SCMP Architecture: An Asymmetric Multiprocessor System-on-Chip for Dynamic Applications

Nicolas VENTROUX and Raphaël DAVID
CEA, LIST,
Embedded Computing Laboratory,
91191 Gif-sur-Yvette CEDEX, France
nicolas.ventroux@cea.fr

ABSTRACT
Future systems will have to support multiple and concurrent dynamic compute-intensive applications, while respecting real-time and energy consumption constraints. Within this framework, this paper presents an architecture, named SCMP. This asymmetric multiprocessor can support dynamic migration and preemption of tasks, thanks to a concurrent control of tasks, while offering a specific data sharing solution. Its tasks are controlled by a dedicated HW-RTOS that allows online scheduling of independent real-time and non-real-time tasks. By incorporating a connected component labeling algorithm into this platform, we have been able to measure its benefits for real-time and dynamic image processing.

Categories and Subject Descriptors
C.1.4 [**Computer System Organization**]: Processor architecture - Multiprocessors; C.3 [**Special-purpose and application-based systems**]: Real-time and embedded systems; D.4.7 [**Software**]: Operating Systems - Organization and Design, Embedded Systems

Keywords
multiprocessor, MPSoC, real-time, adaptive computing, HW-RTOS, image processing

1. INTRODUCTION
In response to an ever increasing demand for computational efficiency, the performance of embedded system architectures have improved constantly over the years. This has been made possible through fewer gates per pipeline stage, deeper pipelines, better circuit designs, faster transistors with new manufacturing processes, and enhanced instruction-level or data-level parallelism (ILP or DLP). An increase in the level of parallelism requires the integration of larger cache memories and more sophisticated branch prediction systems. It therefore has a negative impact on the transistors' efficency, since the part of these that performs computations is being gradually reduced. Switching time and transistor size are also reaching their minimum limits. Besides, only low ILP is possible in a given sequence of processor instructions (within a same sequence, instructions tend to be highly dependent) [1, 2]. As a result, superscalar processors register diminishing returns as they attempt to execute more instructions per clock cycle, while the logic required to process these multiple cycle instructions drastically increases. Besides, existing sophisticated software approaches with VLIW processors do not reach better results, due to inevitable parallelism limitations.

The emergence of new embedded applications for telecom, automotive, digital television and multimedia applications, has fueled demand for architectures with higher performances, more chip area and power efficiency. These applications are usually compute-intensive, which prevents them from being used in general-purpose processors. They must be able to simultaneously process concurrent information flows; and their information must all be efficiently collected, dispatched and processed. This is only feasible in a multithreaded execution environment. Designers are thus showing interest in a System-on-Chip (SoC) paradigm composed of multiple computation resources and a network that is highly efficient in terms of latency and bandwidth. The resulting new trend in architectural design is the MultiProcessor SoC (MPSoC) [3]. MPSoCs are multithreaded architectures and, as such, support the integration of multiple computing resources. These resources concurrently execute threads using explicit multithreaded workloads [4, 5]. They rely on several approaches, the most important of which are simultaneous multithreading (SMT), chip multiprocessing (CMP) and chip multithreading (CMT).

Another very important feature of future embedded computat-intensive applications is the dynamism. For instance, the connected component algorithm [6] is one of them and its computation time depends on the size and the number of handled objects. It is very interesting to analyze its computation time on a complete video sequence. Figure 1 shows these results.

This algorithm is highly data-dependent and the execution time depends on the image content. Consequently, on a multiprocessor platform, optimal static partitionning can-

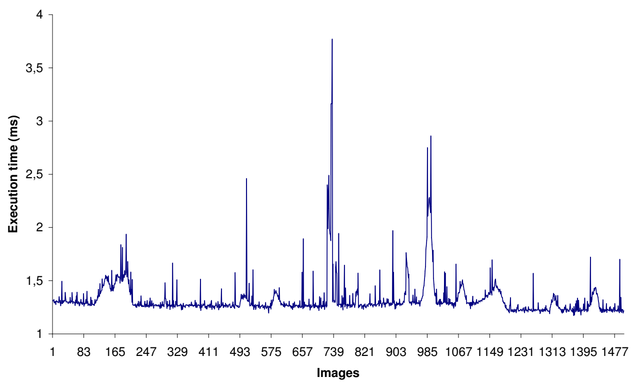


Figure 1: Execution time of the connected component labeling algorithm on a video sequence with an Intel Pentium 4 Xeon processor (2.99 GHz).

not exist since all the processing times depend on the given data. The only optimal solution consists in dynamically allocating tasks according to the availability of computing resources. In this paper, a task is an independent sequence of instructions. Moreover, to share the computation power between concurrent real-time applications or tasks, it is necessary to support the preemption and migration of tasks. If a task has a higher priority level than another one, it must preempt the current task to guarantee its deadline. Besides, the preempted task must be able to migrate on another free computing resource to increase the performance efficiency of the architecture.

Unfortunately, existing architectures offer only partial solutions to the power, chip area, performance, reliability and dynamism problems associated with embedded systems. For this reason, our paper presents an alternative solution. In section 2, we describe existing multithreaded approaches and why they are unsuited to embedded systems. Then, section 3 details our own architecture, which is called SCMP. This CMP-type architecture supports task migration and preemption with very few timing penalties. Section 4 presents its control system, which is a real-time hardware operating system accelerator dubbed OSoC. This component accommodates a high degree of control parallelism and integrates an online dynamic scheduler that supports both real-time and non-real-time tasks. Section 5 focuses on computation, by explaining our concurrent, prefetched configuration mechanism, as well as the solution adopted for data sharing management. Section 6 presents the SCMP simulator that is used to obtain the implementation results of the connected component labeling algorithm in our architecture, which are elaborated in section 7. This paper expects to highlight the impact of the OSoC on the whole system, and to show the benefit that can be obtained by using a hardware OS accelerator with multiprocessor platforms. Finally, section 8 concludes this paper by discussing work already underway on this system, along with that to come.

2. RELATED WORK

Use of thread-level parallelism (TLP) is becoming necessary for embedded applications. It calls for simultaneously executing multiple threads within single execution units (SMT), or dispatching threads to separate physical processing units (CMP). Another approach, called CMT, consists of merging both these capabilities.

Simultaneous MultiThreading -. SMT architectures are multithreaded processors that interleave the execution of instructions from different threads in the same pipeline [7]. In this configuration, multiple program counters are available in the fetch unit, and multiple contexts are saved in local registers. The penalties, that occur during execution of a single instruction stream, due to cache miss or data dependencies, are filled by computations of another thread. This technique has been integrated into solution-server architectures such as MARS-M [8], Compaq Alpha 21464 [9], and IBM's Power5 [10]. Xeon processors also have a simultaneous multithreading technology known as hyperthreading (HT) [11]. Hyperthreading makes a single physical processor appear as two logical processors. Only state resources, such as general-purpose registers, are then duplicated to permit concurrent execution of two control threads. Because an SMT processor simultaneously exploits coarse- and fine-grained parallelism, namely TLP and ILP, it can use its resources more efficiently, achieving better throughput and speedup than single-threaded superscalar processors for multithreaded workloads (Figure 2-a). Its implementation nevertheless requires complex issue stages and generates huge architectures that are not feasible for embedded systems, with their demanding energy consumption and computing density constraints. Moreover, control-dominated processes are executed like regular, time-critical processes within the same computation unit. Consequently, computational resources must support both regular and irregular processes, and this prevents their optimization for embedded applications.

