# Hardware Scheduling Support in SMP Architectures

André C. Nácul
Center for Embedded Systems
University of California, Irvine
nacul@uci.edu

Francesco Regazzoni
ALaRI, University of Lugano
Lugano, Switzerland
regazzoni@alari.ch

Marcello Lajolo
NEC Laboratories America
Princeton, NJ
lajolo@nec-labs.com

## Abstract

*In this paper we propose a hardware real time operating system (HW-RTOS) that implements the OS layer in a dual-processor SMP architecture. Intertask communication is specified by means of dedicated APIs and the HW-RTOS takes care of the communication requirements of the application and also implements the task scheduling algorithm. The HW-RTOS allows to have smaller footprints, since it avoids the need to link to the final executables traditional software RTOS libraries. Moreover, the HW-RTOS is able to exploit the easy task migration feature provided by an SMP architecture much more efficiently than a traditional software RTOS, due to its faster execution and we show how this significantly overcomes the performance achievable with optimal static task partitioning among two processors. Preliminary results show that the hardware overhead in a dual processor architecture is less than 20K gates.*

## 1 Introduction

Commonly, embedded applications are inherently parallel, and typically benefit from a multitasking programming environment and a matching hardware and software support. Multitasking simplifies coding, allowing a modularized solution and increasing code reuse. Support for multitasking can come from two sources: a software layer that multiplexes the hardware among the concurrent tasks, and direct hardware support for the execution of multiple tasks. Both solutions can be combined in a platform, with a software layer multiplexing a multitasking-capable hardware.

With the increase in silicon space, it is possible to use the extra transistors to improve the performance of the hardware platform. An appealing use of the extra space is to design a multiprocessor on a chip [5] [14]. Multiprocessor architectures have been present for some time in high-end server applications, more recently in consumer processors, and now are appearing in the embedded scenario as well. Designers can use the extra silicon space to accommodate multiprocessors and hardware resources that allow the execution of multiple tasks in parallel. The additional hardware support, either built in the processor or as a separate set of modules, can be used to accelerate multitasking management, increasing efficiency and freeing the processor from performing multitasking control.

Multiprocessor architectures are becoming fairly popular, with major processor industries announcing support for multiprocessors on a single chip, such as the processors from ARM (the MPCore family), MIPS (the 34k processor), and Analog Devices (the Blackfin family), and the corresponding software support from major embedded RTOS and tools providers, such as Express Logic's ThreadX, WindRiver's VxWorks, and Green Hills' development tools.

When implementing a multiprocessor architecture, different design decisions come to play, from high-level programming abstractions and compiler support to the number of processors, task scheduling and mobility and interconnection architecture. In this work, we experiment with an SMP architecture, with multiple processors combined, and sharing the memory architecture. We consider the effect that direct hardware support for task scheduling has on the performance of such architecture, as well as the impact of task mobility on the overall system performance. We also propose a hardware locking mechanism for implementing shared memory communication.

The remainder of this paper is organized as follows. Section 2 discusses some related work. Our proposed SMP architecture with hardware-assisted scheduling and communication is introduced in Section 3. Experimental results are described in Section 4. We present our final remarks and conclusions in Section 5.

## 2 Related Work

Recently, the interest in multiprocessor support on a single chip (MPSoC – MultiProcessor System-on-Chip) has peaked in the embedded community. The advantages provided by multiprocessor support are numerous, such as an increase in performance without a more complex base processor [14] [9], lower power consumption [10] [2], the possibility of locking processors to specific tasks [11], adding higher predictability to the system, among others.

Magarshack and Paulin [5] discuss the modifications necessary in tools and in the workflow support to address the rising complexity in the design of embedded devices. Wolf [14] looks into the mix of technologies related to multiprocessor support in embedded systems, and presents a list of related challenges. Among them, particular attention is given to the issues concerning the operating system and its communication primitives.

An interesting approach to the OS issues in multiprocessor systems explores hardware for implementing some key features of the RTOS. Previous works addressed the idea of moving to hardware the functionalities that consume more CPU power to take advantage of the acceleration. Mooney and Blough [6] present the $\delta$ framework. Their framework is targeted at speeding up locking and message passing mechanisms, but no scheduling is performed by the hardware. Lai et al. [3] propose hardware support in the multiprocessor context. However, they only provide queue management in hardware, while support for scheduling and task communication is in software.

