Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2009, Article ID 450607, 22 pages
doi:10.1155/2009/450607

*Research Article*

# OveRSoC: A Framework for the Exploration of RTOS for RSoC Platforms

**Benoît Miramond,[1] Emmanuel Huck,[1] François Verdier,[1] Amine Benkhelifa,[1] Bertrand Granado,[1] Thomas Lefebvre,[1] Mehdi Aïchouch,[1] Jean Christophe Prevotet,[2] Yaset Oliva,[2] Daniel Chillet,[3] and Sébastien Pillement[3]**

[1] *ETIS, CNRS-UMR8051, ENSEA, Université de Cergy-Pontoise, 6 avenue du Ponceau, 95000 Cergy-Pontoise, France*
[2] *IETR INSA—UMR 6164 CNRS, CS 14315, 35043 Rennes, France*
[3] *CAIRN—IRISA/ENSSAT, 6 rue de kerampont, 22300 Lannion, France*

Correspondence should be addressed to Emmanuel Huck, emmanuel.huck@ensea.fr

Received 15 March 2009; Revised 19 October 2009; Accepted 20 December 2009

Recommended by Lionel Torres

This paper presents the OveRSoC project. The objective is to develop an exploration and validation methodology of embedded Real Time Operating Systems (RTOSs) for Reconfigurable System-on-Chip-based platforms. Here, we describe the overall methodology and the corresponding design environment. The method is based on abstract and modular SystemC models that allow to explore, simulate, and validate the distribution of OS services on this kind of platform. The experimental results show that our components accurately model the dynamic and deterministic behavior of both application and RTOS.

Copyright © 2009 Benoît Miramond et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

## 1. Introduction

Nowadays, algorithmic complexity tends to increase in many domains such as signal, image processing or control. In parallel, embedded applications require a significant power of calculation in order to satisfy real-time constraints. This leads to the design of hardware architectures composed of heterogeneous and optimized computation units operating in parallel. Hardware components in SoC (System on Chip) may exhibit programmable computation units, reconfigurable units, or even dedicated data-paths. In particular, reconfigurable units, denoted here as Dynamically Reconfigurable Accelerators (DRA), allow an architecture to adapt to various incoming tasks at runtime.

Due to their intrinsic complexities, such heterogeneous architectures need even more complex management and control. In this context, the utilization of an RTOS (Real Time Operating System) is more and more required to support services such as communications, memory management, task scheduling, task placement, and so forth. These services have to be fulfilled in real-time according to the application constraints. Moreover, such an operating system must provide a complete design framework independent of the technology and of the hardware architecture. As for standard computers, the RTOS must also provide an abstraction and a unified programming model of the heterogeneous platforms. This abstraction permits to drastically reduce the time to market by encouraging re-usability.

Embedded RTOS for SoCs are of great interest and are subject of several significant studies. In the context of reconfigurable architectures, a study in [1] has determined and classified the different services that operating systems should provide to handle reconfigurability. Today, two different approaches have emerged for the development and the integration of these dedicated services. The first consists in utilizing an existing standard RTOS (RTAI, RTLinux, VxWorks, etc.) and in adding functionalities dedicated to the management of the reconfigurable resources [2]. The second is to develop a specific RTOS from scratch by implementing the necessary functionalities devoted to the management of the reconfigurable resources [3, 4].

The design process of such complex and heterogeneous reconfigurable systems requires method, rigor and tools. The OveRSoC framework is developed to take into account

both the RTOS, and the platform to propose an efficient exploration of the design space. The OverSoC methodology is based on 4 important design concepts: exploration, separation of concerns, incremental refinement and re-usability.

Firstly, a number of design choices have to be done prior any implementation, especially when the platform itself is designed and tailored for a specific application. We advocate the use of a high level model of Reconfigurable SoCs (RSoC) in order to explore different critical design choices. Among these important choices we distinguish two exploration issues:

(i) the exploration of the application partitioning onto the processing resources (topic already addressed in the literature [5, 6]),

(ii) the exploration of the RTOS services distribution and their algorithms.

Each design strategy belonging to these exploration levels is manually made by the designer. But the proposed method helps the designer to easily and quickly build the executable specification of the corresponding systems. The underlying tools then bring performance evaluations in order to analyze and compare design strategies. The design choices corresponding to the second exploration issue (RTOS) are the architecture of the embedded RTOS (centralized or distributed, OS services organization, software, or hardware implementation, etc.), the services algorithms (scheduling policies, etc.), the interactions between OS service functions and underlying resources (reconfigurable, memories, interconnects) and the software programming model.

Secondly, once validated the candidate design solutions are incrementally refined toward lower levels of abstraction down to the final implementation. The OveRSoC methodology permits the separation of concerns during the modeling and refinement process. It also defines modeling rules that facilitate independence and re-usability between components. For each design concern specific and related refinement steps are proposed. The resulting methodology serves as a design map for the designer of RSoC platforms.

Finally, the method imposes a functional approach at each level of abstraction which allows the validation of the application functionality besides the performance evaluation.

As a consequence, in the rest of the paper the problem of OS design is presented as a platform management problem. This paper presents the OveRSoC methodology and the related framework that consists of a set of SystemC models. The associated graphical exploration environment is also presented.

The remainder of the paper is as follows. Related work is described in Section 2. Section 3 presents the OveRSoC methodology and the corresponding tool for RTOS design. The flexible SystemC abstract RTOS model which allows RTOS service distribution and customization is presented in Section 4. Section 5 describes the RSoC architecture modeling step. Section 6 provides experimental results while Section 7 brings out our conclusions and presents the perspectives of this work.

## 2. Related Work

One of the main issues in reconfigurable platforms consists in determining efficient control mechanisms that may have dynamical properties in the sense that they must take on-line decisions from unpredictable system properties [7]. Several studies such as [3, 8] aimed at identifying the properties of the RTOS that can take dynamic reconfiguration into account. Specific properties such as application partitioning and tasks placement are described and placed in the context of reconfigurable computing which is often based on a farm of reconfigurable circuits. In [8], authors present one of the first attempts to develop an OS dedicated to the management of reconfigurable resources.

For the particular SoC domain, the authors in [9] list important properties to stress the usefulness of an OS to manage heterogeneous and static resources.

Adding reconfigurable units in a chip brings up many other issues from a design point of view. Introducing an OS for the management of an RSoC is of high interest in the research community [10]. Indeed, the partial reconfiguration abilities of current architectures need to be fully exploited in order to improve performance, cost, power-efficiency and time-to-market. Even if classical software approaches can be used, the OS then needs to be adapted to this new computation paradigm. More precisely, specific services are requested to manage the specific properties and resources of the dynamically reconfigurable units.

Designing a complete RSoC including an RTOS is a very complex task and requires appropriate methodologies. In this section we firstly introduce constraints on a dedicated RTOS for RSoC, we then discuss a proposal of methodologies in order to design these circuits efficiently.

*2.1. Dedicated OS Services.* The required specific services for RSoC can be roughly decomposed in four categories:

*2.1.1. Spatiotemporal Scheduling.* The task scheduling service is obviously one of the most important features of a multitasking OS. Scheduling of hardware tasks on reconfigurable areas adds a spatial dimension to the classical temporal problem [11]. This is defined as the spatiotemporal scheduling problem. The mapping of hardware tasks onto the reconfigurable unit can follow two spatial schemes according to the technology [3]: 1D or 2D schemes. While the 1D technique is simple to support, its performance in terms of computation density is low. On the other side, the 2D placement technique ensures a more efficient utilization of the reconfigurable area, but the associated algorithms are more complex.

*2.1.2. Reconfiguration and Resource Usage Management.* The resource management is very close to the placement service which needs to know the global state of the system. The resource table needs to be extended to store specific information necessary to manage the reconfiguration [2]. We can cite for example the area information for each reconfigurable task, the task communication needs which must be ensured

when the task is placed on the reconfigurable resource, the form factor, and so forth. The area fragmentation problem also appears when managing reconfigurable resources [12, 13]. This problem can prevent the placement of tasks while there is enough area within the reconfigurable resource. In this case, the designer can decide to implement a defragmentation service into the OS to limit the task placement rejection.

The reconfiguration latency of DRA represents a major problem that must be taken into account. Several works can be found addressing temporal partitioning for reconfiguration latency minimization [14]. Moreover, configuration prefetching techniques are used to minimize the reconfiguration overhead [15]. A prefetch and replacement unit modifies the schedule and significantly reduces the latency even for highly dynamic tasks.

*2.1.3. Task Preemption and Migration.* hardware task migration is an interesting property that requires the implementation of the hardware task preemption service [16]. Efficient implementation of preemption and migration requires several additional OS services, such as online communications routing and spatial placement. To limit the scheduling overhead and the number of configuration phases, which can be very time consuming, some OS prevent the preemption of hardware tasks. Non preemptive operating systems are known to be more deterministic, but do not take full advantage of platform flexibility. The conditions allowing more complex hardware task preemptions are defined in [17]. In this article, the authors describe three types of requirements allowing to perform multitasking on FPGA. First, save and restore mechanisms of current state of all registers and internal memories are required. Second, the configuration manager must obviously support fast configuration and readback of the FPGA. It must also have complete control over all the clock signals in order to freeze execution during context switching. Finally, it requires an open bitstream format in order to readback the status information bits.

As an example, preemption of hardware tasks have been studied in [18]. The authors present prospective architectural extensions of SRAM-based FPGA devices allowing a very fast and efficient context save and restoration. The proposed architecture supports the hardware defragmentation.