Chip MultiProcessing -. Contrary to SMT architectures, this solution is widely used in embedded systems, due to its ease of design [12]. A CMP architecture is composed of computing resources driven by a single control unit, which allocates to them the ready-to-be-executed tasks (Figure 2-b). In this paper, we consider a task as a divided part of a thread. This is done to extract parallelism within any one thread, while executing multiple threads in parallel across multiple processors. The CMP approach minimizes the size of the final architecture, since it attempts to exploit a higher degree of TLP, using more processors instead of larger issue widths within a single processor. As a result, its structure is adapted to embedded constraints; and its high degree of TLP can meet future application needs. Such architectures can integrate heterogeneous or homogeneous resources. Typically, heterogeneous structures are dedicated to a specific applicative domain. Because the order of task execution is defined at compile-time, task dispatching is simplified at runtime. Among these application-driven solutions, we note Texas Instruments' OMAP [13], the VIPER architecture developed by Philips [14], and ST Microelectronics' Nomadik platform [15]. Homogeneous structures, on the other hand, can either be dedicated to certain applications, as is the case of the CT3616 proposed by Cradle Technologies [16], or support general-purpose processing like IBM's CELL [17] and

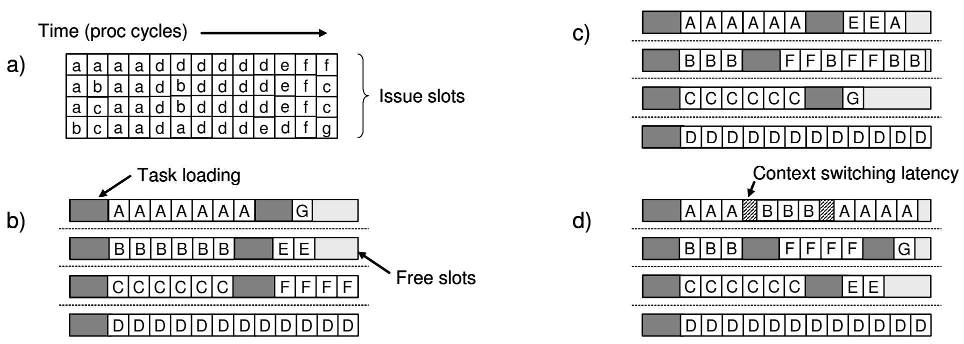


Figure 2: Different approaches to multithreaded task execution: (a) 4-issue SMT; (b) 4-way CMP; (c) 4-way CMT; and (d) 4-way CMP with task migration and preemption. Each line represents the issue slots for single execution cycles.

ARM's MPCore platforms [18].

Chip MultiThreading -. This third multithreaded technique is a combination of CMP and SMT [19]. It supports the execution of simultaneous threads within each distributed processing unit (Figure 2-c). In the same way as for certain SMT architectures, each time a thread is stopped by a cache-miss or a data dependency, for instance, the execution unit executes another thread. In CMT, unlike CMP architectures, the penalty for task switching is reduced to a local context switch. However, like SMT, CMT is only suited to server applications, because of its lower computational density and energy efficiency. The UltraSparc IV [20] or the UltraSparc T3 [21] processor are some example of CMT multiprocessors.

Previous works [7, 5] have shown that SMT performances are better than those of CMP for the same number of issue slots. This is mainly due to poorer use of computation resources in CMP than in SMT, since an n-way SMT has the same theoretical performances as n processing elements with only one issue. In CMP architectures, processing elements mainly stay idle when thread-level parallelism is low. Dynamic partitioning of SMT resources among threads also affords efficient use of both ILP and TLP, depending on the application's properties. Within SMT architectures, communication between threads does not imply the same significant penalties as for CMP. With the CMP approach, the control unit and the processing elements are distributed and are linked by standard system buses that are inefficient in terms of latency and bandwidth. Their latencies penalize the reactivity of the architecture and prevent the whole system from optimizing its resources. For a constant chip area and integration density, however, the complexity of 16-issue CMP design is no greater than that of a 12-issue superscalar or SMT processor [12]. With its higher thread-level parallelism potential, the CMP execution model thus outperforms other multithreaded techniques. While CMP architectures have drawbacks, they are definitely recommended for embedded systems. They consume less power, are easier to implement and afford better performances than other multithreaded approaches, for a same degree of complexity [22].

In conclusion, none of these architectures completely solves the problems raised by embedded systems. The CMP approach can meet embedded constraints, but suffers from a lack of optimization. SMT architectures exhibit good performances at a high level of parallelism, but are not designed for the embedded system market. Previous use of the alternative CMT approach has shown how difficult it is to exploit both these advantages without combining their drawbacks. CMT hybrid architecture provides a good compromise between the excellent performance of SMT and the high energy efficiencies of CMP, but is hindered by large chip areas and energy consumption overheads. The solution proposed in this paper affords an attractive trade-off between multithreaded approaches. At the same time, unlike CMT or existing CMP, our approach is dedicated to dynamic embedded applications. Our architecture is called SCMP. It is a Scalable Chip MultiProcessor having a loosely-coupled interface with the CPU. The SCMP is based on a CMP execution model that supports dynamic migration and preemption of tasks, can hide control latencies and optimizes use of computation resources.

3. SCMP ARCHITECTURE

The SCMP architecture has a CMP structure and uses migration and fast preemption mechanisms to eliminate idle execution slots. As shown in Figure 2-d, our execution model is similar to the CMT approach. While this means bigger switching penalties, it ensures greater flexibility and reactivity for real-time systems. Take the example of task B (on Figure 2-d), which has a higher level of priority than task A but a lower level of priority than task F. In the CMT model, task B execution must wait until the end of task F, i.e. until the necessary resources are freed. Thanks to preemption and migration, in SCMP, task B can continue being executed on another processing unit. The order of priority for task execution is then respected, which is not the case for other multithreaded approaches, which execute tasks according to resource or data availability.

3.1 Programming model

The programming model for the SCMP architecture is specifically adapted to dynamic applications and global scheduling methods. The proposed programming model is based on the explicit separation of the control and the computation parts.

As depicted in Figure 3, each application must be manually (the tool chain is still under development) parallelized and cut into different tasks. Thus, computation tasks and the control task are extracted from the application, so as each task is a standalone program. The control task handles the computation task scheduling and other control functionalities, like synchronizations and shared resource management for instance. Each embedded application can be divided into a set of independent threads, from which explicit execution dependencies are extracted. Each thread can in turn be divided into a finite set of tasks. The greater the number of independent and parallel tasks are extracted, the more the application can be accelerated at runtime.

