hence these TCBs are executed according to the EDF policy. The periodic threads awaiting to be activated are inserted in queue 1 ordered by start time. As time lasts, some tasks can become active and the corresponding TCBs will be removed from queue 1 and inserted into queue 0. This operation is performed by the ServiceTaskQueue routine, which analyzes the TCBs on queue 1 to seek start times less than or equal to RTClock, that is, threads to be activated. Moreover, in queue 2 are stored the TCBs of nonreal-time threads. The queue 2 is managed using a FIFO policy.

The entry point of the kernel is an infinite loop where the interrupt table and the task queue are examined, as described in Pseudocode 1.

Interrupts are served using a deferred mechanism: each interrupt raises a flag on an interrupt table, reported in the Appendix A, and the ServiceInterruptTable routine checks the interrupt table to verify if a pending interrupt flag is set. In this case, it activates the suitable nonreal-time thread for serving that interrupt. In Appendix B, we report more detailed code.

The movement of a TCB from a queue to another at a given priority level is performed with a procedure which inserts the task in the task queue.

3.2. Process states

When a real-time thread is created (see Figure 4), it is in *Ready* state when *start time* > *RTClock* (TCB inside queue 0), it is in *Waiting to be activated* state when *start time* > *RTClock* (TCB inside queue 1). The first *Ready* thread will then be selected for execution and will go into *System run* state. When the execution stops, the process will become *Waiting to be activated* as it is periodic, and the scheduler updates its start time and its deadline adding them the thread period.

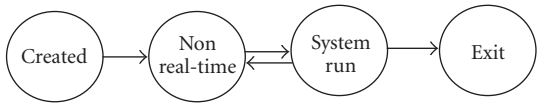


Figure 5: State diagram of a non-real-time thread in Yartek.

```
Interrupt_handler:
        set the flag in the operating system table;
        return from interrupt;
```

Pseudocode 2

The states of a nonreal-time thread are similar and are reported in Figure 5. The main difference is that a nonreal-time thread is preemptive, and it can be interrupted by real-time threads.

To create a process, a TCB is taken with GetTCB() function and filled with process information. The QueueTCB function then inserts the TCB in the requested queue. For queue 0, the TCB is inserted in ascending order according to deadlines, while for the queue 1 the TCB is inserted in ascending order according to activation time. For the other two queues, it is simply enqueued at the end.

3.3. Memory management

An amount of stack and data memory, containing thread-related information such as a local file table and information needed for thread management and user variables, is assigned to each process; furthermore, dynamic memory is also available when requested by system calls. Stack, data, and heap memory are organized in a sequence of blocks managed with first-fit policy.

3.4. System calls

A number of system calls have been implemented using the exception mechanism based on the trap instruction. The system calls are divided into file system management (open, read, write, close, unlink, rewind, chname), process management (exec, kill, exit), heap management (alloc, free), and thread management functions (suspend, resume).

3.5. Timer and interrupts

The microcontroller MCF5282 [9] has 4 programmable timers: one is used as the system's time reference, and it is used as RTClock, and the other timers are used for the measurement of time intervals. The timer is composed of a 16-bit register and a frequency divider. The first timer is used as the system's time reference, and it is used as RTClock. Since four interrupts are used for the timers, there are three interrupt levels for application code. The routines activated by interrupts set a single flag in the interrupt table. Later, the scheduler activates a process to actually manage the request, that is, in deferred mode. The pseudocode of an interrupt service routine is illustrated in Pseudocode 2.

4. PERFORMANCE EVALUATION

Generally speaking, as noted in [10], measuring real-time operating system performance and comparing a real time system to other real-time operating systems are difficult tasks. The first problem is the fact that different systems can have different functionalities, and the second concern is the method used to perform the actual measurements. Many features are worth to be measured: for example, Sacha [10] measures the speed of inter-task communication, speed of context switch, and speed of interrupt handling, while Garcia-Martinez et al. [11] reported measurements of responses to external events, inter-task synchronization and resource sharing, and inter-task data transferring. Finally, Baynes et al. [12] considered what happens when a real-time operating system is pushed beyond its limits; they also report real-time operating system power consumption measurements.

This section reports some measures used to describe the performance of Yartek, namely context switch time, jitter time, interrupt latency time, kernel stability, and kernel overhead. Yartek has been implemented on the Avnet board [13] part number ADS-MOT-5282-EVL, based on the Freescale MCF5282 ColdFire Processor running at 33 MHz. It is equipped with BDM/JTAG interface and has 16 MB SDRAM and 8 MB Flash. The communications are based on general purpose I/O (GPIO) on AvBus expansion connector, two RS-232 serial ports, 10/100 Ethernet. The performances obtained with Yartek have been compared to the performance of RTAI operating system, ported to this board.