Operating systems completely implemented in hardware have also been proposed, such as *Silicon OS*[8] and *FASTCHART*[4]. The former implements most of the $\mu$ITRON functionality on a coprocessor, while the latter is a real time kernel fully implemented on a hardware Real Time Unit, currently commercialized as the *Sierra* kernel [12].

Walder and Platzner [13] went further and developed an operating system based on reconfigurable hardware. They analyze multiple aspects in their work, ranging from design concepts, that are device independent, to the implementation of the proposed system on an FPGA.

In this work, we leverage hardware acceleration for speeding up the scheduling and the data handling of the RTOS. We focus on the homogeneous, shared memory multiprocessor architecture, using a shared bus as its interconnect structure, that is usually referred to as a Symmetric MultiProcessor (SMP). In SMP, all processors are equal, and access a single shared memory, where data and instructions are stored. Therefore, all processors execute the exact same code. Such symmetric approach also facilitates software development, as well as task allocation.

## 3 Proposed Architecture

The architecture proposed in this work is based on a traditional SMP system, and is shown in Figure 1. The SMP has two ARM926EJ-S processors, with their corresponding caches, a shared memory and a shared bus. We introduced a hardware locking module (Lock Unit) to control

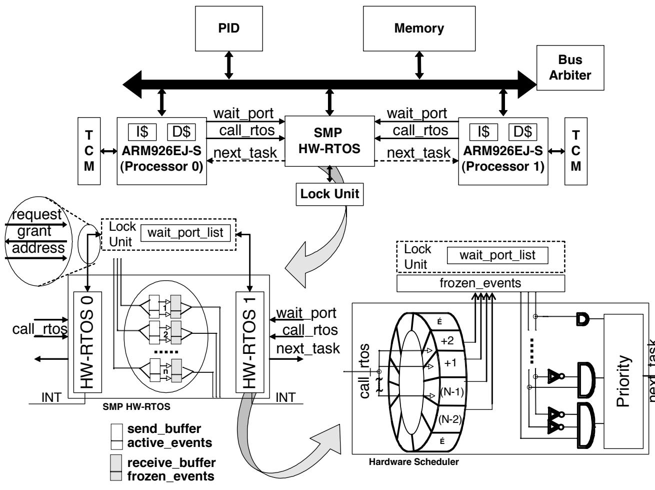


**Figure 1. SMP architecture: dual ARM926EJ–S and HW-RTOS**

access to the shared memory, so that a test-and-set operation on the shared memory can be performed by either processor. Additionally, task scheduling and communication is implemented by a hardware real time operating system (HW-RTOS).

For the few private processor data, such as the processor ID, we make use of the Tightly Coupled Memory available on the ARM processor. Specifically, each processor can access 2k of Data TCM, which is kept private for each processor. The TCM is directly connected to the dedicated TCM interface of each processor, and not to the shared bus. The processor ID is initialized as each processor boots into the operating system code by the Pid unit in Figure 1.

In our environment, a system is described as a set of concurrent, interacting tasks. Tasks are specified in a C-based system design language, and make use of dedicated APIs, based on POSIX, for task management and communication [7]. Two different communication models are supported: message passing and shared memory. Message passing is abstracted out by the concept of ports, and provides primitives *port_send* and *port_receive* to implement the communication. Blocking and non-blocking styles are supported for *port_receive*.

The final implementation of the APIs for communication and task management is transparent to the tasks. The same application can run in a system with traditional software libraries, as well as in an architecture with hardware accelerators in order to speed up execution. In our case, we have used the HW-RTOS to improve the efficiency of the OS and API support, transparently to the application. Furthermore, the same set of APIs can be used to specify tasks that can later be executed in a single or multi processed system, once again transparently to the user.

In this section, we detail the hardware support that was developed in order to implement the communication APIs in an SMP architecture.

## 3.1 The SMP HW-RTOS

The implementation of the HW-RTOS for the SMP architecture was based on our previous implementation in a single processor system [1]. The SMP hardware RTOS is composed of two independent scheduling modules, one for each processor in the architecture. Additionally, the SMP HW-RTOS contains a data handling module, with double buffering to store the data communication between tasks. Figure 1 details the organization of the SMP HW-RTOS.

**Communication Interface.** Each scheduling module of the SMP HW-RTOS communicates with the controlled ARM processor via dedicated ports. Three ports are used to connect each processor with the hardware scheduler, namely *call_rtos*, *wait_port*, and *next_task*.

**Scheduling Granularity.** Task scheduling and context switch can occur in two cases. First, a task can block when invoking a blocking *port_receive* call from the communication API. Alternatively, tasks can be preempted if they reach a pre-determined time slice.