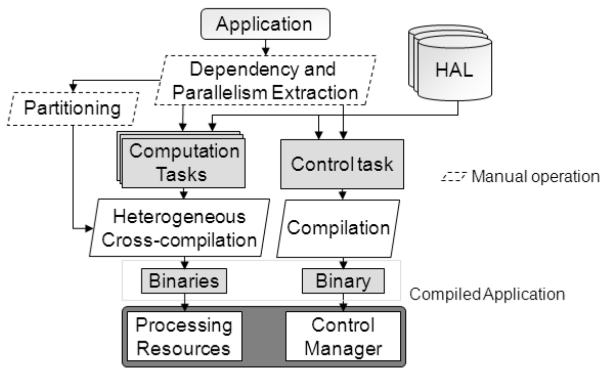
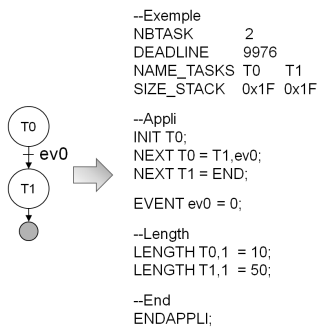


Figure 3: Programming model

A specific Hardware Abstraction Layer (HAL) is provided to manage all memory accesses and local synchronizations, as well as dynamic memory allocation and management capabilities. With these functions, it is possible to carry out local control synchronizations or to let the control manager taking all control decisions. Concurrent tasks can share data through local synchronizations to enable a dataflow execution, or wait for the controller decision before reading input data. Each task is defined by a task identity, which is used to dialog between the control and the computation parts. Then, a manual partitioning must be carried out in case of heterogeneous MPSoCs. Heterogeneous resource management takes place before task compilation. Finally, all tasks are compiled and made available to the processing resources or the control manager.

The control task is a Control Data Flow Graph (CDFG) extracted from the application, which represents all control and data dependencies. The control task handles the computation task scheduling and other control functionalities, like synchronizations and shared resource management for instance. A specific and simple assembly language is used to describe this CDFG and must be manually written. As depicted Figure 4, each control task, for each different application, needs to define the number of computation tasks, the binary file names corresponding to these tasks, and their necessary memory stack size. Then, because their is no instruction order, we must specify which are the first and end tasks of the application. We must also describe each transition with the sentence. Finally, for real-time scheduling, the deadline of the application, as well as the worst case execution time of each task, must be defined. The processor type of each task is also specified and this information is used during the allocation process. Only this representation of the control task is necessary, whereas a specific compilation tool is used for the binary generation.



```
--Exemple
NBTASK        2
DEADLINE      9976
NAME_TASKS  T0    T1
SIZE_STACK   0x1F  0x1F

--Appli
INIT T0;
NEXT T0 = T1,ev0;
NEXT T1 = END;

EVENT ev0 = 0;

--Length
LENGTH T0,1  = 10;
LENGTH T1,1  = 50;

--End
ENDAPPLI;
```

Figure 4: Control task assembly code example. The next task *T1* is activated when the event *ev0* from the current task *T0* is received by the controller

3.2 Execution model

Our execution model calls for executing constrained tasks on computing resources. Two execution models are supported: a *contrained-task* or a *streaming* execution model. Both of them are simultaneously supported.

The first one allows the execution of tasks when all previous tasks have finished their execution and therefore have produced their intermediate results (Figure 5). All tasks are uninterrupted by data or control dependencies. During its execution, a task cannot access data not selected at the extraction step. It cannot use work-in-progress data, and it must follow the execution order established by the control unit. The execution of non-blocking tasks starts as soon as all input operands are available. Consequently, the model eliminates data coherency problems without the need for specific coherency mechanisms. This constitutes an important feature for embedded systems, since their architecture is accordingly simplified.

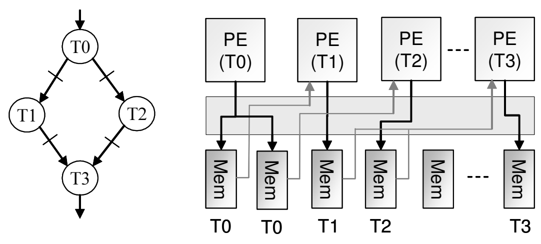


Figure 5: Task-constraint execution model. Reads and writes are non-blocking. Each arrow represents an interconnection through the data network between a Processing Element (PE) and a data stored in local memories (Mem). Dark arrows are read/write accesses, whereas gray arrows represent read-only accesses. Task execution requires only data produced by tasks that have finished their execution on a PE.

We have also the possibility to share data with other concurrent tasks via a dynamic buffer allocation. In this model, a task can wait for data produced by another task in the dataflow pipeline (Figure 6). The granularity of the shared data is the page. Since all tasks have an exclusive access to pages, we eliminate data coherency problems without the need for specific coherency mechanisms.

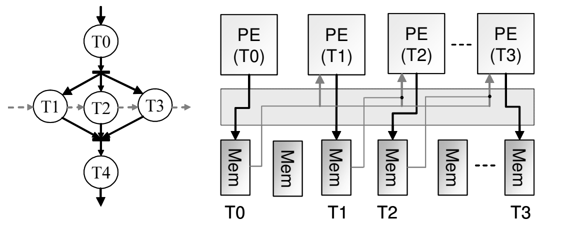


Figure 6: Streaming execution model. Each arrow represents an interconnection through the data network between a Processing Element (PE) and a data stored in local memories (Mem). Dark arrows are read/write accesses, whereas gray arrows represent read-only accesses. Task execution begins as soon as intermediate data are ready. Local synchronization is afforded through a memory management unit.

However, the data sharing problem between processing resources still remains. It is linked to the structure of memory resources. With a single shared memory, concurrent accesses create overheads, due to its incapacity to provide sufficient bandwidth. With a distributed memory system, shared data are dispatched and additional data transfers generate heavy penalties and reduce overall performances. Our architecture must also support dynamic migration of tasks. The solution therefore consists of sharing all distributed resources. In our approach, tasks nevertheless maintain exclusive access to their memories throughout execution (Figure 5 and 6). This is guaranteed, since all the executed tasks are constrained by the others or by the memory management unit; and data modifications by other concurrent tasks does not need to be taken into account. The result of this solution is high functional and temporal predictability, which is important for embedded systems.

This execution model requires a dedicated controller with an efficient scheduling mechanism. Existing control solutions could not ensure the high degree of parallelism or the same reactivity as a hardware solution. For this reason, we have designed a hardware real-time operating system called Operating System accelerator on Chip (OSoC). This component can manage all the distributed resources (processing and storage) of our architecture and dynamically dispatch tasks under real-time constraints. The SCMP architecture thus comprises the OSoC, multiple computing, memories and Input/Output (I/O) resources.

3.3 SCMP structure

The SCMP architecture is a compute-intensive resource that is seen by the CPU as a coprocessor (Figure 7). The software operating system (OS) hosted by the CPU is commonly used for general-purpose processing or interface management. All control of intensive processes must nevertheless be performed by an efficient control system. While using a simple OS would be adequate, this solution is prohibited by parallelism, reactivity and determinism considerations. Because our system has centralized control and implies permanent communication between its control unit and its processors, reactivity also significantly impacts overall performances. Moreover, software OS structures based on interrupts do not allow determinism, since the response to an interrupt depends on OS activity. The various hardware abstraction layers, between the OS and the hardware likewise penalize performance and overall system reactivity. This generates critical sections during hardware/software communication. Use of dedicated hardware components is thus vital to our system. Many acceleration solutions for real-time operating systems (RTOS) have been proposed and their benefits clearly demonstrated [23, 24, 25].