*2.1.4. Flexible Communications.* This property deals with the inter-task communication property of an OS and impacts the routing service [19]. The communication functionality is an important part of the system to ensure the data exchanges between all tasks, whatever their type or localization. Considering the localization of the tasks, communications are classically divided into two different types:

(i) the global communications: this communication level enables data exchanges between the different available resources (e.g., DSP, processors, reconfigurable units, etc.).

(ii) the local communications: this level ensures the data routing between different tasks placed simultaneously into the same reconfigurable area.

The global communication structures have to support flexible throughput and guaranteed bandwidth. In this case, OS services must provide the capacities to manage these structures. The requirements of the local communications within the reconfigurable area are quite different. Tasks implemented within this area are dedicated to intensive computation and are generally constrained by real time execution. In this case, communications do not support any delay nor excessive latency.

*2.2. RSoC Dedicated Methodologies.* Several studies tend to abstract the reconfiguration management by working at a system-level model. This level enables the exploration of systems while software, hardware or reconfigurable parts are not completely defined. It also enables the validation of various configurations to find the most efficient solution.

In order to introduce the reconfiguration in Symbad [20] which is a system-level codesign platform for SoC, the refinement phase has been modified to handle static and reconfigurable modules [21]. Specific simulation parameters, such as the reconfiguration time, are taken into account. Associated tools enable the evaluation of the reconfigurable contribution to the system performances. In [22], the authors propose a methodology in order to implement an application in an RSoC. This methodology is based on a UML descriptive model of the software parts and on a SystemC description of the architecture. Currently, these works do not take the dynamicity of the reconfiguration into account.

The collaborators of the Adriatic project propose an original methodology that handles dynamic reconfiguration [23]. The reconfigurable block is composed of a controller that launches or stops reconfigurable tasks, and features an input router that dispatches data among active blocks. Adriatic then proposes high level estimation of performances. Different strategies and approaches of estimation, simulation and partitioning are implemented in the Perfecto [24] and ReChannel [25] frameworks. Unfortunately, none of these works considers the development of an OS in order to dynamically manage the RSoC.

New approaches tend to provide a high-level hardware design model while managing the hardware implementation efficiently. This goal is achieved by a multi-languages approach.

In [26], the authors develop a framework based on the RTL language HIDE for implementation purpose, and on Handel-C to describe hardware at a higher level of abstraction. At present, the proposed framework does not handle dynamically reconfigurable architectures.

The multi-languages strategy is also used in the European Project Andres [27] which addresses heterogeneous systems. It is built around the HetSC methodology for the specification of the software part and the OSSS+R SystemC library for the reconfigurable part. Andres also includes a part of analog mixed design by supporting the SystemC AMS.

A special case of RTOS generation is the definition of dedicated OS services for DRA. The work presented in [28] addresses this problem by proposing a RTOS/SoC codesign framework. The customized RTOS is automatically generated from existing OS basic blocks which are available in software and/or in hardware. The 4S project [29] provides a design flow to develop RSoC platforms including an OS. In this project, algorithms are implemented into tasks which are mapped onto reconfigurable or non reconfigurable modules. The proposed tool provides information about the performances of each task for a given mapping. In an exploration step, the OS manages the implementation of tasks within reconfigurable units and generates flexible communication mechanisms. At present, these projects do not include the OS definition as part of the design process.

As a conclusion, adding reconfigurability in a platform imposes the management of hardware tasks at run-time. These tasks have to be placed into a reconfigurable unit in a dynamic and flexible manner. To ensure this management, some OS services need to be adapted (synchronization, migration, etc.), but some other services are completely new and need to be developed from scratch (spatiotemporal scheduling, fragmentation management, etc.). In the literature, to the best of our knowledge, no work proposes a complete solution, neither on real platforms nor in simulation, for the DRA management. The main contribution of this paper is to propose a unified modeling environment where all the needed services can be specified, tested and validated when distributed onto an heterogeneous multiprocessor platform. In this paper we do not provide and describe new spatiotemporal algorithms nor defragmentation methods but an open platform for the exploration of these complex algorithms where existing and upcoming methods for DRA management may be evaluated and compared. The services and the underlying platform are part of the exploration process. This objective has been reached thanks to the following contributions:

(i) a design methodology adapted to the exploration of the RSoC specific services,

(ii) a tool implementing this methodology,

(iii) a set of generic simulation models of MPSoC (Multi Processors System-on-Chip) components,

(iv) a high-level model of a DRA,

(v) a top-down refinement process.

## 3. The OveRSoC Methodology

In this section, we describe the methodology which is developed in the OveRSoC project and the tool that implements it. Our main goal is to provide a simulation framework for hardware/software design exploration of an RSoC including a dedicated RTOS. The framework is based on four main concepts: a methodology based on several design and analysis steps, the automation of simulation code generation from a library of basic blocks, the separation of concerns and the capability to simulate heterogeneous abstraction levels during the modeling process.

*3.1. Platform Exploration Flow.* The global methodology focuses on the original concepts addressed by OveRSoC, that is, the exploration of a distributed control of dynamic reconfiguration. In this way the methodology aims to explore the appropriate OS services that will be necessary to manage the RSoC platform. It relies on an iterative approach based on the refinement concepts as depicted in Figure 1.

The input of the exploration flow consists in specifying both the application and the system constraints. The RSoC platform model requires parametrization. The application is described as a set of tasks implemented whether in hardware or software. Their communications and synchronizations are also described as a graph of connections and dependencies. These dependencies can represent either pure data streams or synchronization mechanisms. Since version 2.0, SystemC supports a very powerful generic model of computation [30] but at the present time we only consider Communicating Sequential Process, Data-flow, and Kahn Process Networks [31]. These models satisfy the set of properties of the digital and signal processing domains that we address in this work. As imposed by the methodology, the functional behavior of each task must be defined as a pure C specification whether the tasks are executed in software or in hardware. During the early modeling steps, we use a common specification for the software or hardware implementation of a task. But for all the tasks, information about the execution time, periodicity, deadline are taken into account and considered as implementation specific attributes. This type of information may be either first estimated and refined afterwards or directly obtained by other tools that are capable of delivering accurate timing in the case of reused software or hardware IPs.

The basic RSoC platform considered is composed by three main types of components: the OS that manages the entire structure, the Processing Elements (PEs): the processors and the DRA, and the Communication Elements (CEs) composed of a communication media and a memory hierarchy. The OS may be distributed on the PEs of the RSoC platform (at least one processor and one DRA). The framework provides a set of models stored into the system library for each type of component. The library can be extended by adding new models to take into account new architectures. All the components feature their own list of design attributes. These attributes are used to customize each block within the RSoC platform. For example, the scheduler algorithm of a specific instance constitutes an attribute for an operating system, the latency of a specific task corresponds to an attribute for the application, the numbers and types of available resources within the reconfigurable area constitute one of the attributes that describe the DRA.

Once the platform architecture is defined and customized, the central work for the designer is to specify the different services that are required by the operating system in order to manage the global platform. Some services are available in a service library, but it is also possible to create new ones by specifying their behavior.

The validation of the design is based on the notion of metrics. Metrics are component specific measurements that can be reported to the designer during the simulation. They

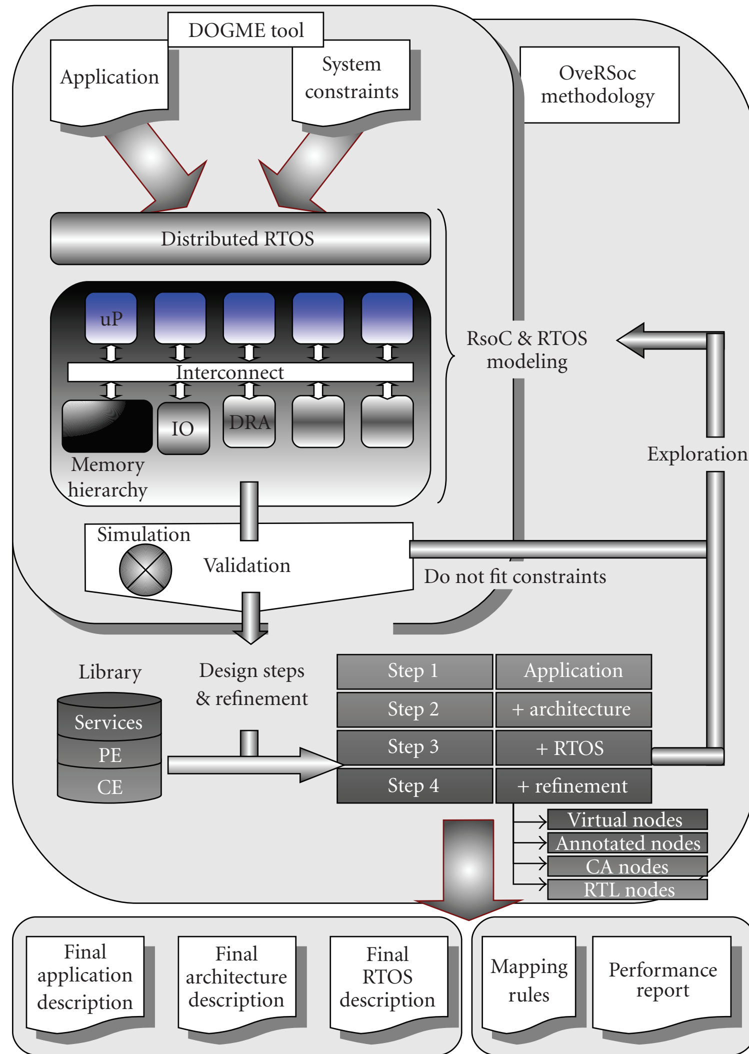



Figure 1: The OveRSoC exploration and refinement flow. Exploration is defined as an iterative process: modeling, simulation/validation and exploration. The inputs of the method are the specification of the application as a pure functional C code, and the system constraints. Once the system validated, the design process starts again at a lower level of abstraction until the final system description. At each level of abstraction, the goal of the exploration depends on the separation of concerns paradigm (Section 3.2). This paradigm is defined as a 4 steps process where the following concerns are successively addressed: application specification, architecture description, RTOS definition and platform refinement.

help the designer to verify the system constraints such as the PE workload, the communication congestion and so forth.