4.1. RTAI

The RTAI project [14–16] began at the Dipartimento di Ingegneria Aerospaziale del Politecnico di Milano (DIAPM) as a plug-in which permits Linux to fulfil some real-time constraints. RTAI allows real-time tasks to run concurrently with Linux processes and offers some services related to hardware management layer dealing with peripherals, scheduling, and communication means among tasks and Linux processes. In particular, RTAI port for Coldfire microcontroller is a uCLinux [17] kernel extension that allows to preempt the kernel at any time to perform real-time operations. Unlike the implementations of RTAI for x86, PPC, and MIPS, the Coldfire version is not based on the "deferred interrupt mechanism" but uses the capability of the MC68000 architecture to priorities interrupts in hardware by the interrupt controller. In the current implementation, Linux interrupts are assigned lower-priority interrupt levels than RTAI interrupts. However, RTAI presents drawbacks which restrict the fields of integration:

(1) the preemptive scheduler works with static priorities and there is no built-in nonpreemptive EDF scheduler,
(2) both aperiodic and periodic tasks can be used but the priority of a periodic task bears no relation to its period,
(3) deadlines are not used,
(4) it is a hybrid system with high footprint.

```
void test_context_switch( ) {
  int i;
  CreateRTtask(thread, thread_code);
  ActivateTask(thread);
  t0 = StartTimer();
  for (i = 0, i < LOOPS; + + i){
   Resume(thread);
  }
  t1 = StopTimer();
}

void thread_code( ) {
  while (1) {
   Suspend();
  }
```

Pseudocode 3: Pseudocode of the context-switch test.

```
void test_jitter_time( ) {
  CreateRTtask(thread, thread_code);
  ActivateTask(thread);
}

void thread_code( ) {
  int loops=LOOPS;
  while (loops–) {
   t1[loops] = RTClock();
   t2[loops] = CurrentTCB->StartTime;
  }
}
```

Pseudocode 4: Pseudocode of the jitter time test.

4.2. Performance measures and comparisons

Experimental results with respect to kernel performances are based on the following parameters.

Context switch time

It is the time spent during the execution of the context switch routine of the scheduler.

Jitter time

It is the time delay between the activation time of a periodic real-time process and the actual time in which the process starts.

Deferred interrupt latency

It is the time delay between an interrupt event and the execution of the first instruction of the deferred task scheduling its service routine.

Kernel stability

It establishes the robustness of the kernel.

Kernel overhead

It represents the time the kernel spends for its functioning.

4.2.1. Test for context switch time

The test program used for measuring the context switch time is shown in Pseudocode 3.

The test program creates one thread and measures the time which lasts from the thread execution to subsequent suspension repeating the process *LOOPS* times. More precisely, in Yartek the only thing the thread does is to suspend itself, that is to put its TCB on the queue 1 while the loop calls the resume *LOOPS* times. *Resume*() moves the TCB on the queue 0 so it will be immediately scheduled. It is worth noting that *Resume*() is a blocking primitives, that is, it exits only when the TCB is put indeed in queue 0.

In RTAI, *rt_task_suspend*() suspends the execution of the task given as the argument, *rt_task_resume*() resumes execution of the task indicated as argument previously suspended by *rt_task_suspend*().

In Yartek, the average context switch time is 130 μs, while in RTAI is 124 μs.

4.2.2. Test for jitter time

It is worth recalling the detailed operation of Yartek to manage periodic real-time tasks. The TCB queue 1 contains the periodic threads, ordered by activation time, whose activation time is in the future. The starting time of the periodic tasks that lie on queue 1 is tested to detect the scheduling condition, that is, the starting time in TCB is greater than current time. In this case, the TCB is moved to the queue 0 which is sorted by deadline.

The test program used for measuring jitter time is shown in Pseudocode 4.

In Yartek, TCB is the data structure containing the data related to threads, and CurrentTCB is the pointer to the current thread. The system call RTClock() returns the value of the real-time clock used by Yartek for scheduling real-time tasks. However, in Yartek, real-time threads are activated by the ServiceTaskQueue routine, and therefore the exact starting time does not necessarily correspond to the start time written in the TCB. The delay can vary depending on the state of the routine. As a consequence, the uncertainty of activation time can be quite high. In fact, results of our tests show that Yartek has an average jitter time of 750 μs, while RTAI has an average jitter time of 182 μs.