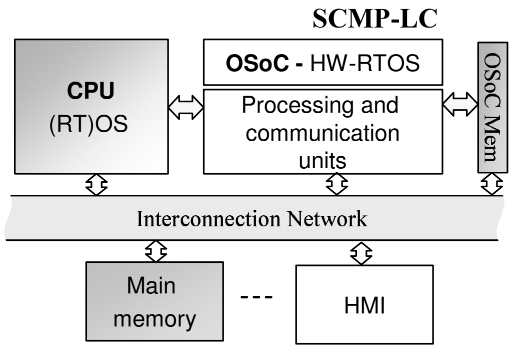


Figure 7: SCMP architecture: The HW-RTOS or OSoC controls multiple computing resources dynamically to afford intensive computation capability.

To ensure an efficient control of our multiple computing resources, we have designed a hardware real-time operating system named OSoC (Operating System accelerator on Chip). The OSoC dynamically determines the list of eligible tasks to be executed, based on control and data dependencies. It also manages exclusive accesses to shared resources, and non-deterministic processes. Then, task allocation follows online global scheduling, which selects real-time tasks according to their dynamic priority, and minimizes overall execution time for non-real-time tasks. Dynamic power management exploits the processor's dynamic voltage and frequency scaling (DVFS) [26], and sleep mode mechanisms. In addition, the OSoC manages memory allocations and the exclusive sharing of physically distributed and logically shared memory space. It also prefetches tasks in these memories so that the beginning of the execution of the task do not suffer from memory latencies. Heterogeneous resource management takes place before task compilation.

The CPU can accelerate applications through the SCMP architecture at runtime. It communicates with the OSoC via the interconnection network used in the system (Figure 7). It can ask for execution of a new application, stop or suspend it, or wake up a suspended application. Periodic tasks are automatically synchronized and deadlines of tasks are updated. Multiple applications and multiple instances of a same application can be executed and managed concurrently by the OSoC. To execute a new application, the CPU must load all necessary instructions or data for the application into the OSoC Memory. When the transfer is completed, the CPU informs the OSoC and sends an execution order. After execution of the application, an acknowledgement is sent to the CPU.

Computing resources can be heterogeneous. This affords better chip area and power efficiency for our system. Heterogeneity may concern either software or hardware. It is possible to implement general-purpose or dedicated processors (DSP, VLIW, etc.), coarse-grained reconfigurable resources (ADRES [27], DART [28], XPP [29], etc.), time-critical I/O controllers (video sensor, etc.), and dedicated or accelerated hardware components (IP). These dedicated units can take part in critical processes, for which no programmable or reconfigurable solution with sufficient computing performances exists. All processing elements (PEs) must be able to execute a task without additional external control, support the preemption of tasks, and access available memories. Finally, each task is executed by a predefined processor.

These processing elements communicate through local memories and a multi-bus network. This network connects all PEs and I/O controllers to all memory resources. Each PE owns two Level-1 instruction and data memories, which are not shared with other PEs or I/Os. The write-through cache policy is used to allow fast preemptions. Scratchpad memories are supported and must be preferred to reduce the energy consumption, but its content must be saved during a preemption. Preemptions and executions can be concurrent without execution conflicts. For better performances, the interconnection network supports simultaneous accesses without writing or reading conflicts; and each PE can access all distributed local memories. Moreover, all PEs can communicate with memories and I/O resources without undetermined latency, even if the consumer is absent or not ready. Data exchanges are non-blocking and deterministic, regardless of network load or execution constraint.

The following sections detail both OSoC architecture and SCMP computing mechanisms.

4. THE OSOC: A HW-RTOS

Accelerated components for RTOSs can be divided into two main categories. The first category provides dynamic, static or configurable scheduling mechanisms, supports external interrupts for aperiodic tasks and can eventually manage dependencies through a set of semaphores. Mooney et al., for example, have proposed the CHS architecture [30] to reduce scheduling time for highly reactive systems. Its scheduler provides three scheduling methods: priority-based, rate monotonic (RM) and earliest-deadline-first (EDF). This hardware implementation eliminates scheduling and time-tick processing overheads in RTOS. Nakano et al. have designed a solution called Silicon OS, which associates the μITRON real-time kernel and a hardware platform called Silicon TRON [31]. The Silicon TRON platform implements the scheduler and a set of semaphores, flags and timers, whereas OS software supports overall system management. Software primitives maintain the applicative coherency and the programmability of the system.

The second category of RTOSs additionally provides multiprocessor management. This is the case of the Mälardalen University's real-time unit (RTU) [32] and the real-time processor operating system (RTPOS) proposed by Isaacson et al. [33]. RTU architecture can drive three heterogeneous or homogeneous resources through a VME bus protocol. It provides static scheduling of 64 tasks on eight levels of priority, and manages external interrupts, resources and synchronizations through flags and timers. Like other solutions, it has decreased overall system overhead, resulting in improved predictability and response time. Both these types of architecture afford significant improvements. However, they do not meet our specific needs [31, 33]. That is why we have designed the OSoC.

The RTOS architectures just described are limited in terms of possible number of tasks. In contrast, the OSoC can create tasks dynamically, according to execution mode and environment, without user intervention or preprogramming functions. At runtime, it can adapt its resources to application needs. An OSoC can execute 256 tasks per application, 16 concurrent applications simultaneously and 16 instances of the same application. Furthermore, its architecture owns only 32 task resources. Whereas the number of processed applications does not impact performances, the number of available resources modifies both scheduling time and the chip area. With 32 task resources, it is possible to simultaneously activate 32 tasks. Each task resource is updating at run-time and will represent different tasks, and very complex application can be executed as long as it never runs over a 32-task parallelism. This degree of parallelism is sufficient to control a 32-resource platform. Existing multiprocessor HW-RTOS solutions use priority-level scheduling policies. Even if they allow priority change during execution, they are unsuited to our execution model. The performance of a CMP architecture depends on the number of available tasks. If only one level of priority is associated with each task, the number of available tasks is reduced by the resulting limited levels of priority. It is then impossible to exploit the parallelism of the architecture. Dynamic priority scheduling without priority levels is thus necessary to manage CMP architectures.

As shown in Figure 8, the OSoC architecture is made up of a control unit in charge of communications with the CPU and the external OSoC memory, and of functional blocks dedicated to application scheduling. The first block, known as the Task Synchronization and Management Unit (TSMU), is carried out by a modified version of RAC architecture [34, 35]. As already seen in the discussion of our programming model, applications are divided into a set of independent precompiled tasks and a set of Petri Nets that represent data and control dependencies for each application. The RAC is a self-adapting architecture, which can exploit its resources dynamically to construct Petri Nets. This block can select all active tasks and prefetch all tasks that are going to be executed afterwards. This permits to ask for the execution of multiple tasks in parallel and to reduce access latency to memories. The explicit description of data and control dependencies ensure the sequential execution of tasks. Non-deterministic processes are supported through conditional execution without speculation.