**Context Switching.** When invoking a blocking *port_receive*, the blocked task will send the port on which it blocked, waiting for a communication, via the *wait_port* signal. The hardware scheduler maintains information regarding the port each task is blocked on in the *wait_port_list*. Immediately after, the task will trigger the hardware scheduler execution via the *call_rtos* signal. At this time, the hardware will compute the next task to be scheduled in the processor. In order to determine which tasks are schedulable, the hardware reads *wait_port_list*, as shown in Figure 1. When the scheduling module has computed the next task to be scheduled in the processor, it generates an interrupt to the processor, updates *wait_port_list*, and indicates the next task to be executed in the *next_task* port.

When a task is preempted for expiring the time slice, an interrupt is generated from the hardware scheduler, along with the next task indication in *next_task* port. The scheduler will not modify *wait_port_list*, because the task is not effectively blocked pending any communication. Instead, the preempted task is schedulable, and is considered by the scheduler in the next scheduling cycle. Note that when expiring the time slice, the task does not send any signal to the HW-RTOS (neither *call_rtos*, nor *wait_port*).

**Task Context Management.** Although the scheduling decision is performed efficiently in hardware, the proper context save and restore has to be handled by software. The software part of the task switching mechanism services the interrupt request generated by the HW-RTOS, saves the processor state for the current task to the shared memory and restores the processor state of the next task from the shared memory. This step of task scheduling has to be performed in the processor, because it involves reading and writing of the register file and status words, which is not accessible to the HW-RTOS without software intervention.

The context of a task is always saved to the shared memory space. Therefore, it is accessible by any processor, effectively enabling task migration. Specifically for the ARM9 architecture, we have reserved 1Kb of space per task in the shared memory to store a task stack with the context of each task. The values of general purpose registers (R0-R10), followed by FP, IP, LR, PC and SPSR registers, are stored in the task's stack before it is preempted from the processor. Additionally, the task's stack pointer SP is stored in a dedicated array, which has one entry per system task, also in the shared memory.

**Task Communication.** Task communication is managed by the data handling module of the SMP HW-RTOS. Port communication between tasks is controlled by a double buffered scheme. Tasks will always write to the *send_buffer*, while they read from the *receive_buffer*. Similarly, every write will result in an event to be stored in *active_event* buffer. Whenever a task $T_1$ blocks in a *port_receive*, all of $T_1$'s communications will be copied from the *send_buffer* to the *receive_buffer*, and immediately become available to all other tasks. Additionally, the corresponding *active_event* entries are copied to *frozen_event*, indicating the presence of a new communication event. If any task $T_2$ is waiting on a port that was written by task $T_1$, then $T_2$ will be eligible to be scheduled in the next scheduling cycle. Currently, the scheduling module performs round robin scheduling. Other policies can be supported, without changes to the interface between the HW-RTOS and the processor.

Note that, while there are multiple hardware scheduling modules, one for each processor, there is only one data communication module managing communication from and to every processor. Therefore, there is exactly one copy of *send_buffer*, *receive_buffer*, *active_event*, and *frozen_event*.

**Shared Memory Lock Unit.** A dedicated hardware module is included in the proposed architecture to allow a test-and-set instruction to be implemented. This is an important operation to support shared memory communication in multiprocessor systems, as it allows a task to read and subsequently write to a shared memory location without concurrency from other tasks. In our implementation, the lock unit is used to provide test-and-set support in *wait_port_list* for the SMP HW-RTOS.

For each scheduling module in the SMP HW-RTOS, the Lock Unit contains one request and one grant bits. The particular address to be locked is specified in the address field. The Lock Unit is a memory-mapped device, so modules can access the bits by reading and writing memory addresses. The implementation of the Lock Unit can be extremely efficient. The Lock Unit takes a single cycle to assert grant bits after the request bit and address are set.

**Locking API Primitives.** As with communication primitives, tasks use our dedicated API primitives to request locks in the shared memory, specifically *shared_memory_lock* and *shared_memory_unlock*. Note that it is the job of the programmer to ensure locking and unlocking requests are properly present in the code. Our architecture will not automatically detect shared memory access conflicts. Additionally, the lock unit is designed to allow the implementation of a test-and-set instruction, and is not an explicit mutex primitive. Instead, mutexes can be built on top of test-and-set. Therefore, it is guaranteed that no context switch happens while performing a test-and-set. For this reason, the lock unit has one entry per processor in the architecture, instead of one request/grant line per task.