4.2.3. Test for deferred interrupt latency

Yartek serves the interrupt using a deferred mechanism, that is, only a flag is raised immediately. A nonreal-time thread is activated only when the scheduler processes the Interrupt table.

```
void test_deferred_interrupt_latency( ) {
  CreateRTtask(thread1, thread_code1);
  ActivateTask(thread1);
}

void thread_code1( ) {
  CreateRTtask(Thread2, thread_code2);
  t1 = StartTimer();
  ActivateTask(thread2);
}

void thread_code2( ) {
  t2 = StopTimer();
}
```

Pseudocode 5: Pseudocode of the deferred interrupt latency test.

As RTAI does not implement on the Coldfire porting any interrupt service, we have estimated the time needed to schedule a task, thus implementing a deferred interrupt service.

So, the test program used for measuring deferred interrupt latency is schematically presented in Pseudocode 5. There are no periodic real-time threads in execution during the tests.

By using Yartek, an average deferred interrupt latency of 780 μs is obtained, while using RTAI a latency of 17 microseconds is obtained.

4.2.4. Kernel stability and overhead

Although we did not make a specific stability test, Yartek has shown a good robustness since it run for several hours both for performing applications and for evaluating the performance tests.

The time the kernel spends for itself and not for the application is mainly divided in scheduling time, context switch time, and memory management time. The essential code of scheduling is reported in the Appendix B.

In summary, the worst case complexity for interrupt management is about 110 assembler instructions to deferred schedule a nonreal-time thread for serving one interrupt. The management of the real-time task queues, under the hypothesis of one real-time TCB to be activated, requires about 150 assembler instructions. Regarding memory management, the alloc system call requires about 300 assembler instructions for allocating one block of contiguous memory using first fit and the free system call about 270 assembler instructions. These overheads express in time depend on the actual CPU frequency and for this reason have been left in number of instructions.

As previously computed, the overhead for context switch is 130 μs.

4.3. Discussion

The first important difference between RTAI and Yartek is the operating system footprint. The footprint of an operating system concerns the usage of RAM and flash memory resources. As noted in Section 4.1, the RTAI plug-in works with a Linux kernel. In our tests, we used uCLinux which has a minimum kernel size of 829 kbytes. The RTAI modules have a size of 97 kbytes, so the whole image is of about 900 kbytes. Instead, the footprint of Yartek is about 120 kbytes. This big difference in size is due to the fact that Linux plus RTAI brings some of the powerful tools and features of Unix. However, these tools are not necessary for an embedded system. The second difference between RTAI and Yartek is the nonpreemptive scheduling. It is worth remarking the adequacy of nonpreemptive scheduling in real-time embedded systems on low-power microcontrollers. RTAI does not offer nonpreemptive scheduling. Of course, it could be introduced by coding a new scheduler and integrating it in RTAI, but we instead decided to modify our previous kernel [4] for footprint reasons.

In conclusion, we decided to use Yartek for developing embedded solutions. The performance evaluation tests show that the time performances of Yartek are similar to RTAI for the context switch time and that the time for task creation is much lower for Yartek than RTAI. However, the jitter time for Yartek is much worst than RTAI, due to the Yartek architecture. It has to be noted, however, that the embedded systems for nonvisual sensors is not critical with respect to the jitter time, due to the time constants of such sensors. Yartek allows to manage task allocation that will be much more complex on a nonreal time or a sequential system providing periodic threads scheduled with a nonpreemptive EDF policy; the schedulability can be tested using the simple methodology presented in Section 2.

5. APPLICATION: EMBEDDED MAP BUILDING SYSTEM FOR MOBILE ROBOTS

In mobile robotics, several tasks require the strict satisfaction of time constraints, so real-time systems are needed. The sensors generally used for mobile robot navigation, such as inertial navigation systems, sonar sensor array, and GPS, laser beacons, should be processed considering real-time constraints. As map building is a fundamental task in mobile robotics, we considered an application of this type. Its design is described, and its implementation using Yartek is outlined in the following.

5.1. Previous work in map building for robot navigation

The problem of robotic map building is that of acquiring a spatial model of a robot's environment; this model is used for robot navigation and localization [18]. To acquire a map, a robot must be equipped with sensors that enable it to perceive the outside environment. Commonly used sensors include cameras, range finders using sonar, laser, and infrared technology, tactile sensors, compasses, and GPS. However, none of these sensors can furnish a complete view of the environment. For example, light and sound cannot penetrate walls. This makes it necessary for a robot to build a map of the environment while navigating in it.

Working in fusing multiple sensor readings for map building falls into two broad categories: target tracking models and occupancy grid models. In target tracking, one or more geometric features of the environment are modeled and tracked, that is, their location is estimated at each new sensor reading [19, 20].

Target-tracking methods are appropriate when there is a small number of targets, such as a few landmarks, and their interaction with the sensor is well known. A key issue in the target-tracking paradigm is the data-association problem: how to identify the target that a given sensor reading is associated with. While target-tracking is a good method for navigation using landmarks, in many situations it may be important to determine not just the position of a few landmarks, but the complete surface geometry of the environment.