With the OSoC, the configuration of resources is done before the execution of tasks. They can thus begin their execution without suffering from external memory latencies. In ad-

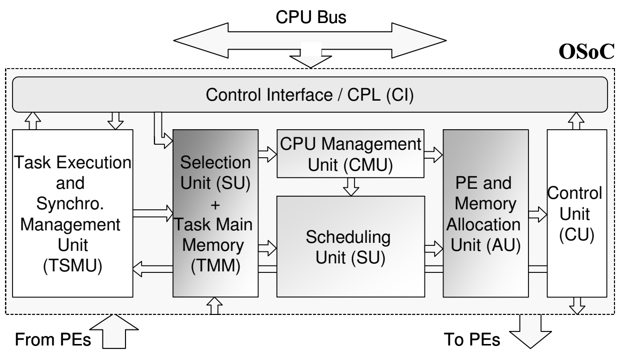

CPU Bus
OSoC
Control Interface / CPL (CI)
Task Execution and Synchro. Management Unit (TSMU)
Selection Unit (SU) + Task Main Memory (TMM)
CPU Management Unit (CMU)
Scheduling Unit (SU)
PE and Memory Allocation Unit (AU)
Control Unit (CU)
From PEs
To PEs

Figure 8: Detailed OSoC architecture: this component accelerates main RTOS functions. The Task Execution and Synchronization Unit (TSMU) manages control and data dependencies of tasks and directly implements Petri Nets. The Selection Unit (SU) synchronizes demands from the CPU, the TSMU and PEs. The CPU Management Unit (CMU) and the Scheduling Unit (SU) schedule tasks according to CPU demand, the set of ready tasks and PE information. Finally, the PE and Memory Allocation Unit (AU) assigns eligible tasks to resources through the Control Unit (CU).

dition, specific memory are used into the OSoC to enable multi-application execution. With the identity of the task, it is possible to retrieve its corresponding information into the memory, thanks to specific content addressable memory structures.

The OSoC integrates a modified version of the preemptive least-laxity-first algorithm proposed by J. Hildebrandt et al. [36]. The task whose time-to-end-of-execution is shortest is the one with the highest priority. Thus, the priority of each active task is evaluated at every clock-tick. Moreover, if two tasks have the same laxity, the task with the shortest deadline has a higher priority level, and the other task is excluded to prevent trashing. Non-real-time tasks are scheduled according to their execution time to optimize overall execution time and PE utilization [37]. Non-real-time tasks are executed only when a PE is available and when no more real-time tasks are waiting to be executed.

The OSoC can also schedule periodic real-time tasks or non-periodic tasks. Like any application, periodic tasks are specified in the form of task graphs. A task graph is a directed acyclic graph in which each node is associated with a task and each edge is associated with an event conditioned by the execution. For all tasks, the OSoC supports delayed execution and variable execution time as long as the deadline is respected. According to embedded application needs, each real-time application has a deadline, which represents the maximum time-to-end-of-execution. The deadline for each task of the application is then computed at runtime, according to the type of execution and environment. The deadline is a function of previous task deadlines and durations. Unlike all previous solutions, this permits non-deterministic execution.

In addition, the OSoC supports dynamic and parallel migration and preemption of tasks on multiple homogeneous resources. When a task is eligible, the allocation process selects a free computing resource identical to the expected target. If all resources are already used, the lowest priority task is preempted.

Finally, the OSoC manages system energy consumption by exploiting the DVFS and sleep modes of processing units and memories. Because we know the next task to be executed, we can locally optimize the consumption of our system. The optimization method used in the OSoC is based on making use of slack time [38]. This involves cumulating differences between worst case execution time (WCET), which is used to schedule tasks, and actual execution time. When the cumulative time is sufficient to activate a DVFS mode for the next task, the next resource is configured to the corresponding DVFS mode. In case of AND divergences, the slack time is allocated to the longest task. Sleep modes, on the other hand, are not used dynamically, but instead at the start and end of applications.

5. SCMP PROCESSING EXAMPLE

As shown in Figure 9, the SCMP architecture is made of multiple PEs and I/O controllers. This architecture is designed to provide real-time guarantees, while optimizing resource utilization and energy consumption. The next section describes execution of applications in a SCMP architecture.

When the OSoC receives an execution order of an application, its Petri Net representation is built into the Task Execution and Synchronization Management Unit (TSMU) of the OSoC. Then, the execution and configuration demands are sent to the Selection unit according to application status. They contain all identifiers of active tasks that can be executed and of coming active tasks that can be prefetched. Scheduling of all active tasks must then incorporate the tasks for the newly loaded application. If a non-configured task is ready and waiting for its execution, or a free resource is available, the PE and Memory Allocation Unit sends a configuration primitive to the Configuration Unit.

Based on the task identifier, the Memory Management and Configuration Unit (MCMU) allocates a memory space for the context, the code and the stack of the task. Then, its loads the instruction code related to that task, from the Main Instruction Memory; and initializes the context. Configuration of these local memories is sequential and takes place only once before execution of the task. Thus, no reading conflicts are possible when the Main Instruction Memory is accessed. Once the transfer is finished, the address of the selected memory along with the task identifier are written into the MCMU.

After the configuration of the task, if an execution demand has been received from the TSMU, the Scheduling Unit again updates priorities and the state of each task. If there is a free resource, or one task has a higher priority over another that uses the same type of resource, the OSoC sends an execution primitive to the selected PE. If a task is being executed by the selected PE, the OSoC demands its preemption. Then the task execution context is saved. Because all

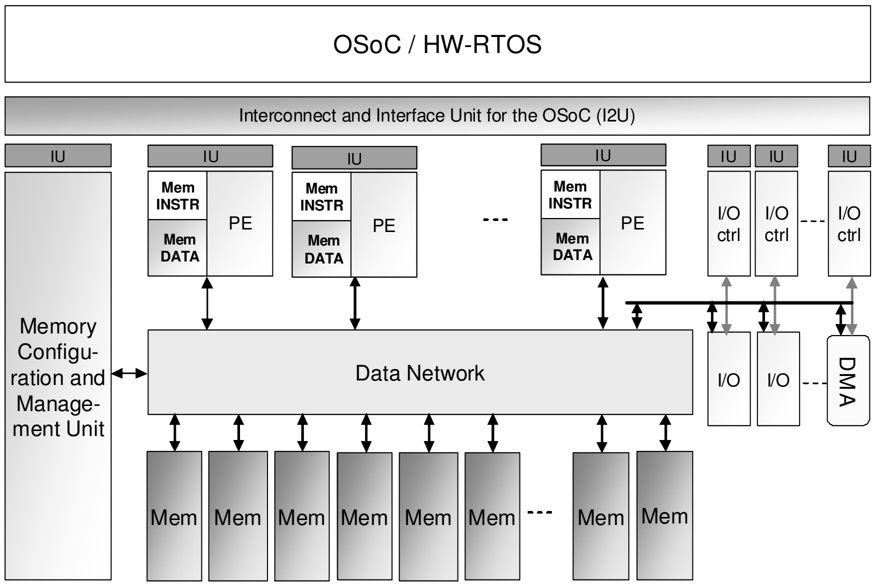

OSoC / HW-RTOS
Interconnect and Interface Unit for the OSoC (I2U)
IU
Mem INSTR
Mem DATA
PE
I/O ctrl
Memory Configu-ration and Manage-ment Unit
Data Network
I/O
DMA
Mem

Figure 9: SCMP structure: computing portion of CMP architecture.

memories are shared, this execution context can be accessed by another PE, enabling easy migration of task execution from one PE to another.