Examples of metrics that are already provided by the library components concern the tasks sequencing, the number of preemptions, the usage of resources and all events that may occur during the execution (semaphore's pend and post, etc.).

In particular, these metrics help to check the respect of the timing constraints. Obviously, the functional behavior of the application can still be validated by the designer. Once the attributes are completely defined, the whole platform is simulated in SystemC and metrics are evaluated. The analysis step is then manually performed by the designer in order to analyse the results of the simulation and to estimate the impact of specific attributes on the overall performances. The designer may then modify the value of some attributes and iterate the global simulation of the platform to explore the design space.

For the validation of the design choices, both the application (functionality) and the underlying RSoC platform (concurrency and timing) are simulated at high level in order to substantially decrease the simulation time of the whole platform. The exploration flow is conceived in a hierarchical way, according to the refinement concepts, and allows the designer to refine progressively his description of the platform to get more and more detailed results. We identified 4 refinement levels described in Section 3.3. At the highest level, we only consider the duration of tasks and RTOS calls, but not the memory nor the communication time. Then new attributes and metrics may appear as the description

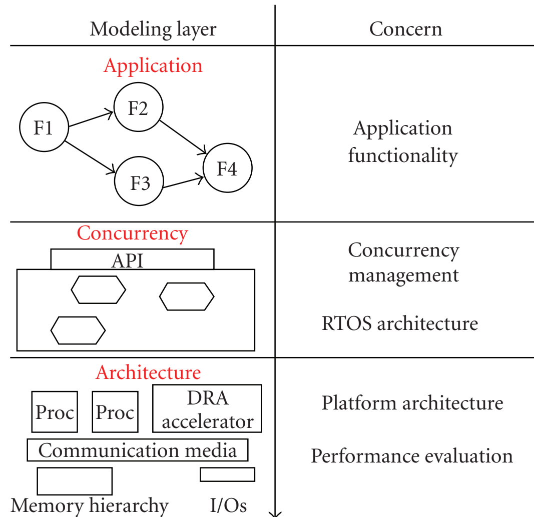



Figure 2: Our modeling approach follows the separation of concerns paradigm. The Application layer is a set of pure C functions and focuses on the functional specification of the algorithms. The Concurrency layer is a set of RTOS services and focuses on the distribution of these services. This layer also brings concurrency between threads according to the type of the associated PE. The Architecture layer is a set of parametrizable PEs, CEs and memories and represents the embedded platform. This layer also brings accurate timing evaluations.

becomes more accurate. For example, communications that are not taken into account in a coarse level of description may be accurately described to get more realistic values of the execution latency. New metrics like deadlocks on an interconnection network may also appear and provide the designer with new information about the global functioning of the platform.

*3.2. Separation of Concerns.* One of the main challenges of the proposed method is to keep the RTOS model as abstract as possible for exploration reasons while providing accuracy of performance estimation. The RTOS is maintained at a high level of description in order to easily add, remove, and deploy services without impacting the binary code of the cross-compiled application. The application is compiled once and the designer cannot only modify and refine the implementation of the RTOS services, but also scale the number of processors and DRA in the platform. As a result, the modeling space is separated into three independent layers depicted in Figure 2 according to the principle of the separation of concerns [32].

The top layer focuses on the functional specification of the application. This is described as a pure functional C code partitioned in C functions.

Then, some of these functions are associated with the notion of task in the following layer. Functional code calls RTOS services through a standard API (Application Programming Interface) as explained in Section 4. Communications between tasks depend on the synchronization services provided by the RTOS, for example, mutex, semaphores, fifos, mailboxes, and so forth.

In the next step, the OS layer deals with the concurrency between explicitly defined software processes. To reach this goal, we have developed a flexible SystemC model of a RTOS which is described in Section 4. Concurrent tasks are created thanks to specific services within the RTOS API. Multiple scheduling algorithms can be tested at this level according to the application constraints and possible task mapping to the underlying architecture without modification of the functional layer. In this layer, the designer can also explore the architecture of services into the distributed RTOS.

Finally, at the Architecture layer, the architecture of the embedded system is specified as a composition of heterogeneous processing elements (PEs) and communication elements (CEs). Each PE and CE may be modeled at different levels of abstraction and a refinement process can be performed without impacting the other modeling layers. Precisely, an ISS (Instruction Set Simulator) of a general-purpose processor executing a sequence of instructions is a refined model for an abstract function block. The independence of the hardware layer is ensured by a low level API, the Hardware Abstraction Layer (HAL) that always provides the same low-level services but with more or less accuracy as described in Section 3.3. This layer is also responsible for metrics' evaluation: execution time, processor utilization, memory usage, and so forth. Adopting such a modeling approach allows to reach the presented objective, that is, to explore the RTOS implementation at a high level while providing accurate performance evaluation of the entire system. According to the RTOS timer frequency, we observed on our application (see Section 6) that the execution time of the RTOS services represents ≈3 percent of the total application execution time. This observation corresponds to the results presented by Kohout et al. in [33]. Authors characterized the RTOS overhead according to the processing power used by the applications. The measured overhead grows from 2% to 9% for a preemptive RTOS and from 0.6% to 1.25% for a RTOS using a nonpreemptive strategy. But this is only for a monoprocessor system. In our case, when deploying an application on a MP-R-SoC, scheduling strategies and communication will completely change the system behavior and the waiting state durations. To deal with the OS overhead, we propose to keep the OS services at high level to ease exploration of its distribution or implementation. This observation is consistent with our approach that will provide accurate performance estimation on the Application layer which thus represents at least 90% of the total execution time.

*3.3. HAL Transactor and System Refinement.* Independence between modeling layers is ensured by a set of constant and standard services provided to the upside neighbor layer:

(i) independence between the *Application layer* and the *Concurrency layer* is ensured by the OS API,

(ii) independence between the *Concurrency layer* and the *Architecture layer* is ensured by the HAL API.

Table 1: Example of services provided by the OS and HAL API.

| Service component | OS API |
|---|---|
| Task management | void OScreateTask(code_pointer_t f, intu8 priority);<br>void OSdeleteTask(int task_id);<br>... |
| Semaphore management | sem_desc OScreateSem(sem_state init);<br>void OSreleaseSem(int sem_id);<br>... |
| Timer management | void OS_time_delayHMSM(<br>int h,int m, int s, int ms)<br>... |
| | ... |
| **Architecture component** | **HAL API** |
| PE | void compute(task_t* t);<br>save_context(task_t* t);<br>restore_context(task_t* t);<br>timer_set(int nbms);<br>timer_set_irq_handler(<br>code_irq_handler_t f);<br>timer_start();<br>timer_stop(); |
| CE | oversoc_t_rsp_t transport(<br>oversoc_t_req_t *REQ); |

The set of services provided by the OS depends on the chosen services. An example of service functions provided by the OS and HAL API is presented in Table 1. PEs and CEs provide to the OS components execution and transaction services similar to those presented in [34]. The call to the HAL services remains constant during all the refinement process but their implementation depends on the accuracy of the underlying layer. So both the OS and the HAL API allow to explore and refine lower layers while keeping higher layers unchanged.

Indeed PEs can represent abstract processing components when modeled at high level. They can also represent cycle-accurate processor, FPGA, or dedicated hardware models when described at lower levels. When the embedded application is partitioned and assigned to a PE, the PE mainly provides a *compute()* and *transport()* pseudo service to the RTOS, allowing a timed simulation for the computation and the communication. It also provides a service to trigger interrupts as components of the corresponding RTOS HAL.

The simulation accuracy then depends on the description of the internal architecture of the PE. We identify and advocate 4 refinement levels depicted in Figure 3.

(i) *Virtual nodes*: the PEs are used as empty boxes and the simulation is not timed. It corresponds to the Programmer View of the TLM approach [35], that is, a pure functional verification at high simulation speed.

(ii) *Annotated nodes*: the PEs are described as simple tables containing predicted execution times. The timings correspond to a back annotation of the execution time of each application basic block (Programmer View plus Time [35]) but without any modification of the application source code.

(iii) *Cycle accurate nodes*: at an intermediate level, software PEs are classically modeled as ISS (cycle-accurate) as explained in Section 5.2. In Section 5 we describe an equivalent model for the hardware PE (the DRA). From this refinement level, the HAL is implemented as a transactor, that is, a modeling artifact that translates transactional calls to RTL signals activations.

(iv) *RTL nodes*: at the lower abstraction level, a PE can still be described as an RTL model providing cycle-accurate timing evaluations and bit-accurate informations.

In a more general manner, thanks to the SystemC blocking calls mechanism, the Architecture layer interacts with the simulation core (SystemC) to advance the simulation time of the caller process according to the executed task. As for the synchronization and the preemption of the SystemC processes, it is ensured by the upper level which manages notification and waiting of SystemC events as described in [37]. In the case of MPSoC platforms, synchronization between processors is ensured by interruptions and by a hardware shared semaphore model. But whatever the chosen abstraction level of the Architecture layer, the Concurrency layer (i.e., the OS services) remains at the same abstraction level. This level is called SAT (Service Accurate plus Time) and is described in Section 4.