**Conflict Resolution.** The Lock Unit implements a priority mechanism to resolve conflicts in shared memory access. If both modules request exclusive access to the same shared memory address, the module with the lowest ID will be granted access to the detriment of the other. In our implementation, the scheduler module connected to processor with ID 0 has higher priority than the module connected to processor with ID 1.

**Task Migration.** The shared memory in the SMP architecture facilitates task migration, or dynamic task scheduling. All task context information is saved in the shared memory. Therefore, it can be retrieved by any other processor when a task is resumed. It is the scheduler's job to decide whether a task can migrate to another processor, or should resume execution in the same processor it was last executed. Alternatively, the scheduling of tasks to processors can be static, i.e., each task can run only in a single, and pre-determined processor.

Each approach has its trade-offs. When tasks migrate, processor resources are better utilized, since any task can be scheduled in any processor. Consequently, all tasks can run, as long as there is a processor available. On the other hand, there is a penalty on cache misses. While in the static scheduling case, there is a chance that task data will still be present in the processor's cache, when tasks migrate, the cache on the new processor will have to be filled with the task's data from the main memory. Our experiments evaluate this trade-off. Results are presented in Section 4.

## 4 Experimental Results

We provide implementations of the SMP HW-RTOS with and without support for task migration. When there is no task migration, each of the hardware scheduling modules will be provided with a static set of tasks from which to choose when scheduling. With task migration enabled, all tasks are available to be scheduled by all scheduling modules. In this case, tasks are scheduled in a first-come, first-served fashion, i.e., when a task is available to be scheduled, it will be scheduled in the first processor that becomes available, regardless of the processor in which it executed last.

The experimental environment is comprised of in-house behavioral hardware synthesis tool (Cyber) and cycle accurate simulator (ClassMate). Hardware synthesis produces synthesizable RTL for each hardware module. Software modules are compiled by a gcc cross-compiler.

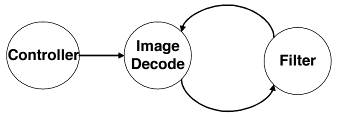
ClassMate platforms are built as compositional blocks, each of which models a specific part of the hardware/software architecture, including processors, memories, caches, buses, interfaces, and arbiters. Additionally, it is possible for the designer to provide application specific modules to be integrated in the simulation environment. We used this resource to implement simulation versions of the SMP HW-RTOS and the Lock Unit. The simulator provides cycle accurate results about the execution of the processor, as well as dedicated hardware modules modeled by the designer. Additionally, it also gives information about software execution and communication interfaces.

**Platforms.** To evaluate the performance of the proposed architecture, we modeled the SMP system, with and without task migration enabled. Additionally, we also simulated a traditional single processor, using eCos as a complete software RTOS, and a single processor with HW-RTOS.

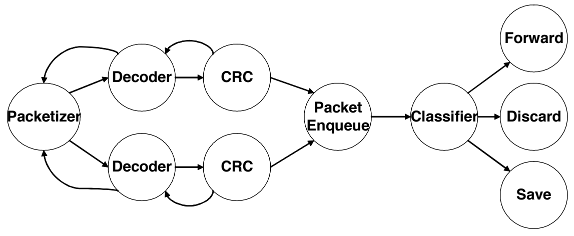
**Benchmarks.** We used two applications as benchmarks to perform our tests. The first is a graphic filtering application, representing typical operations performed in the kernel of multimedia applications. The task graph shown in Figure 2(a) has three tasks, namely a `Control` to coordinate the execution of the tasks, the `Image Decoder`, which fetches image data from memory and decodes it, and the proper `Filter` task, which executes the filtering algorithm and produces the filtered image.

The second application used in the experiments is a network packet processing engine, which receives packets from a wireless media, processes and classifies each packet, routing it to its destination. Its task graph is shown in Figure 2(b). The first task, the `Packetizer`, will receive the data from the medium and organize it in packets. Decoding and CRC checking is performed in each packet by two separate tasks, `Decoder` and `CRC`. To allow higher parallelism, there are two Decoder and CRC checker task instances. Following is the `Packet Enqueue` task, the `Classifier` and the final output tasks, which can `Forward` the packet to another destination, `Store` it in the main memory, or `Discard` the packet under special conditions.

**Results.** We simulated the architectures for one million cycles after processor initialization was complete. Table 1 shows the total number of pixels processed in the image filtering benchmark for each architecture, as well as the total number of packets processed in each architecture.