The occupancy grid method [21–23] provides a probabilistic framework for target detection, that is, determining whether a region of space is occupied or not.

Elfes [21, 22] reformulated the problem as a probabilistic Bayesian updating problem using gaussian noise with a very large variance to account for the gross errors entailed by multiple reflections. He also addressed the problem of geometric uncertainty associated with sensor beam width by considering target detection under all possible configurations of the environment. In practice, given the overwhelming combinatorics of keeping track of data associations for each reading, independence and other simplifying assumptions are made to reduce the computational complexity of Bayesian update. That is, each cell of space is treated as an independent target in the presence of the geometric uncertainties induced by the beam width. This leads to unrealistic estimates for target map updates, for example, all the cells at the leading edge of the beam have their probabilities raised, when in fact usually only one cell is responsible for the echo.

Borenstein and Koren [24, 25] introduced the vector field histogram (VFH) method. They use a spatial histogram of sonar points, along the axis of the sonar beam, to identify areas that are likely to contain obstacles. The histogram is updated rapidly as new returns come in, and older ones are abandoned. The VFH method has the advantage that it can deal with dynamic and noisy environments. Since it is based on a continuous update of the map, this method is particularly suitable for mobile robots. The updating of the map can in fact be performed during the robot movement.

5.2. Implementation

Yartek has been installed on an embedded system composed by the Coldfire microcontroller board connected to an array of 6 sonar sensors placed in front of a mobile robot, and a PC-104 board on top of the robot, as shown in Figure 6.

The embedded system controls the ultrasonic sensors, calculates the robot position from the odometry readings, and updates the internal map. The last N_c sensor readings are stored internally in a circular list and used for updating the map with N_c set according to the odometry error model. The map is a histogram grid [24, 25] which is quite simple for a computational point of view. It is suitable for rapid in-

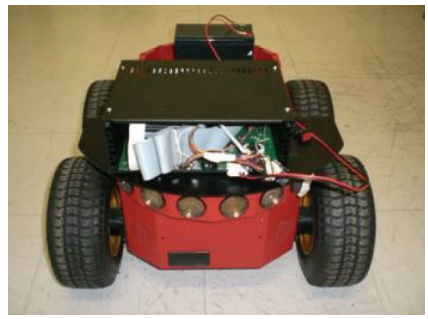


Figure 6: Mobile robot, type PIONEER 3-AT, equipped with the embedded system.

motion sampling of onboard range sensors and modeling of inaccurate and noisy range-sensor data, such as that produced by ultrasonic sensors. The system provides the internal map through a serial port and receives commands such as clear the map, compute the map, and send the map.

The application is designed as follows.

Sensor acquisition tasks

There are 6 real-time periodic tasks that perform the acquisition of the data from the sonar sensor and 2 real-time periodic tasks that read the odometers.

Map updating tasks

There is a real-time periodic task that updates the map according to the acquired sensorial data provided by the sensor acquisition task and antisensor real-time periodic task with larger period for filtering erroneous readings.

Map requests

These are aperiodic tasks scheduled upon requests. There is an aperiodic task that allows to obtain the complete map from the embedded system using serial line connection. There is also a task for clearing the map and resetting the robot's position.

There is one real-time task for each ultrasonic sensor that controls the firing of the sensor and waits for the reflected ultrasonic burst. This is compatible with the sonar array operation since only one sensor fires the ultrasonic burst at a time in order to avoid crosstalk. Crosstalk is a common problem occurring with arrays of multiple ultrasonic sensors when echo emitted from one sensor is received by another sensor leading to erroneous reading. In order to avoid crosstalk only one sensor emits the ultrasonic burst and listens for the echo. Only after the echo has been received or a maximum allowed time has elapsed in the case there is no obstacle in front of the sensor, the process can be repeated for the next sensor, and so on.

The range distance of the sensor is 3 m which is equivalent to a time of flight (TOF) of 17 milliseconds. We assume that the set of sensors is read every 500 milliseconds. This fact poses a physical limit on the maximum robot speed