When a PE receives an execution request, it asks the MCMU for the translation address table of the task memory. This table contains all translations of allocated pages for the context, the code and the stack. With these addresses the PE can begin executing the task. The distribution of such data management units among the PEs allows concurrent communications and data transfers between tasks. I/O controllers are used by the OSoC as other PEs. For example, a data transfer from a DMA implies moving external data to the task memory. Local transfers can take place, where necessary, via another PE to distribute large amounts of data among other local memories, thereby improving access memory parallelism. Because all data required to execute the task are ready and all synchronization has been completed at the task selection level, the execution then simply consists of processing the data from local memories and storing the result in a memory open to the other tasks.

In the following section, we present the simulator of the SCMP architecture. This simulator is used to execute our application and to obtain the results presented in this article.

6. SCMP SIMULATOR

The SCMP simulator has been modelized within the *SESAM* (Simulation Environment for Scalable Asymmetric Multiprocessors) framework [39]. SESAM is a tool that has been specifically designed to demonstrate the feasibility of our architecture and its performance. This tool can also be used for the architecture exploration and optimization.

This framework is described with the SystemC description language and uses the ArchC tool [40] to generate Instruction Set Simulators (ISS) at the functional level. All the blocks of the simulator are timed, according to synthesized results, and communications use an approximate-timed Transactional Level Modeling (TLM) protocol [41]. This model brings hardware behavior to the simulator and exhibits, for instance, communication bottlenecks that are essential to correctly size the architecture. A study in [41] depicts a 90 % accuracy compared to a fully cycle accurate simulator.

We use ArchC MIPS32 processors as processing resources with data and instruction cache memories. To execute an application on the simulator, we use a terminal to send execution commands to the CPU. They boot on a read-only memory that contains all system code dubbed *system memory*. When the initialization is done, they wait for OSoC requests. We can dynamically execute tasks on processors. All communications are done at the transactional level and we can accurately estimate the time spent in every communication. Preemption and migration of tasks are done through an interruption mechanism that switch the context of the processing unit, saving and loading context code from the *system memory*. Because we use cache memories and write through policies, the preemption mechanism needs to save only file registers of the processor. With scratchpad memories, its content should have been saved. Besides, we use a fifo with each memory to store memory accesses from computing units. The arbiters use a fair round-robin policy. All features of the architecture can be modified after compilation to allow its sizing according to a specific application or customer needs.

All parts of the architecture have been described at the Register Transfer Level (RTL) in VHDL, except processors and cache memories. All latencies and constraints, characterized by the Synopsys Design Compiler tool with a low-leakage 0.13 μm@1.2V technology have been added into the behavioral SystemC model. The time between two successive schedulings, named time-tick, is about 19 μs with an

OSoC that can support 32 simultaneous active tasks and 8 processors. All networks have a latency of 2.5 ns and each memory has an access time of 2.5 ns. Each processor has a 1KB data and instruction cache memory and has an access to 64 memory banks of 16KB each.

We have deliberately used a homogeneous architecture to highlight our parallelism management approach rather than overall system performance. Nonetheless, we use a DMA unit to carry out input image transfers between internal local memories and the external data memory.

SCMP architecture offers a very high degree of parallelism. Thanks to dedicated hardware scheduling and fast reactivity, it also enhances resource utilization. To measure SCMP performance for embedded applications, several applications have already been implemented with it. In the following section, we discuss implementation of the connected component labeling algorithm, which is a critical application for embedded vision systems and is particularly relevant to this study in terms of dynamism, parallelism and control dependencies.

7. RESULTS

As an example, we decided to implement a labeling algorithm on our architecture. One of the most fundamental operation in pattern recognition is the labeling of connected components in a binary image. The labeling algorithm transforms a binary image into a symbolic image in order that each connected component is uniquely labeled based on a given heuristic. Connected component labeling is used in computer vision to detect unconnected regions in binary images. Various algorithms have been proposed ([42, 43]) but we have chosen a contour tracing technique that is interesting to stress the architecture [6]. This very fast technique labels an image in only one single pass over the image. It can detect external and internal contours, and also identify and label interior area for each component.

First of all, the image is scanned from top to bottom and from left to right per each line. When we detect an external or an internal contour point for the first time, we make a complete trace of the contour until we return to that point. We assign a label to this point and to all of that countour. When a labeled internal or external contour is encountered, we follow the scan line to find all subsequent black pixel and assign them to the same label. Because this application is very dynamic and its computation time depends on the size and the number of components, it is a good candidate to demonstrate the benefits of our solution.

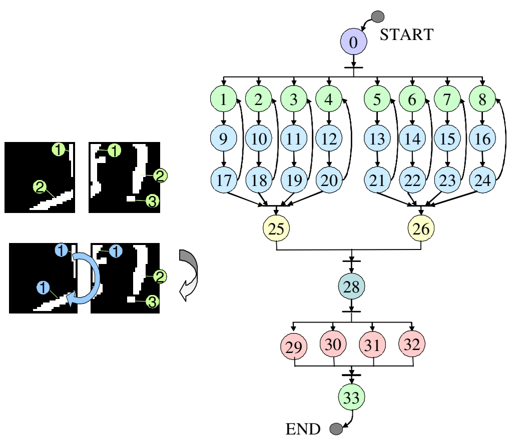
We simulated the platform parameters that affect our results and the OSoC penalties (e.g. communication, control, access time to memories, etc.). The whole architecture described in this paper is considered in our evaluation. We also simulated the connected component labeling algorithm on the SCMP architecture. We parallelized the initial algorithm by creating independent tasks and carried out the corresponding application graph. According to Figure 10, we cut the image into sub-images and applied the algorithm on each sub-image. Then, we carried out successively a vertical and a horizontal fusion of labels in analyzing frontiers between sub-images. We constructed corresponding tables between labels and changed in parallel all labels in sub-images. To get multiple independent tasks, we executed the application on a 512x512 image, cut into 128 sub-images. At the end, the parallelization brought new software complexity but involves independent and parallel tasks without modifying the algorithm. In this example, the application is restricted to a 8-task parallelism.

As shown in Figure 11-a, thanks to the multi-bank memory architecture, the overhead due to data accesses (waiting for ready data) or the OSoC is very low and represents only 2.5 % with 1 PE to 7 % with 8 PEs. This leads to an acceleration that can reach around 7 on 8 PEs (Figure 11-b). Unlike a software OS, the OSoC does not increase the overhead as the number of executed tasks rises [44]. Our fast migration mechanisms ensure a good occupation of our multiple resources. They increase the global performance and the flexibility of our multiprocessor. This capitalizes on free resources after the preemption of tasks. The migration just consists in saving execution contexts in a shared memory space, to allow any computing resource to continue the execution. No additional mechanisms are required except an efficient shared memory management. This is also possible because data dependencies are managed by the OSoC. Figure 11-c also confirms its good resource utilization rate (more than 80 %) even with 8 processors.