*3.4. The DOGME Tool.* Due to the complexity of the exploration process, the HW/SW designer needs tools to apply the OveRSoC methodology. The DOGME (Distributed Operating system Graphical Modeling Environment) software provides an integrated graphical environment to model, simulate, and validate the distribution of OS services on RSoC. The goal of the tool is to ease the use of the exploration methodology and to generate automatically a complete executable model of the RSoC platform (hardware and software). The automation is based on a flexible SystemC model of RTOS described in Section 4. This RTOS model is a package of modular services. To develop each service, an Object Oriented Approach has been adopted and implemented using the SystemC 2.2 library. This tool allows an application specific RTOS to be built by assembling generic and custom OS service basic blocks using a graphical editor [38]. The application is linked to the resulting OS thanks to a standard POSIX API. Finally, the entire platform is simulated using the SystemC kernel.

The developed tool follows five main design steps represented in Figure 4.

(i) *Design of the platform*: the design phase consists in choosing and instantiating toolbox components into the graphical workspace editor in order to assemble the OS services and distribute them onto the RSoC processing elements. Figure 5 shows an example of RTOS composition including services like task management, scheduling, semaphore, IRQ controller... At this step the designer will successively, and according to the separation of concerns paradigm, take decisions about

   (i) functions mapping into threads,
   (ii) hardware/software partitioning,
   (iii) instantiation of the required services,
   (iv) distribution of the services onto the PEs.

(ii) *SystemC source code generation*: after interconnecting all components and verifying the bindings between services, the structural source code of all the objects that are instantiated into the platform is automatically generated.

(iii) *Compilation and simulation of the platform*: to complete the design of the platform, the parametrized structural SystemC description is combined with the behavioral source code of the components provided by the user. The global SystemC description is compiled and simulated.

(iv) *Analysis of the simulation results*: graphical diagrams are produced to visualize the evolution of the system metrics during the simulated time. This step helps the designer to evaluate the current design quality. It acts as a decision guide for the exploration of the design solution space.

We are currently implementing the DOGME tool as a stand-alone application based on an Eclipse Rich Client Platform [39]. Typical project management functions like importation of platforms or components into the standard library are supported as well as the creation of new platform models. Reusability is achieved in the tool by the possibility to add the newly created platform to the standard library. All data manipulated by DOGME are loaded and stored using a proprietary XML format dedicated to embedded software modeling as depicted in Figure 4.

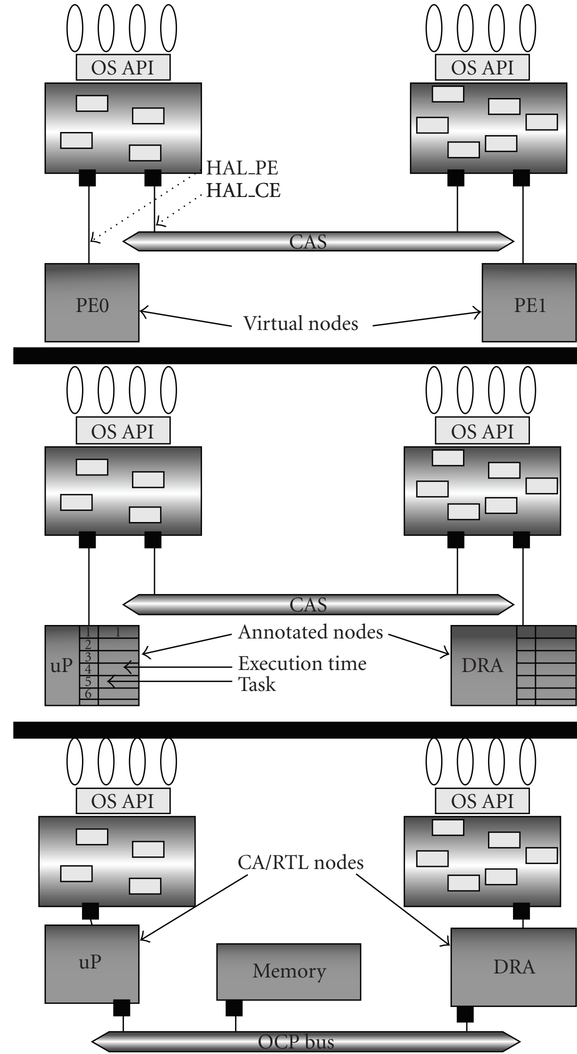



Figure 3: Example of refinement of the minimal RSoC platform. The first level begins after hw/sw partitioning of the application and corresponds to virtual nodes. The second level refines PEs to annotated nodes. At this level, each task has an estimated execution time. The CE is modeled as a transactional bus called CAS (Calling Abstraction Service). The two last levels correspond to Cycle Accurate or RTL nodes. The global CE has been refined to an OCP bus [36]. The memory accesses are now taken into account accurately, that is, all the communications can be evaluated at the lower level.

## 4. Distributed RTOS Model

This section presents the essential mechanisms needed to jointly model and simulate hardware/software tasks and the RTOS in SystemC.

*4.1. A RSoC Model Based on RTOS Services.* In order to model complex embedded platforms composed of multiple parallel and heterogeneous (and reconfigurable) resources, it is important to be able to jointly model the functional software, the underlying hardware and the glue between, which is generally composed of RTOS instances.

In the step 3 of the design process (see Figure 1), to explore the design solution space, we choose to model the system at a high level of abstraction, where the hardware is partially hidden. We focus our modeling process on the services provided by the platform.

At the Concurrency layer (see Figure 2), we address the SAT level of abstraction: Service Accurate plus Time. This allows us to very quickly simulate the behavior of the application, compared to lower detailed levels of abstraction. This level of concern is different from the Donlin's CP+T

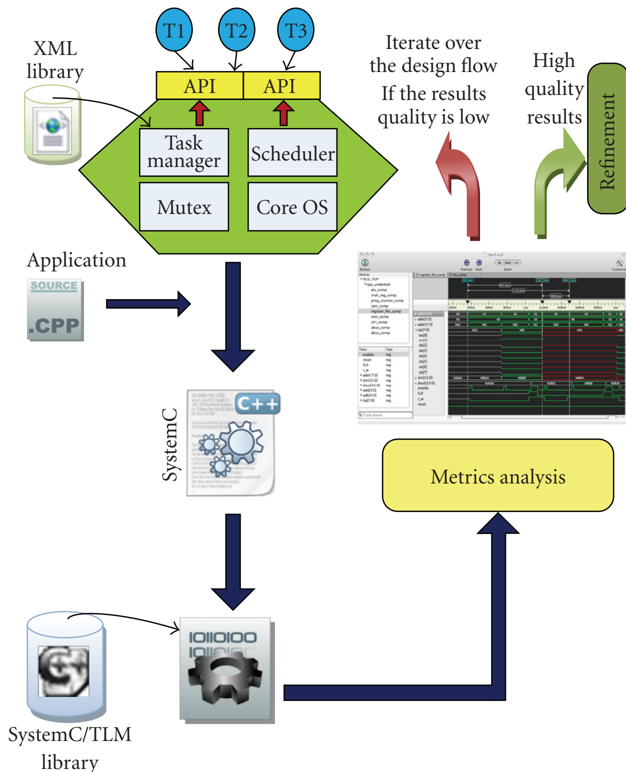



Figure 4: The DOGME tool brings facilities to manipulate the components of the library. These components model RTOS services for the control of an RSoC platform. In the library the services are described both by a SystemC generic source code and an XML exchange file. The designer graphically instantiates the components, then the tool automatically adds debug components for metric evaluation into the specification and generates the code of the corresponding platform. The platform is compiled and linked with the SystemC libraries and simulated thanks to graphical interfaces. The designer can finally evaluate the metrics of his platform and can take decisions about exploration or refinement.

(Communicating Processes with Time) level [35] which mainly focuses on hardware modeling but which does not include the RTOS services. This level of modeling implies that the architecture is not modeled explicitly, all the application tasks are functional, annotated with approximated or measured execution timing, and all the RTOS services are explicit and timed.

The core element of our distributed architecture model is a high-level functional model of a RTOS written in SystemC. Since SystemC does not support OS modeling facilities in its actual version, a first step was then to extend SystemC with embedded software modeling features [37]. The works presented in [40–43] are examples of simulation environments dealing with this challenge.

The proposed RTOS model [37] acts as a *Service Accurate + Time* model of a virtual PE (processor or DRA) in the sense that all the necessary services of an embedded RTOS are modeled as independent modules with their own behavior and timing. The RTOS model is built as a collection of *service modules* implemented in the form of a hierarchical `sc_modules` to foster high level exploration of custom architectures. The main RTOS model instantiates all its modules and uses `sc_export` to provide a global API to the application code as illustrated in Figure 6. Each *service module* has its own interface that furnishes the corresponding services' functions to the embedded application. This model includes mechanisms for modeling dynamic creation of tasks, task preemption and interrupt handling as described in [37]. Figure 6 illustrates the hierarchical structure of the SystemC RTOS model composed of the following service modules:

(i) a task manager that keeps the information and properties of each task according to its implementation (software or hardware): state, context, priority, timings, area, used software or hardware resources . . .

(ii) a scheduler that implements a specific algorithm: EDF [7], HPF [7], horizon [44], . . .

(iii) a synchronization service using semaphores.

(iv) a time management service that keeps track of time, timeouts, periods, deadlines . . .

(v) an interrupt manager that makes the system reactive to external or internal events.

(vi) a specific simulation service (advance time).

Each service module is modeled as a SystemC hierarchical `sc_channel` and is symbolized in the figure using the SystemC representation [30]. A service module thus provides several service functions through its interface.

For example, the task manager provides the following functions: create (dynamically) a task, delete a task, get the state of a task, change the state of a task. . . The task creation function associates a simulation process (and thus concurrency) to one of the pure C function present at the Application layer.