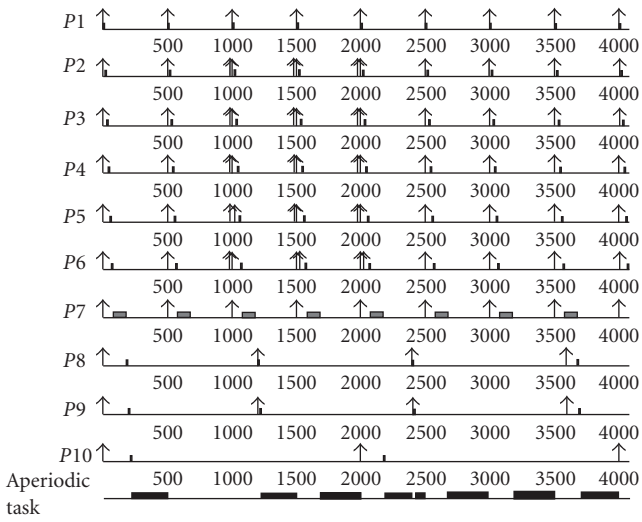


Figure 7: Time diagram of real-time scheduling for the map building application.

```
robot.getSonarN()
{
   fire ultrasonic sensor N (set logical 1 on digital pin)
   while time < maxtime and echo not detected
   {
        listen for echo (check the I/O pin of sensor N to go
                         to logical level 1)
        time++
   }
   calculate distance from time of flight
}
```

Pseudocode 6

```
robot.getOdoN
{
  read OdoN counter;
  diff=counter-previous;
  if diff<0 diff=maxcounter-previous+counter;
  calculate traversed distance from counter value;
  previous=counter;
}
```

Pseudocode 7

to approximately 0.5 m/s (if the robot is not to bump into obstacles and considering that minimum detectable distance is set to 20 cm). This is more than enough for our application. Finally, note that in this application the interrupts are generated only by the serial port.

The design methodology described in Section 2.1 has driven the development of this application. We call:

getSonar1, getSonar2, ..., getSonar6 the periodic tasks which read the sensors with period $p_1 = p_2 = \cdots = p_6 =$ 500 milliseconds; updateMap a periodic task with period $p_7 = 500$ milliseconds; getOdo1, getOdo2 periodic tasks with period $p8 = p9 = 1200$ milliseconds; antiSensor a periodic task with period $p_{10} = 2000$ milliseconds.

Also, we choose the worst case task duration as $C_1 =$ 20 milliseconds ($B_1 =$ 500 milliseconds), then $B_2 =$ 480 milliseconds. Choosing $C_2 = 20$ milliseconds, then $B_3 =$ 460 milliseconds. Choosing $C_3 = 20$ milliseconds, then $B_4 =$ 440 milliseconds. Choosing $C_4 = 20$ milliseconds, then $B_5 =$ 420 milliseconds. Choosing $C_5 = 20$ milliseconds, then $B_6 =$ 400 milliseconds. Choosing $C_6 = 20$ milliseconds, then $B_7 =$ 380 milliseconds. Choosing $C_7 =$ 100 milliseconds, then $B_8 =$ 280 milliseconds. Choosing $C_8 =$ 20 milliseconds, then $B_9 = 500(1 - 220/500 - 20/1200) = 271$ milliseconds. Choosing $C_9 =$ 20 milliseconds, then $B_{10} = 500(1 - 220/500 - 40/1200) = 263$ milliseconds. Finally we choose $C_{10} = 20$ milliseconds.

As the condition holds, then the set of task can be scheduled using nonpreemptive scheduling. In Figure 7, the corresponding time diagram of the nonpreemptive scheduling designed for the map building application is reported. In black is shown the part of times which can be dedicated to the aperiodic tasks.

The pseudocode for sonar real-time tasks is shown in Pseudocode 6.

Two other real-time periodic tasks read the odometers: the odometer increments the counter register for each tick of the wheel encoder. The task reads the register and compares it with the previous reading to find a traversed distance. The wrapping of the counter when passing from maximum value to 0 is easily handled since max wheel speed is such that relative distance to previous reading is limited (see Pseudocode 7).

The real-time periodic process that updates the map reads the sensor readings stored in the internal circular list, calculates the robot's position, and updates the cells in the histogram grid corresponding to sensor readings. The most recent readings increment the histogram grid cells while the oldest ones decrement the cells, that is, only the last Nc readings compare in the map. In this way the portion of the map with revealed obstacles moves with the robot (see Pseudocode 8).

As opposed to the updateMap task a second real-time periodic task called antisensor decrements all cells in a specified area around a robot (an area in front of the robot with range 1.5 m in the implementation). This mechanism is used in order to eliminate moving obstacles and to correct erroneous sonar readings. The antisensor task is scheduled with much larger period than the map updating task allowing for the cells in which real obstacles are present to reach high values before they get decremented.