(a) Image Filtering



(b) Packet Processing

**Figure 2. Application Benchmarks**

**Table 1. Experimental Results**

| | SW-only | HW-RTOS | | |
|---|---|---|---|---|
| | eCos | single | SMP static | SMP dynamic |
| Image Filtering | | | | |
| Pixels | 6 | 278 | 340 | 330 |
| Packet Processing | | | | |
| Packets | 3 | 30 | 40 | 43 |
| Idle Cycles | 0 | 0 | 238555 | 22688 |

Note that task migration increases the throughput of the packet processing benchmark by 7.5%. However, there is a negative impact on the performance of the image filter, which executes 1.2% slower. This is due to cache effects. Since the image filter has a small number of tasks, migrating from one processor to the other affects the cache hit ratio, and reduces the performance. The effect does not happen in the packet processing because of the larger number of tasks, which will cause cache misses in both cases. In fact, the performance increases due to the more efficient use of the processor cycles (i.e., a reduction of idle cycles). In general, if the application is small enough to fit in the cache, there is a loss in performance by allowing migration, which is the case with the image filter benchmark.

Table 1 shows a significant performance increase due to the use of our HW-RTOS module. While a solution based on traditional software RTOS, in our case eCos, processes only 6 pixels in the simulated time (i.e., one million processor cycles), the same single processor architecture with the addition of HW-RTOS can process 278 pixels. Similarly, the software RTOS can process 3 packets, while the single processor with HW-RTOS can process 30, and the SMP with HW-RTOS and task migration can process 43.

We can also observe that the SMP architecture improved the throughput of the image filter by 23% in the static scenario and 22% in the scenario with task migration. Similarly, the throughput of the packet processor improved by 33% in the static allocation and by 43% with task migration, when compared to the single processor with HW-RTOS.

The HW-RTOS is used to accelerate the context switch and communication between tasks. While in the software only RTOS based on eCos takes approximately 10,000 cycles to complete a context switch, the proposed hardware scheduling module can execute the same context switch in 947 cycles. The context switch is detailed on Figure 3(a). It shows all the stages of the context switch, performed by the HW-RTOS and by the software.

Figure 3(b) shows the task scheduling in the SMP architecture with task migration disabled. The first two rows show the tasks running on each processor of the SMP. The two bottom rows show the scheduling in the HW-RTOS units. Every pulse in the waveform represents the next task computation procedure in the HW-RTOS. Tasks are always scheduled in the same processor, and it is possible to notice an idle period on processor 1. Similarly, Figure 3(c) shows the trace of task scheduling with task migration enabled. Note that some tasks are actually executing in both processors, such as tasks Packetizer, Decoder 0, and Enqueue.

Finally, we observe that the performance gain provided by the HW-RTOS comes at a small cost. Using Synopsys Design Compiler, we estimated the area overhead to implement the extra hardware components for HW-RTOS support to be about 20k gates in the SMP architecture.

## 4.1 Future Developments

There are some topics related to the HW-RTOS and task scheduling and migration that we plan to explore in the near future. Specifically, we want to explore different strategies for task migration, as well as verifying the scalability of our approach as the number of tasks and processors increases.

**Task Migration.** Currently, our approach to task migration is either enabled for all tasks, or for no task. It would be interesting to experiment with different strategies for migration, specially one involving selective migration. In this case, task migration decision would consider where the task executed last, and other issues such as timing constraints and current available tasks to perform migration.

**Scalability.** Our HW-RTOS solution supports a fixed number of tasks and a specific number of processors. Increasing the number of tasks impacts linearly on the memory requirements for the HW-RTOS, since more resources are needed to store task context and the *wait_port_list* structure. Increasing the number of processors that are managed by the HW-RTOS has a direct impact on the area requirements for the HW-RTOS, along with an increase in the number of possible parallel accesses to the shared memory. While the extra area is due to the scheduling modules for each processor, and could be managed in the design, the increasing strain on the memory could limit the solution to a