Some service functions are accessible from the Application layer through the OS API. Those are called external service functions. Others are only accessible from the other service modules through a SystemC port to establish intermodule communications and are called internal service functions.

At this layer, timed simulations of the application use a specific simulation call (called OS_WAIT()), associated to each bloc of task code between two system-calls and redirected through the Concurrency layer toward the Architecture layer. This service, represented in Figure 6, allows each function to progress in time. In addition, each OS service function within the OS itself may also be annotated with timing information (depending on the processor) allowing a timed simulation of a realistic system.

Actually the system library provides a set of basic generic services: interrupt management, timer management, intertasks synchronization, and memory management. It also provides hardware and software specific services such as the task management of software or hardware tasks, software scheduling policies and hardware placement algorithms.

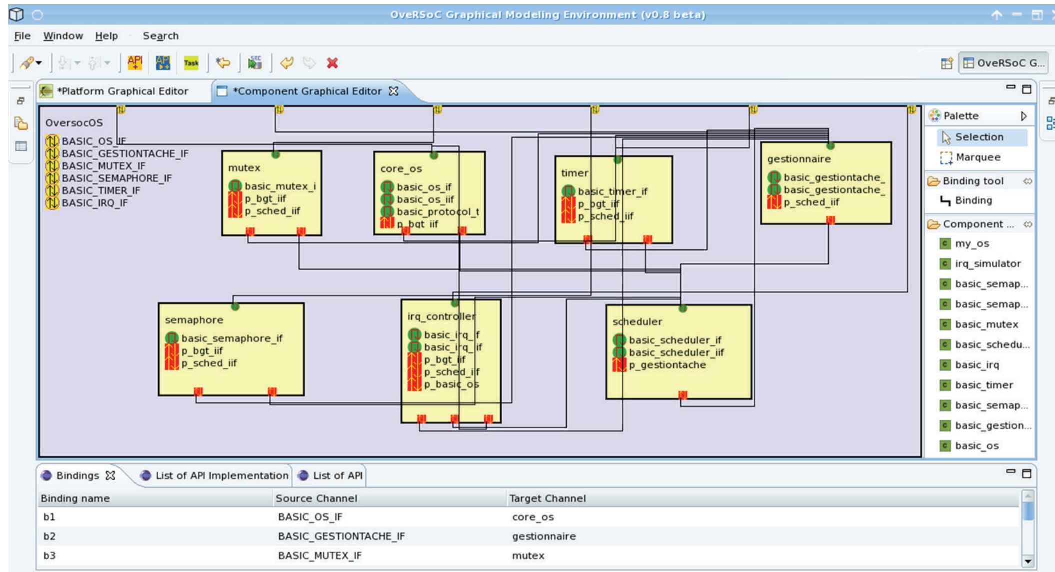



Figure 5: The DOGME tool represents a distributed RTOS through hierarchical views: the Component Graphical Editor, where the services are organized inside each PE, and the Platform Graphical Editor, where the groups of services are composed according to the number and type of PEs into the RSoC platform. Here the Component Graphical Editor is shown. It uses toolbox components to specify and customize the services of a dedicated group. Each service is modeled as a software (C++) component having ports and interfaces. Each service component provides several service functions.

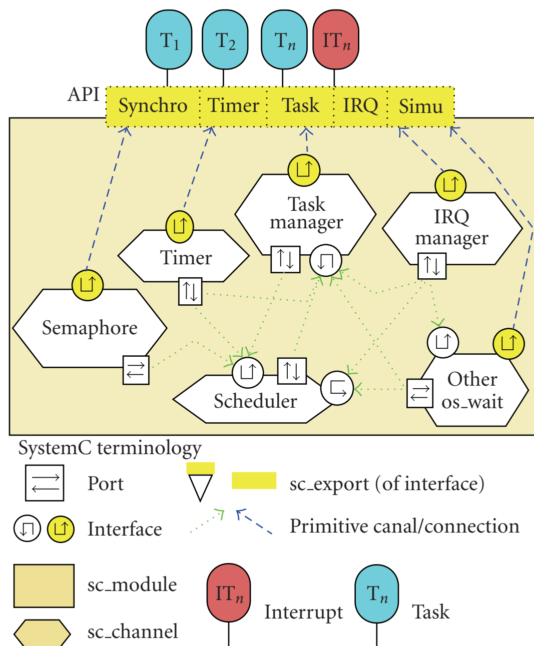



Figure 6: The modular RTOS model and its composed API. Each OS service exports its own interface to the application. Services are connected together to ensure the global OS coherency and behavior.

*4.2. Distant Communications and Services Requests.* We extend the model for distributed multiprocessor architectures exploration with the following features: the whole application is decomposed into multiple threads sharing the same addressable memory, the application is statically partitioned onto multiple processing nodes, each processor has its own scheduling strategy (policy, priorities, etc). All inter-processor communications are modeled using transactions with respect to TLM 1.0 methodology. A unique `transport` method is used for both requests and replies. All communications are currently performed instantaneously but this allows a communication refinement process and thus a time accurate simulation by introducing bus-related or network-related timings into transactional ports.

Our approach for modeling distributed OS services is inspired from the middleware philosophy which consists in using proxies and skeletons services. A proxy service provides a local entry point to a distant service accessible through an interconnection infrastructure. This adds dedicated ports and interfaces to the RTOS (and also on services modules needing to communicate).

Figure 7 illustrates transactions between two local semaphore services (proxies) and a shared distant semaphore implementation (skeleton). `Get` and `release` semaphore invocations are performed locally to the proxy which forwards transactions to the distant service. By using a simple `transport` method, all distant calls put the caller tasks into an active waiting state. In case of access conflicts, the shared service has its own arbitration policy. Then, replies are sent back to the caller at the end of the service execution.

Communication from a distant service to local proxies are performed by using signals which are similar to interrupt requests that are managed by local proxies. Suspended tasks may then be resumed by their own schedulers depending on local policies.

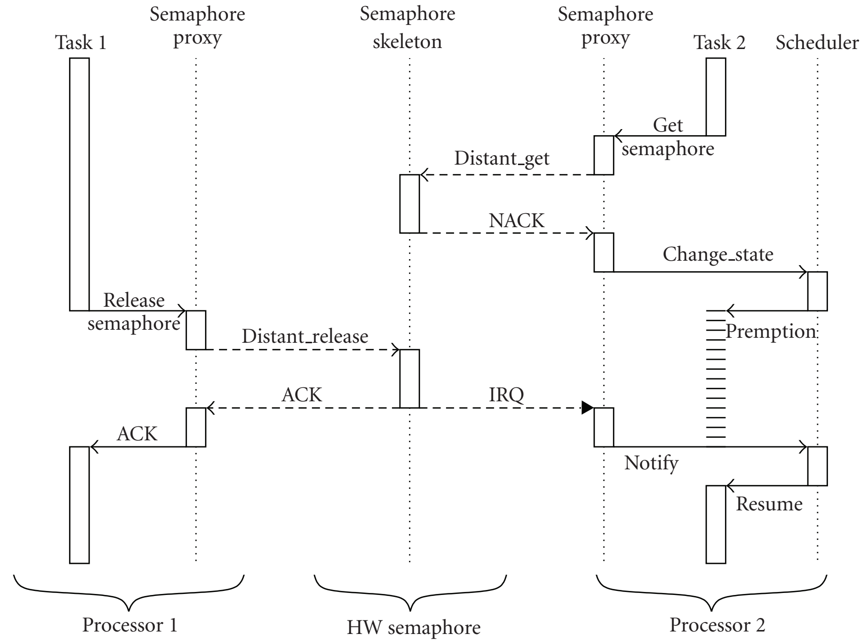



Figure 7: Activity diagram of local/distant calls to a shared semaphore proxy/skeleton between two OS models.

Based on this distant service invocation, we can easily imagine and construct a model of a shared distant synchronization service (potentially implemented in hardware), like a semaphore. Then it allows to quickly map the application onto a multiprocessor platform and evaluate the potential acceleration that distribution of computations could potentially allow, as shown on Figure 8.

Based on this mechanism, we can design a new RTOS with dedicated services for a DRA. We can then explore and evaluate their behavior, as shown in Figure 9, and try different scheduling policies specific to hardware IP placement on the DRA.

As illustrated in Figure 9, we propose a set of high-level models for the preceding specific services. We are able to create, schedule, preempt, and delete hardware tasks onto a distant DRA. All these tasks execute and communicate with the other local or distant tasks indifferently. At this first level, the specific properties of the hardware implementation remain abstract and the scheduler only considers the current free area to take scheduling decisions. At lower levels of abstraction, the services implementation directly depends on the properties provided by the DRA model in the hardware modeling layer as described in the next section.

## 5. Abstract Models of the Reconfigurable Platform

During the refinement steps of the methodology, we need to refine some elements of the design, as the Dynamically Reconfigurable Area, and the processors for software tasks. This implies to integrate more detailed elements as ISS for processors and also a detailed DRA model referred as a CSS (Configuration Set Simulator). These refined models allow to automatically annotate software and hardware tasks timing and to analyze more accurately their behavior during execution.

*5.1. Reconfigurable Architectures Modeling.* Reconfigurable modeling is a well known issue and has been addressed for example by Becker in [45] for 1D partial regions.

In the OveRSoC project, the DRA model is composed of both an *active* and a *reactive* component. *Active component* models the hardware physical architecture. It encapsulates the constraints of the physical circuits. It corresponds to the internal organization of the DRA and ensures the execution of hardware tasks. The *Reactive component* models the dynamic behavior of the architecture. It represents the API of the DRA which provides several OS services and attributes through a fixed logical interface. In the OveRSoC project this component constitutes the interface between the external OS model and the DRA model.