From both Figures 11-d and 11-e, it can be ascertained that a dynamic task scheduling brings a large benefit with dynamic applications. We considered two kinds of images: an asymmetric image and an image extracted from a real video sequence. With the static allocation strategy, we supposed that each column of the image is executed by a different processor. For instance, the left column of the image is executed by the first PE. Because the processing length of each labeling task depends on the pattern size and number, the execution time on each processor is variable. With an asymmetric image, the content of some sub-images are empty and only one processor is loaded whereas the others are rapidely free. In the example presented in Figure 11-e, the execution time increase with a static allocation strategy is about 75 %. With a dynamic allocation strategy, a load balancing is possible and all available resources can be used. In average, benefits are still important and reach more than 30 %. In a real video sequence, the content of sub-images is very heterogeneous and a dynamic approach can improve performances and therefore the energy and the transistor efficiency.

8. CONCLUSION

This paper presented a CMP architecture that supports task migration and preemption. The new architecture, which has been called SCMP, consists of a hardware real-time operating system accelerator (HW-RTOS), and multiple computing, memory, and input/output resources. The HW-RTOS, itself called OSoC, can manage all of the architecture's distributed resources and dispatches tasks dynamically under real-time constraints. This architecture has been designed to answer dynamic application needs. It supports very fast preemption and migration mechanisms to ensure dynamic load balancing depending on processor availabilities. A complete timed simulator, based on SESAM, has been carried out to demonstrate all implemented and new mechanismsn presented in this paper. This simulator environment allows the implementation of real applications, while keeping a high



Tasks	Function
0	Initializations
1 to 8	DMA transfers to internal shared memories
9 to 16	Labelling on subimages
17 to 24	Synchronizations
25 to 26	Horinzontal correspondence and fusion
28	Data synchronization
29 to 32	Vertical correspondence and fusion
33	DMA transfer to external memory

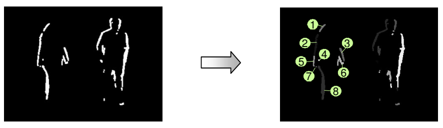


Figure 10: Labeling application after its parallelization. The labeling is done in all sub-images and two horizontal and vertical fusions are done to get the final labelized image. The corresponding graph is executed by the OSoC, whereas all tasks are processed by MIPS32 ISS.

accuracy level (90 %). Implementation of a connected component labeling algorithm with this system has confirmed the efficiency of our multiprocessor architecture, in terms of both processing element utilization and performance. Its resource utilization reaches more than 80 % on 8 MIPS32s. The overhead due to control and execution management is limited by our highly efficient task and data sharing management scheme, despite of using a centralized control that would have been induced important overhead to multiple requests to a same remote device. In addition, even if it was not presented in this paper, a complete hardware prototype of our architecture is currently running on a FPGA platform. This contributes to validate the architecture in order to design it on silicon to integrate future embedded systems. For the moment, we do not have a complete development flow, thus parallelization requires manual efforts. Future works will focus on the development of tools to ease the programmation of the SCMP architecture.

9. REFERENCES

[1] D.W. Wall. Limits of instruction-level parallelism. In *Int'l Conf. on Architectural Support for Programming Languages and OperatingSystems (ASPLOS)*, Santa Clara, USA, April 1991.
[2] M. Butler, T.-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow. Single instruction stream parallelism is greater than two. In *18th Int'l Symp. on Computer Architecture (ISCA)*, Toronto, Canada, May 1991.
[3] A. A. Jerraya and W. Wolf. *Multiprocessor Systems-on-Chips*. Elsevier, 2005.
[4] R.A. Iannuci, G.R. Gao, R. Halstead, and B. Smith. *Multithreaded Computer Architecture: A Summary of the State of the Art*. Kluwer Academic Publishers, 1994.
[5] T. Ungerer, B. Robic, and J. Silc. Mutithreaded Processors. *The Computer Journal*, 45(3):320–348, 2002.
[6] C.J. Chen F. Chang and C.J. Lu. A Linear-Time Component-Labeling Algorithm Using Contour Tracing Technique. *Computer Vision and Image Understanding*, 93(2):206–220, 2004.
[7] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *ISCA-22*, Santa Margherita Ligure, Italy, June 1995.
[8] M. Tremblay. The El'brus3 and MARS-M: recent advances in Russian high-performance computing. *J. Supercomp.*, 6:5–48, 1992.
[9] J. Emer. Simultaneous multithreading: multiplying Alpha's performance. In *Microprocessor Forum*, San Jose, USA, 1999.
[10] R. Kalla, B. Sinharoy, and J. Tendler. IBM Power5 chip: a dualcore multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.
[11] D. Koufaty and D.T. Marr. Hyperthreading Technology in the NetBurst Microarchitecture. *IEEE Micro*, 23(2):56–65, 2003.
[12] L. Hammond, B. Nayfeh, and K. Olukotun. A Single-Chip Multiprocessor. *IEEE Computer*, 30(9):79–89, 1997.
[13] Texas Instrument. High-Performance OMAP Platform: OMAP3430. http://www.ti.com, 2008.
[14] S. Dutta, R. Jensen, and A. Rieckmann. Viper: A Multiprocessor SOC for Advanced Set-Top Box and Digital TV Systems. *IEEE Design and Test of Computers*, 18(5):21–31, 2001.
[15] M. Paganini. Nomadik: A Mobile Multimedia Application Processor Platform. In *IEEE Asia and South Pacific Design Automation Conference (ASP-DAC)*, Yokohama, Japan, 2007.
[16] Cradle Technology. The Cradle 3616. http://www.cradle.com, 2008.
[17] D. Boerstler et al. D.C. Pham, T. Aipperspach. Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor. *IEEE Journal of Solid-State Circuits*, 41(1):179–196, 2006.
[18] ARM11 MPCore Processor: Technical Reference Manual. Technical report, ARM, 2005.
[19] L. Spracklen and S. Abraham. Chip Multithreading:

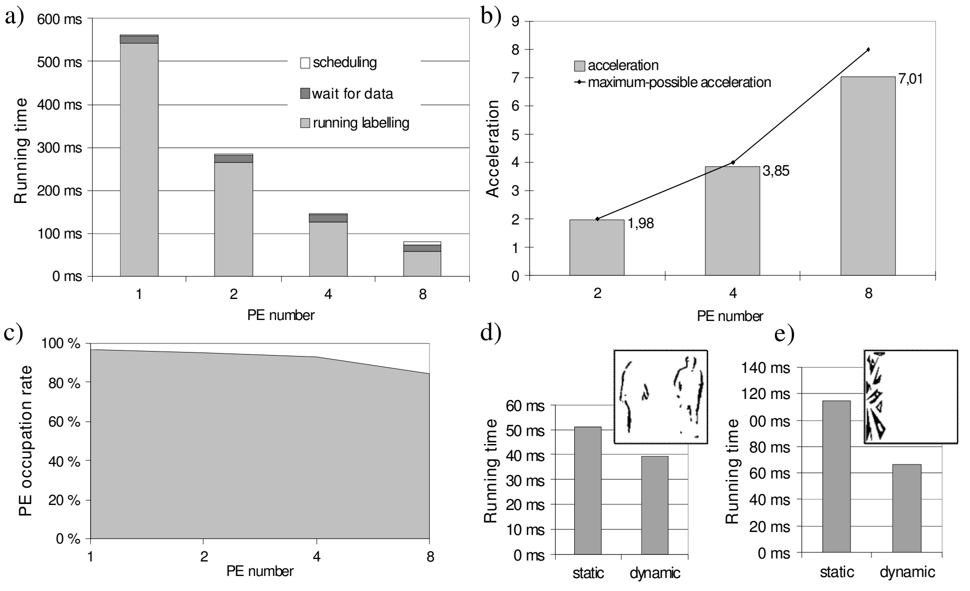


Figure 11: Connected component labeling algorithm execution in an SCMP architecture: (a) total execution time and overhead details depending on the Processing Element (PE) number (MIPS32 processors); (b) acceleration rate with multiple PE versus only one PE, and comparison with the maximum possible acceleration that we could obtain without overheads; (c) utilization rate of PEs; (d) total execution time with a static and a dynamic allocation strategy (realistic image); and (e) total execution time with a static and a dynamic allocation strategy (asymmetric image).