The map transfer task is a real-time aperiodic task scheduled when a request for a map arrives. The array containing the map is then compressed with a simple (value, count) scheme and transmitted to a serial port running at 9600 bps. The number of bytes to be transmitted does not grow with the array size due to the updating mechanism discussed earlier where only the last readings compare in the map (see Pseudocode 9).

```
robot.updateMap();
{
      getLastOdoReadings();          // the most recent odometry readings in circular list
      calculateRobotPosition();
      getLastSonarReadings(); // the most recent sonar readings
      calculateSonarReadings();      // polar to cartesian based on relative sensor position
      updateMapPositive();           // increments histogram map cells

      getFirstOdoReadings();         // the oldest odometry readings
      calculateRobotPosition();
      getFirstSonarReadings();       // the oldest sonar readings
      calculateSonarReadings();
      updateMapNegative();           // decrements histogram map cells
}
```

Pseudocode 8

```
robot.sendMap()
{
  compressMatrixData();  // the map stored in memory array
  transferMatrixData();  // is transmitted to serial port
}
```

Pseudocode 9

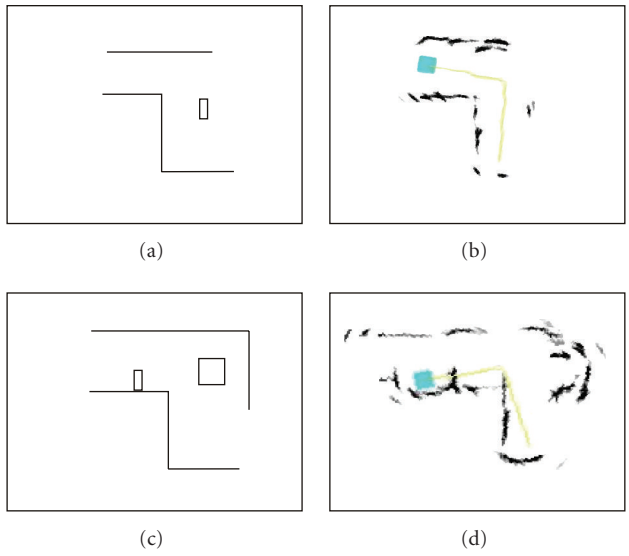


Figure 8: Map building results. On the left the real maps, on the right the maps estimated with the embedded system.

5.3. Experimental results

The bitmap estimated with the *updateMap()* method is referred to absolute coordinates and requires absolute localization results obtained with odometry. In other words, the map is an array of bytes where each element represents a cell of the grid by which the environment is divided. In our case, each cell represents a square of 50 mm long sides.

```
/*Components interfaces */
#include <TCB.h>    /* Thread Control Block */
#include <yartek.h> /* Scheduler */
#include <irq.h>    /* Interrupt management */
```

Algorithm 1

In Figure 8, some examples of maps obtained with the described embedded system are reported: on the left the actual map of the environment and on the right the perceived map obtained from the embedded device.

6. CONCLUDING REMARKS

In this paper, a real-time kernel we called Yartek, designed to implement embedded systems in mobile robotics, is described. It has been used to develop an embedded system for ultrasonic sensors-based environment mapping using the VFH algorithm. The embedded device we developed is currently used in our mobile robots: the ultrasonic sensors are used by the embedded system to compute an environmental map which is transferred to the robot without any other computational cost. The development of Yartek was motivated by the impossibility to find an open source real-time kernel for the Coldfire microcontroller used in our embedded system. Yartek is a small size kernel, which could be simply ported to other microcontroller architectures and implements nonpreemptive EDF scheduling. It also implements a deferred interrupt mechanism; by using this mechanism, interrupts are served by nonreal-time aperiodic tasks, which are scheduled in background.

Applications developed externally, that is, using cross-development systems, must be compiled inside the operating system. Therefore, a development environment must be available externally. Other features of the kernel include a thread management mechanism, dynamic memory management using first-fit, availability of a serial port driver which allows the connection of an external terminal for data

```
/* the TCB (Thread Control Block) type */
struct Tcb {char    SMI,        /* sending module */
            RMI,                /* receiving module */
            PR,                 /* task priority */
            PX,                 /* process index */
            DATA[4],            /* short data field */
            SR[2],              /* Cpu condition codes (SR) */
            *a[8],              /* Cpu address registers: A0–A7 */
            *d[8],              /* Cpu data registers: D0–D7 */
            *PC;                /* Cpu Program Counter */
     short  Stack,              /* Assigned Stack */
            Heap;               /* Assigned Heap */
     int    BornAbsTime,        /* Time of birth of process */
            Mem;                /* Memory address */
     char   ST_PR;              /* Starting task priority */
     void   (* fun)();          /* Pointer to the function to execute */
     unsigned int
            PID,PPID;           /* Process ID, Parent Process ID */
     CommandLine Command;       /* Record field for longer data */
     char   Name[NameDim];      /* Name of command or process */
     struct Tcb *CP;            /* Chain pointer */
     char   Type;
     unsigned int
            Born,               /* Born time */
            Start,              /* Next start time */
            Dline,              /* Deadline */
            Period,             /* Period of the thread */
            Maxtime,            /* Maximum duration of the thread */
            Stat;               /* Duration of the previous execution */
};
typedef struct Tcb TCB;
```

Algorithm 2

exchange and system monitoring purposes, and availability of a RAM-disk which provides a convenient data structure for temporary storage of information and for easily extending the operating system features.