These two components represent the reconfigurable hardware unit and must support the exploration strategy and the refinement of all manipulated objects. To ensure the exploration process of OveRSoC and keep complexity under control, the DRA is defined through a multilevel model.

Both active and reactive components are tightly coupled and the refinement of each impacts the other. The exploration process of the *active* and *reactive* parts of the DRA is constrained by the level of description of each component.

Three levels of abstraction for each component are proposed (see Figure 10). The refinement process applied to the DRA consists in successively defining the three proposed levels and their properties.

In the model, the *level 1* corresponds to an annotated node (Section 3.3). The different components are modeled

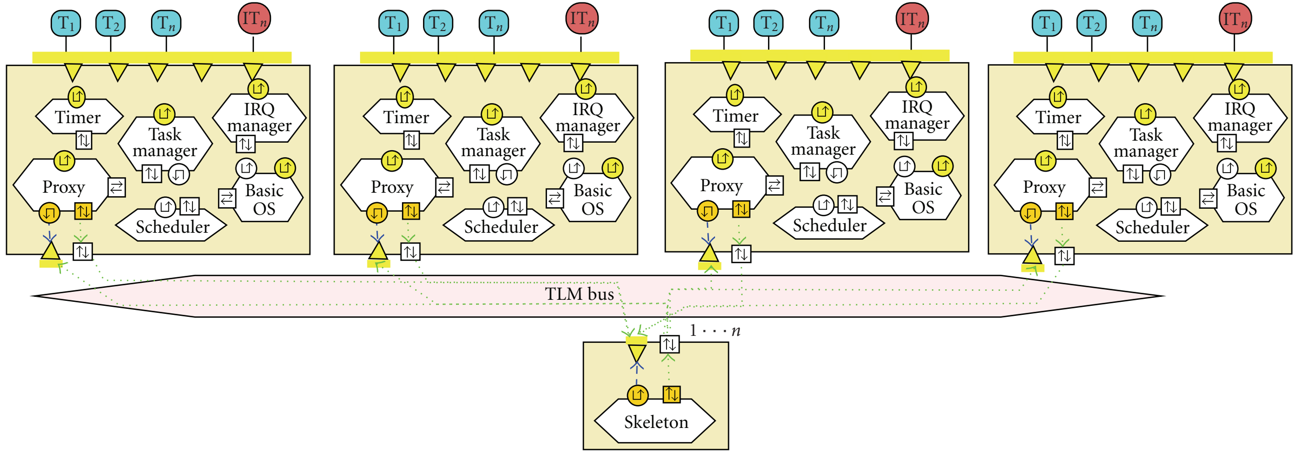



FIGURE 8: Model of MPSoC RTOSes with a hardware shared semaphore service. Each RTOS has a local *Proxy* service which forwards a (semaphore) request to an external device (the *skeleton*) that processes the real service, as a RPC (Remote Procedure Call), except the skeleton services could be refined in hardware.

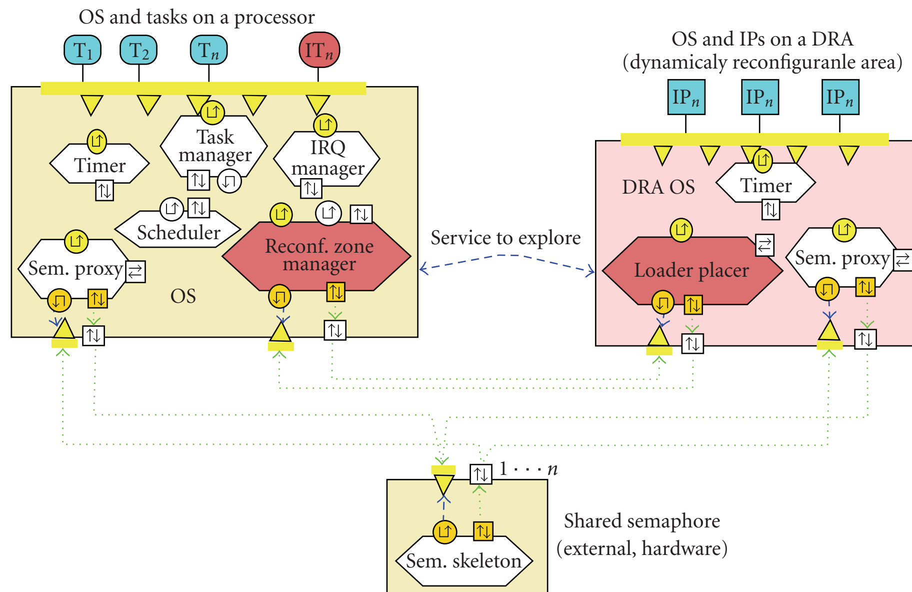



FIGURE 9: Model of RSoC specific OS: one standard customized with a DRA manager, one specific into the DRA, and another one specific for a refined shared semaphore service alone as an external device.

through a small number of parameters and permit a fast and coarse evaluation of methods and performances. The *active component* is considered as an homogeneous unconstrained rectangular area with a reconfiguration memory. The only parameter which is required to execute the tasks in the DRA is the task's area. At this level, the resources of the DRA are considered as unconstrained, that is, no bandwidth limitation, no latency, no area constraints, and so forth. In terms of performance, the designer evaluates the global area required in the active components, as well as the reconfiguration overhead introduced by its task management services.

The second level refines the *active components* defined as a rectangle which contains a set of heterogeneous resources such as memory, abstract running blocks and interconnect resources with limited bandwidth. The task heterogeneity is present at this level and a minimal placement service is required. At this level, the *reactive component* uses the structural information of the *active component* to verify the constraints of tasks. The corresponding definition of tasks must be completed by parameters, such as the rectangular size, the form factor of the area and so forth.

The *level 3* is the most accurate level of description and all the elementary blocks of the *active components* are described. They are defined as an array of LUT (Look Up Table) with glue logic for arithmetic computation and the corresponding sequential elements, a set of memory allocated throughout the array, columns of hardwired blocks and eventually

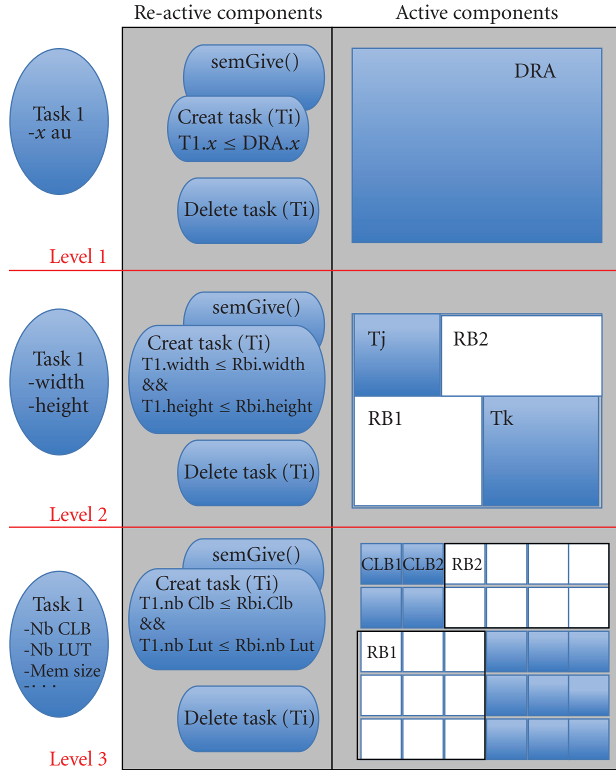



Figure 10: Hierarchical model of the active and reactive components of the DRA. The different levels permit to represents the DRA with more or less details. Refinement process leads to the complete definition of the internal architecture of the DRA. Belonging to the refinement of architectural aspects (*active* component), the supported services can be developed and evaluated (*reactive* component).

hardware core processors like PowerPCs in last Xilinx's technology. The corresponding *reactive component* must implement all the services described in Section 2. These services take both the application constraints and the precise circuit organization into account.

From this model, the DRA management can be explored through the implementation of distributed OS services.

For example, we present a particular implementation of the `createTask` OS service in Figure 11. In this example, a placer and a loader service are also implemented in the DRA. The first sequence of Figure 11 shows the hardware task creation call, `createTask(T3h)`. This OS call is performed by the software task `T1` and is handled by a processor OS service. Since the task to create must be executed onto the DRA, the OS service call is passed to the DRA through the interface, `createHWTask(T3h)`. This interface, implemented by the *reactive component*, calls the DRA OS service of task creation. Before loading the task, the DRA must verify if this new task can be loaded and placed in the reconfigurable area. To do that, the hardware OS calls the placer service, `isLoadable(T3h)`. At this step, the verification depends on the level of DRA description. For example, at level one, the placer checks if the available area is sufficient for this new task. In this case, we can model this verification as

$$\sum_{i=1}^{Nt} A_i + A_{newtask} \leq totalArea, \quad (1)$$

where $A_i$ is the necessary area for the task $i$, $A_{newtask}$ is the area of the new task to instantiate onto the DRA, $Nt$ represents the number of tasks already instantiated within the DRA, and $totalArea$ the total DRA area.

In the first part of this diagram, we illustrated the case where the placement of a new task is possible. In this case, the placer calls the loader service, `loadTask(T3h)`. The loader ensures the loading of the task bitstream, `loadBistream(T3h)`, and finally starts the task, `start(T3h)`. This sequence can be modified in order to evaluate potential overhead of different implementation solutions.