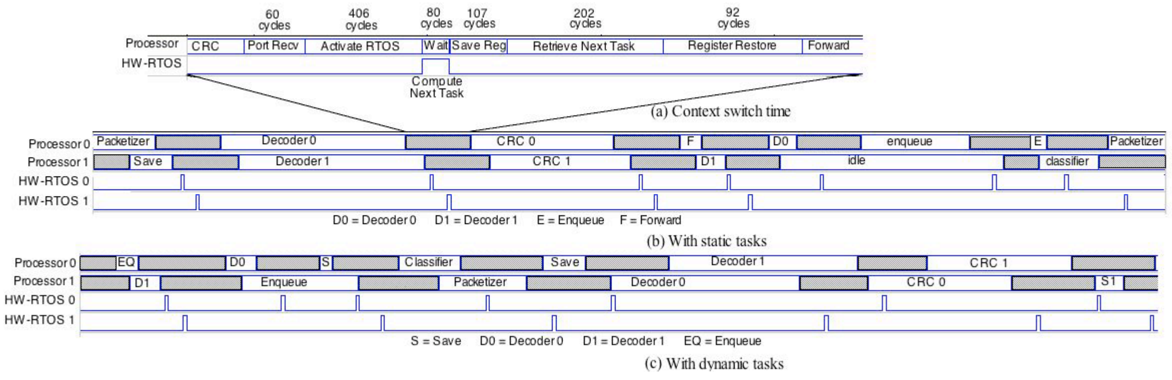


**Figure 3. Task scheduling in SMP**

certain number of processors.

**Cache Coherence Protocols.** Data cache coherence is an important issue. With static scheduling, all data that are shared among tasks in different processors need to be coherently updated. One possibility is to use the data cache only to non-shared data. This way, shared data always have to be fetched from the main memory. When tasks can migrate, however, cache coherence is essential, as even non-shared data need to be correctly transferred from one processor to another when tasks migrate. We are investigating the impact of different cache coherence protocols on the proposed architecture, and evaluating the impact each has in the system performance and its effect in task migration.

**Dynamic Task Priorities.** Given the complexity of current embedded systems, dynamic priority can expand the set of applications that benefits from the HW-RTOS solution.

## 5 Conclusions

In this work, we presented a SMP architecture with dedicated hardware support for tasks scheduling and communication. We also provided a quantitative analysis of the proposed SMP architecture, comparing them to single processor architectures. We also showed the impact of some design decisions regarding SMPs, specifically the ability to schedule a task in different processors, supporting task migration, and the impact of a shared bus in the performance of an SMP. Results show that our SMP architecture with HW-RTOS support is significantly more efficient than a SW only solution. Additionally, we were able to improve performance of the system by adding a second processor and the corresponding modifications in the HW-RTOS.

## References

[1] S. Chandra, F. Regazzoni, and M. Lajolo. Hardware/software partitioning of operating systems: a behavioral synthesis approach. In *Proc. of ACM GLSVLSI*, pages 324–329, 2006.

[2] T. Kogel and H. Meyr. Heterogeneous mp-soc – the solution to energy-efficient signal processing. In *Proc. of DAC*, 2004.

[3] B.-C. C. Lai and I. Verbauwhede. A light-weight cooperative multi-threading with hardware supported thread-management on an embedded multi-processor system. In *Proc. of Asilomar Conference on Signals, Systems, and Computers*, 2005.

[4] L. Lindh and F. Stanischewski. Fastchart-idea and implementation. In *ICCD*, pages 401–404, 1991.

[5] P. Magarshack and P. Paulin. System-on-chip beyond the nanometer wall. In *Proc. of DAC*, 2003.

[6] V. J. Mooney III and D. Blough. A hardware-software real-time operating system framework for socs. *IEEE Design & Test of Computers*, 19(6):44–51, 2002.

[7] A. C. Nacul, M. Lajolo, and T. Givargis. Interface-centric abstraction level for rapid hardware/software integration. In *Forum on Specification and Design Languages*, 2005.

[8] T. Nakano, A. Utama, M. Itabashi, A. Shiomi, and M. Imai. Hardware implementation of a real-time operating system. In *Proc. of the 12th TRON Project International Symposium*, pages 34–42, 1995.

[9] P. Paulin and C. Pilkington. Application of a multi-processor soc platform to high-speed packet forwarding. In *Proc. of DATE*, 2004.

[10] R. Sasanka, S. Adve, Y.-K. Chen, and E. Debes. The energy efficiency of cmp vs smt for multimedia workloads. In *Proc. of ACM International Conference on Supercomputing*, 2004.

[11] P. Schaumont, B.-C. C. Lai, W. Qin, and I. Verbauwhede. Cooperative multithreading on embedded multiprocessor architectures enables energy-scalable design. In *Proc. of DAC*, 2005.

[12] Sierra:. `http://www.realfast.se/RFIPP/-products/sierra/sierra.shtml`.

[13] M. Walder, Herbert Platzner. Reconfigurable hardware operating systems: From design concepts to realizations. In *Proc. of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pages 284–287, 2003.

[14] W. Wolf. The future of multiprocessor systems-on-chips. In *Proc. of DAC*, 2004.