Yartek has been developed for the Coldfire microcontroller family; if a porting to different architecture is to be performed, there are some short pieces of assembler code, mainly for the context switch and for timers management, which must be rewritten. It should be pointed out that Yartek could be simply modified for working both with preemptive and nonpreemptive scheduling. The main difference between nonpreemptive and preemptive scheduling is that in the preemptive modality some more operations should be performed to assure the preemption. In the preemptive case, another table is required to contain the TCB ID, the next activation time, and the task period. Before executing a TCB code, a timer is set in order to execute the scheduling when a periodic task arrives. The interrupt routine first updates this table, adding the period to the next activation time of the current task, it sorts the table by the next activation time in the ascending order, and then the scheduler is called.

In Appendix C, we report what should be done if one wants to adapt Yartek to another application.

The system we present in this paper is small, and it is highly reconfigurable. Almost all the kernel has been written in C language, and the source code can be freely downloaded [3]. The executable image takes less than 120 kbytes. We are currently developing other embedded systems for nonvisual sensors, namely inertial and infrared, for mobile robots.

APPENDICES

A. DATA STRUCTURES

In the following the Yartek data sructures are briefly summarized. To compile the application, a programmer needs to include headers of the required components (see Algorithm 1). The main data structure in Yartek is called TCB and is defined in Algorithm 2.

The Interrupt table is defined as follows:

```
int InterruptTable[3];          /* Interrupt Table */
```

B. EDF SCHEDULING

The scheduling algorithm of Yartek implements an EDF nonpreemptive scheduling. A representative C code of this routine for interrupt and real-time threads is reported in Algorithm 3.

```
void MainLoop()
{
   bool Found;
   register int i,  j;
   TCB *p, *p1, *p2;
mainloop:
      /* ServiceInterruptTable */
      /* deferred mechanism, nonreal-time threads are scheduled using Call */
      /* Call takes a TCB from the free list and put it on queue 2 */
      if (InterruptTable [0]! = 0)
      {
         Call( ServiceInterrupt0 );
      }
      if (InterruptTable [1]! = 0)
      {
         Call( ServiceInterrupt1 );
      }
      if (InterruptTable [2]! = 0)
      {
         Call( ServiceInterrupt2 );
      }

      /* ServiceTaskQueue */
      /* if a task to be activated is found on queue 1
         then concatenate task on queue 0 (using concatTCB function) */
      p=fipt[1];
        while (p!=NULL && p->Start<=RTClock)
      {
         fipt[1] = p->CP;    /* fipt: root of the queue 1 */
         if (fipt[1]==NULL) lipt[1]=NULL;
         p->PR = 0;
         p->ST_PR = 0;
         concatTCB(p);
         p = fipt[1];
      }
      if (p!=NULL)
      while (p->CP!=NULL)
         if (p->CP->Start<=RTClock)
         {
            p1=p->CP;
            p->CP=p1->CP;
            if (p->CP==NULL) lipt[1]=p;
            p1->PR = 0;
            p1->ST_PR = 0;
            concatTCB(p1);
         } else p=p->CP;
      if (Found==TRUE)
      {
       savedTCB=fipt[0];
      }
      if (savedTCB->CP == NULL)
         lipt[0] = NULL;    /* Updates the last element queue pointer */
      /* EXE execute the real-time thread */
      asm{
         jmp EXE;
      }
      /* When the real-time periodic thread has completed its period,
       then it is enqueued on queue 1 */
      concatTCB(savedTCB);
      goto mainloop;
}
```

Algorithm 3

C. HINTS FOR DEVELOPERS

A Yartek application developer should

(i) write the code for the interrupt service routines *ServiceInterrupt0*, *ServiceInterrupt1*, and *ServiceInterrupt2*;
(ii) write the *Init()* routine that schedules the periodic real-time threads;
(iii) write the code of the periodic real-time threads.

REFERENCES

[1] K. Lenac, E. Mumolo, and M. Nolich, "Fast genetic scan matching using corresponding point measurements in mobile robotics," in *Proceedings of the 9th European Workshop on Evolutionary Computation in Image Analysis and Signal Processing (EvoIASP '07)*, vol. 4448 of *Lecture Notes in Computer Science*, pp. 375–382, Valencia, Spain, April 2007.

[2] E. Mumolo, K. Lenac, and M. Nolich, "Spatial map building using fast texture analysis of rotating sonar sensor data for mobile robots," *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 19, no. 1, pp. 1–20, 2005.