In the second part of this sequence the *CreateTask* OS call is performed by the software task T2, `createTask(T4h)`. The beginning of the sequence is the same as the first sequence presented above, but in this case, we consider that the placement of the new task into the DRA is not possible, ie. the return value for the `isLoadable(T4h)` function is No OK due to unavailable area. In this case, the task execution is refused by the DRA, and an error signal is returned. To finish this example, we suppose that a software version of task `T4h` exists and the system decides to switch to the software version, `create(T4s)`, and to schedule it immediately.

*5.2. Processor Modeling.* In this work, we use ISS for software simulation. As a proof of concept of our embedded software modeling approach, we developed a SystemC ISS corresponding to the ATMEL AVR Instruction Set Architecture. Targeting either hardcore processors or ISS follows the same compilation flow. We can thus reuse standard compilation tools. The binary code must then be loaded into SystemC memory models by external modules (bootstrap). The ISS communicates with memory through standard hierarchical channels. At this level of the model framework, communications can be refined towards Register Transfer Level. The ISS fetches instructions and simulates opcode execution. We implemented two modes of operation for the ISS: accurate and fast mode. When functioning in its main (accurate) mode the ISS classically extracts, executes 16-bits opcodes and increments the program counter. Once a basic block, has been executed, the ISS keeps track of the simulated execution time into specific tables to minimize the simulation overhead. Each basic block is thus associated with a block ID which corresponds either to an entire software task code or to instruction blocks within the task code. The ISS can also be interrupted and can thus model task preemption at a very fine level. In fast mode, preemption is also possible but at a coarser level since simulated time advances with a basic block precision. Interrupts can not occur before the end of the single SystemC `wait` time parameter. Once interrupted, the remaining time is saved in tables and reused when the basic block is started again.

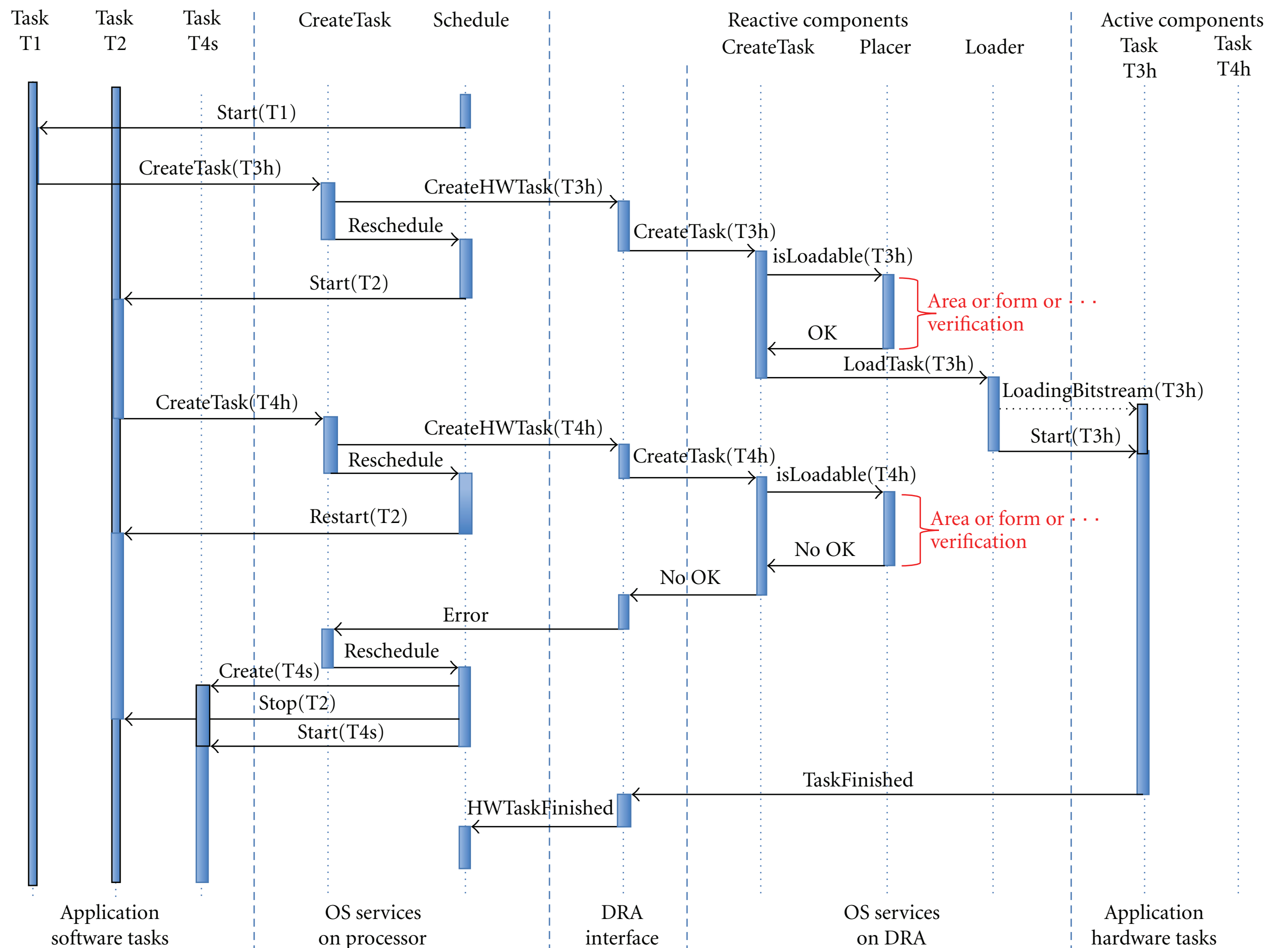



Figure 11: Sequence diagram of the *CreateTask* service implementation. After a Create task system call a sequence of system call depends on the services implemented on the DRA. Here we can evaluate and develop the loader and the placer services of the DRA dedicated RTOS.

Since components within each layer can be described at different levels of abstraction, the challenge is therefore to synchronize the functional and timed simulation across the layers. This is particularly difficult for the software models that exist at three different layers simultaneously: the draft application specification is modeled as C functions in the Application layer, RTOS services as SystemC transactions in the Concurrency layer, and advanced version software as instruction-accurate (compiled) descriptions in the Architecture layer. Thanks to the adopted separation of concerns approach, functional (Application layer) and timed (Architecture layer) aspects can be separated. Functional and timed aspects are thus limited to the corresponding layers. Consequently, a cycle-accurate software description has its high-level functional equivalent inside the top layer. Here, the duplication of the application description follows and reinforces the separation of concerns. It eases embedded software design by allowing software IP reuse, simulation of code portions with heterogeneous development levels, and RTOS services exploration. Furthermore, the method can be equally applied to hardware implementation of the application tasks since the Application layer makes no assumption about the hardware/software partitioning. This co-existence of the task description and its implementation version is referred as a Simulation Couple (SC) in our framework. Thus coherent execution of the SC only depends on a common definition of synchronization points. Those correspond to the RTOS system calls present both in the high-level code and in the binary code. So the granularity of the Basic Blocks (BB) for the ISS is defined as the sequences of instructions between two system calls. Each task is associated a SystemC process and a synchronization event managed by the RTOS model and shared by all the BB of the task.

As depicted in Figure 12 the scheduler launches the highest priority ready task by notifying its synchronization event. The corresponding process is activated and its functional code executes in the top layer in zero simulation time till encountering a call to the RTOS API. The RTOS service first uses the HAL API and delegates the execution time evaluation of this BB to the PE. Without interrupt, the PE estimates the duration of the BB and advances the simulation time. If an interrupt occurs in the middle of a BB, the ISS stops at the corresponding date and saves task

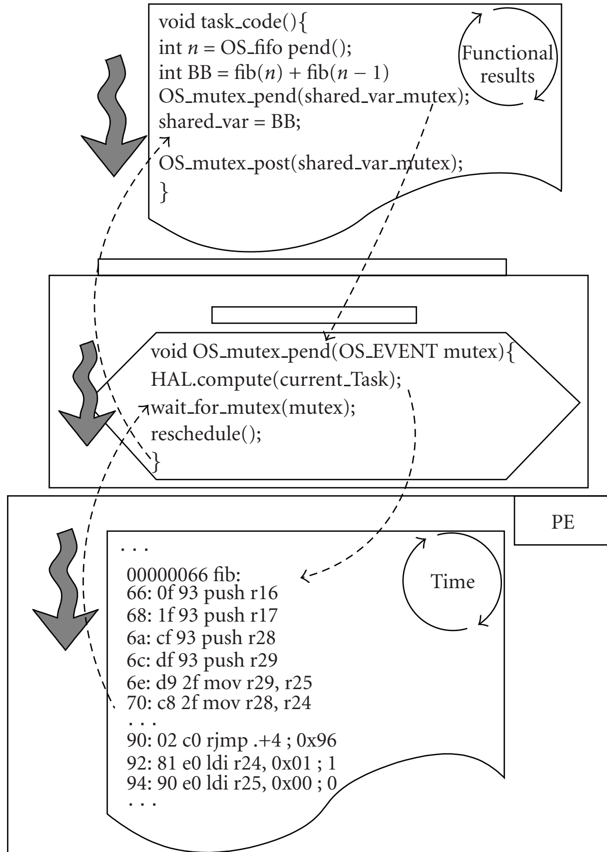



Figure 12: Example of a Simulation Couple. The software part of the application has two representations: a functionnal one used in high level abstraction layer and a timed one based on the use of an ISS.

context. The interrupt is then processed and the related routine is executed in the Concurrency layer. When the scheduler is reactivated, it can decide (according to the chosen scheduling policy) to resume the task or to elect a new one. The same scenario is repeated again until the end of the simulation.