Opportunities and Challenges. In *11th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, San Francisco, USA, February 2005.

[20] UltraSPARC IV Processor Achitecture Overview, http://www.sun.com.

[21] M. Tremblay and S. Chaudhry. A Third-Generation 65nm 16-core 32-Thread Plus 32-Scout-Thread CMT SPARC Processor. In *IEEE International Solid-State Circuits Conference (ISSCC)*, San Francisco, USA, 2008.

[22] R. Sasanka, S.V. Adve, Y.-K. Chen, and E. Debes. Comparing the Energy Efficiency of CMP and SMT Architectures for MultimediaWorkloads. Technical Report UIUCDCS-R-2003-2325, University of Illinois at Urbana-Champaign, March 2003.

[23] J. Lee, V.J. Mooney III, A. Daleby, K. Ingström, and T. Klevin andL. Lindh. A Comparison of the RTU Hardware RTOS with a Hardware/Software RTOS. In *Asia and South Pacific Design Automation Conference (ASPDAC)*, Yokohama, Japan, January 2003.

[24] T. Samuelsson, M. Akerholm, P. Nygren, J. Starner, and L. Lindh. A Comparison of Multiprocessor Real-Time Operating Systems Implementedin Hardware and Software. In *Int'l Workshop on Advanced Real-Time Operating System Services (ARTOSS)*, Porto, Portugal, July 2003.

[25] M. Sindhwani, T.F. Oliver, D.L. Maskell, and T. Srikanthan. RTOS Acceleration Techniques - Review and Challenges. In *6th Real-Time Linux Workshop*, Singapore, November 2004.

[26] W. Kim, D. Shin, H.-S. Yun, J. Kim, and S.-L. Min. Performance Comparison of Dynamic Voltage Scaling Algorithms for Hard Real-TimeSystems. In *8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Halmstad University, Sweden, January 2002.

[27] B. Mei, S. Vernalde, D. Verkest, and R. Lauwereins. Design Methodology for a Tightly Coupled VLIW/Reconfigurable Matrix Architecture: A Case Study. In *Design, Automation and Test in Europe (DATE)*, Paris, France, February 2004.

[28] R. David, S. Pillement, and O. Sentieys. *Low Power Electronics Design*, volume 1 of *Computer Engineering*, chapter Energy-Efficient Reconfigurable Processors. CRC Press, 2004.

[29] V. Baumgarte, G. Ehlers, F. May, A. Nückel, and M. Vorbach. PACT XPP - A Self-Reconfigurable Data Processing Architecture. *Supercomputing*, 26:167–184, 2003.

[30] P. Kuacharoen, M.A. Shalan, and V.J. Mooney III. A Configurable Hardware Scheduler for Real-Time Systems. In *Engineering of Reconfigurable Systems and Algorithms*, pages 95–101, Las Vegas, USA, June 2003.

[31] T. Nakano, A. Utama, M. Itabashi, A. Shiomi, and M. Imai. Hardware Implementation of a Real-time Operating System. In IEEE Computer Society Press, editor, *TRON Project International Symposium*, pages 34–42, Tokyo, Japan, November 1995.

[32] J. Adomat, J. Furunäs, L. Lindh, and J. Stärner. Real-Time Kernel in Hardware RTU: A step towards

deterministic and highperformance real-time systems. In *ECRTS*, L'Aquila, Italy, June 1996.

[33] S. Isaacson and D. Wilde. The Task-Resource Matrix: Control for a Distributed Reconfigurable Multi-ProcessorHardware RTOS. In *Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, USA, June 2004.

[34] N. Ventroux, S. Chevobbe, F. Blanc, and T. Collette. An Auto-Adaptative Reconfigurable Architecture for the Control. In LNCS 3189, editor, *9th Asia-Pacific Conf. on Advances in Computer Systems Architecture (ACSAC)*, pages 72–87, Beijing, China, September 2004.

[35] S. Chevobbe, R. David, F. Blanc, T. Collette, and O. Sentieys. Control unit for parallel embedded system. In *Reconfigurable Communication-centric SoCs*, Montpellier, France, July 2006.

[36] J. Hildebrandt, F. Golatowski, and D. Timmermann. Scheduling Coprocessor for Enhanced Least-Laxity-First Scheduling in HardReal-Time Systems. In *ECRTS*, York, England, June 1999.

[37] N. Ventroux, F. Blanc, and D. Lavenier. A Low Complex Scheduling Algorithm for Multi-Processor System-on-Chip. In *IASTED Int'l Conf. on Parallel and Distributed Computing and Networks (PDCN)*, Innsbrück, Austria, February 2005.

[38] D. Zhu, R. Melhem, and B. Childers. Scheduling with Dynamic Voltage/Speed Adjustment Using Slack Reclamationin Multi-Processor Real-Time Systems. In *22nd IEEE Real-Time Systems Symposium (RTSS)*, London, UK, December 2001.

[39] N. Ventroux, A. Guerre, T. Sassolas, L. Moutaoukil, C. Bechara, and R. David. SESAM: an MPSoC Simulation Environment for Dynamic Application Processing. In *IEEE International Conference on Embedded Software and Systems (ICESS)*, Bradford, UK, July 2010.

[40] M. Bartholomeu G. Araujo C. Araujo R. Azevedo, S. Rigo and E. Barros. The ArchC Architecture Description Language and Tools. *Parallel Programming*, 33(5):453–484, 2005.

[41] A. Guerre, N. Ventroux, R. David, and A. Merigot. Approximate-Timed Transactional Level Modeling for MPSoC Exploration: a Network-on-Chip Case Study. In *12th Euromicro Conference on Digital System Design*, Patras, Greece, August 2009.

[42] I. Horiba K. Suzuki and N. Sugie. Linear-time connected-component labeling based on sequential local operations. *Computer Vision and Image Understanding*, 89(1):1–23, 2003.

[43] Lionel Lacassagne and Bertrand Zavidovique. Light speed labeling: efficient connected component labeling on RISC architectures. *Journal of Real Time Image Processing*, December 2009.

[44] P. Kohout, B. Ganesh, and B. Jacob. Hardware Support for Real-Time Operating Systems. In *International Conference on Hardware/Software Codesign and System Synthesis(CODES-ISSS)*, Newport Beach, USA, October 2003.