[3] Yartek, http://www.units.it/smartlab/yartek.html.

[4] E. Mumolo, M. Nolich, and M. OssNoser, "A hard real-time kernel for motorola microcontrollers," *Journal of Computing and Information Technology*, vol. 9, no. 3, pp. 247–252, 2001.

[5] K. Jeffay, D. F. Stanat, and C. U. Martel, "On non-preemptive scheduling of periodic and sporadic tasks," in *Proceedings of the 12th Real-Time Systems Symposium*, pp. 129–139, San Antonio, Tex, USA, December 1991.

[6] W. Li, K. Kavi, and R. Akl, "A non-preemptive scheduling algorithm for soft real-time systems," *Computers and Electrical Engineering*, vol. 33, no. 1, pp. 12–29, 2007.

[7] M. Garey and D. S. Johnson, *Computer and Intractability, A Guide to the Theory of NP-Completeness*, W.H. Freeman Company, San Francisco, Calif, USA, 1979.

[8] L. Georges, P. Muehlethaler, and N. Rivierre, "A few results on non-preemptive real-time scheduling," Research Report 3926, INRIA, Lille, France, 2000.

[9] Freescale, "MCF5282 ColdFire Microcontroller Users Manual," November 2004 http://www.freescale.com/files/32bit/doc/ref manual/MCF5282UM.pdf.

[10] K. Sacha, "Measuring the real-time operating system performance," in *Proceedings of the 7th Euromicro Workshop on Real-Time Systems*, pp. 34–40, Odense, Denmark, June 1995.

[11] A. Garcia-Martinez, J. Conde, and A. Vina, "A comprehensive approach in performance evaluation for modern real-time operating systems," in *Proceedings of the 22nd EUROMICRO Conference (EUROMICRO '96)*, pp. 61–68, Prague, Czech, September 1996.

[12] K. Baynes, C. Collins, E. Fiterman, et al., "The performance and energy consumption of embedded real-time operating systems," *IEEE Transactions on Computers*, vol. 52, no. 11, pp. 1454–1469, 2003.

[13] AVNET, http://www.em.avnet.com/.

[14] P. Mantegazza, "Diapm rtai for linux: why's, what's and how's," in *Proceedings of the Real Time Linux Workshop*, Vienna, Austria, November 1999.

[15] RTAI, http://www.rtai.org/.

[16] M. Silly-Chetto, T. Garcia-Fernandez, and A. Marchand, "Cleopatre: open-source operating system facilities for real-time embedded applications," *Journal of Computing and Information Technology*, vol. 15, pp. 131–142, 2007.

[17] "uClinux Embedded Linux Microcontroller Project," http://www.uclinux.org/.

[18] S. Thrun, "Robotic mapping: a survey," in *Exploring Artificial Intelligence in the New Millenium*, G. Lakemeyer and B. Nebel, Eds., Morgan Kaufmann, San Fransisco, Calif, USA, 2002.

[19] J. Crowley, "World modeling and position estimation for a mobile robot using ultra-sonic ranging," in *Proceedings of IEEE International Conference on Robotics and Automation*, vol. 2, pp. 674–681, Scottsdale, Ariz, USA, May 1989.

[20] J. J. Leonard, H. F. Durrant-Whyte, and I. J. Cox, "Dynamic map building for an autonomous mobile robot," *International Journal of Robotics Research*, vol. 11, no. 4, pp. 286–298, 1992.

[21] A. Elfes, "Occupancy grids: a stochastic spatial representation for active robot perception," in *Autonomous Mobile Robots: Perception, Mapping, and Navigation*, S. S. Iyengar and A. Elfes, Eds., vol. 1, pp. 60–71, IEEE Computer Society Press, Los Alamitos, Calif, USA, 1991.

[22] A. Elfes, "Dynamic control of robot perception using multi-property inference grids," in *Proceedings of IEEE International Conference on Robotics and Automation*, vol. 3, pp. 2561–2567, Nice, France, May 1992.

[23] H. P. Moravec and A. Elfes, "High resolution maps from wide angle sonar," in *Proceedings of IEEE International Conference on Robotics and Automation*, vol. 2, pp. 116–121, St. Louis, Mo, USA, March 1985.

[24] J. Borenstein and Y. Koren, "Histogramic in-motion mapping for mobile robot obstacle avoidance," *IEEE Transactions on Robotics and Automation*, vol. 7, no. 4, pp. 535–539, 1991.

[25] J. Borenstein and Y. Koren, "The vector field histogram-fast obstacle avoidance for mobile robots," *IEEE Transactions on Robotics and Automation*, vol. 7, no. 3, pp. 278–288, 1991.