## 6. Experiments

We applied our framework to a realistic application in the field of image processing for robotic vision. The application (see Figure 13) is used to learn object views or landscapes and extracts local visual features from the neighborhood of image's keypoints.

We specified at the Application layer the application as a set of 30 different communicating tasks and some of them could be run 400 times dynamically in parallel depending on the entry data as depicted in Figure 14. The full description of our application is out of the scope of this paper [46]. However, following a biologically inspired approach, this vision architecture belongs to a larger sensorimotor loop that brings interesting dynamical properties: the degree of parallelism and the execution time varies according to input data, namely the number of interest points, and the robot speed mode (high, intermediate, and low detail mode).

*6.1. Software Exploration.* In this context, we performed the profiling of the entire application on a hardware SoC platform. We also built the profile of the $\mu$C/OS-II [47] services (deterministic). For the purpose of the exploration we have targeted a Nios-II [48] based multiprocessor architecture (MPSoC) prototyped onto an Altera Cyclone-II FPGA circuit. The profiling of embedded software is a long and rigorous work which needs a non-preemptive and non-intrusive measurement technique. For this purpose, we modified the source code of the RTOS in order to provide such a measure technique both for the application basic blocks and for the OS services. After several executions onto a set of representative images, we built a timing data base for this application. For a simulation purpose, assigning a unique and representative execution time to the application tasks is a complex problem when the variance of the measured values is important. According to the refinement layers presented in Section 3.2, we currently recommend the use of an average value as a first approximation of the execution time and a stochastic draw into the timing data-base as a better estimation. Then, these timing data must be back-annotated into the high-level model in order to explore and evaluate the architecture dimensioning and the implementation strategies: tasks distribution, services distribution, scheduling algorithms, and so forth.

At this step, the application and the soft RTOS services were fully annotated into the Architecture layer. Following the design flow presented in Figure 1, we then performed a first set of simulations in order to evaluate the critical parts of the application when partitioned onto several processors. During these simulations the SystemC models related to the Architecture layer estimate the global system execution times. Figure 15(a) summarizes this information. Each plot represents an average value of the system performance for different images (number of keypoints). We can see that a pure software application could not be more accelerated using more than three processors (only a small gain between two and three). This MPSoC implementation reaches a global execution time of ≈27000 ms. Moreover we identified that the gaussian pyramid [46] represents the critical part of the application. So, we then explored the implementation of the related tasks into hardware in a reconfigurable device.

*6.2. Heterogeneous Exploration Based on System Metrics.* We deployed our application using a static partitioning between software and hardware tasks (more details can be found in [46]). The result of the partitioning is a set of 12 software tasks and a set of 18 hardware regular treatments. We realized the design of the hardware blocks in VHDL and back annotated the synthesis results (number of slices, execution times, communication latencies and configuration times) into the functional DRA model. The acceleration of

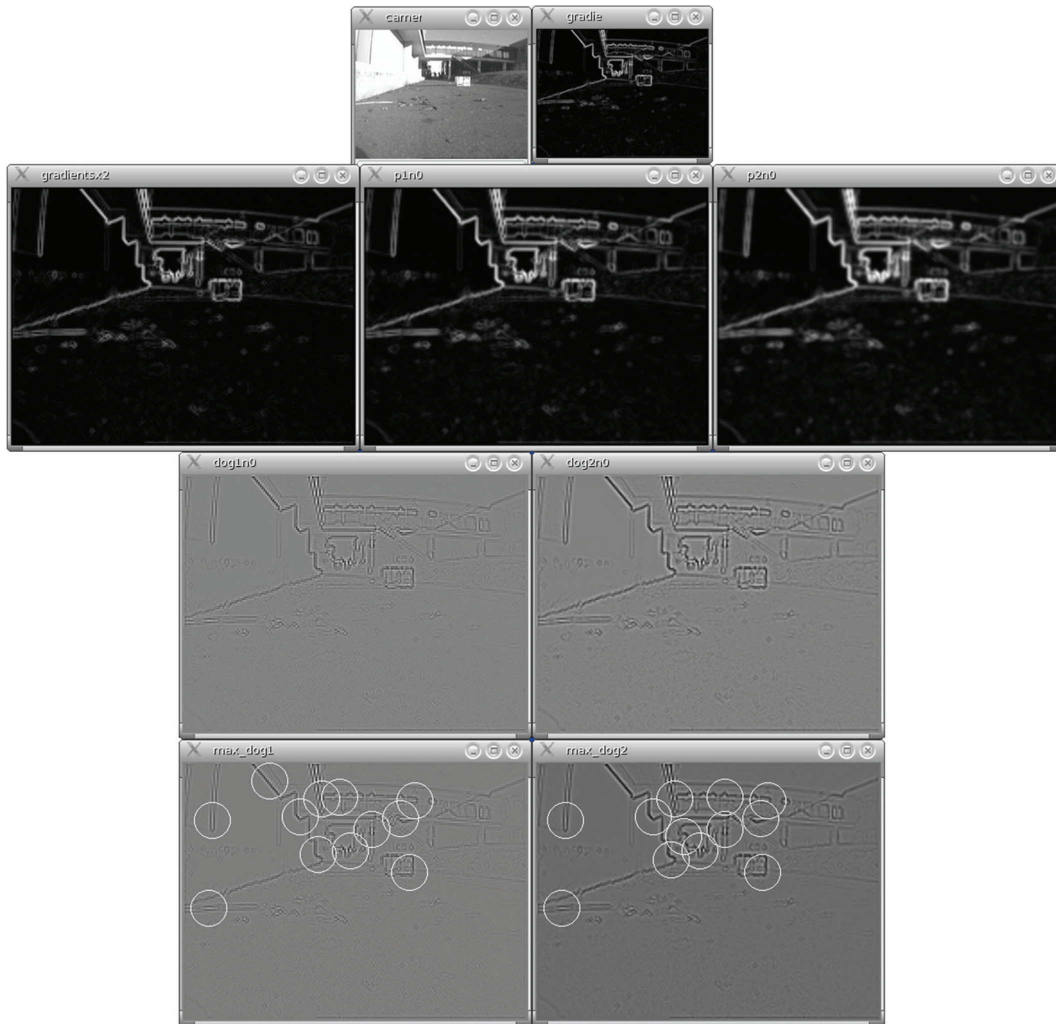



Figure 13: Graphical results of the SystemC functional model simulation of the robotic vision application.

a hardware implementation for the critical software tasks is very important: their total execution time is divided by a factor 4000. A second iteration of simulations (upper loop of Figure 1) was processed in order to define the new adequate architecture.

To figure out the right number of processors, we performed a new set of experimentations, as shown on Figure 15(b). The result of the second exploration is an architecture composed of 3 processors and a DRA with a yet undefined size. Indeed, the gain obtained by the hardware implementation of the gaussian pyramid permits to parallelize the 12 remaining software tasks to have a significant gain.

During this exploration/refinement process, the designer can use the system metrics presented in Section 3.1 and automatically extracted by the tool. Some examples of metrics used for the system dimensioning are the Gantt chart, the DRA chart (Figure 16) and the Communication chart depicted in Figure 17. The Gantt diagram represents the state of each task (software or hardware) along time: ready, running, waiting states and a configuring state for hardware tasks only. The Gantt charts of Figure 16(a) depict the new configuration of the system architecture. The 12 upper lines represent the ordering of the software tasks onto the 3 processors and the remaining lines represent the 18 hardware tasks running in parallel in the DRA. This architecture corresponds to the best achievable performances since the size of the DRA has been computed as the sum of the hardware tasks occupation. More precisely, the hardware partition uses near to 1200 slices (In the Virtex-5 FPGA slices are organized differently from previous generations. Each Virtex-5 FPGA slice contains four LUTs and four flip-flops -previously it was two LUTs and two flip-flops-), 14 BRAM and 12 DSP48 blocks. Hence, the estimated resource utilization for the global architecture (DRA + three processors) is about 4375 slices, 21 BRAM and 16 DSP48 blocks. This estimation would correspond for example to the size of a LX30T Virtex 5 circuit [49]. The global system latency ranging from 950 ms (Gantt of Figure 16) to about 60 ms depending of the application mode. We obtain about x28 acceleration compared to the pure software implementation.

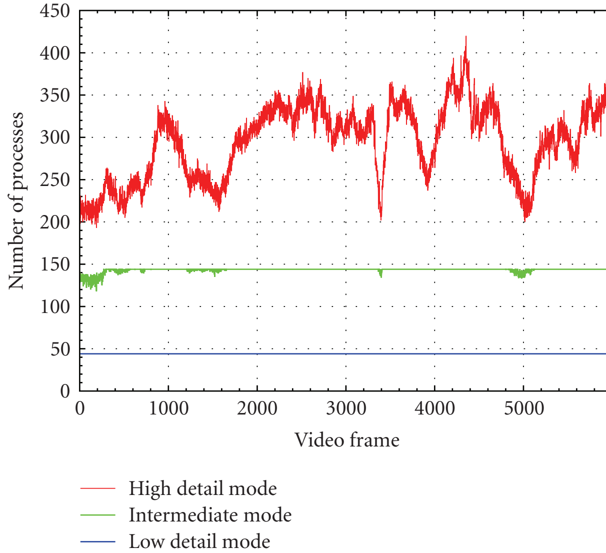



Figure 14: Number of processes created and managed by the OS model during the application simulation on a set of 6000 images for each modes.

*6.3. Reconfiguration Management.* In order to reduce the size of the hardware partition we vary the number of slices of the DRA and evaluated the capability of the system to adapt the hardware scheduling to a restricted area. In Figure 16, we present the results for one of the explored restricted architecture. We observe on the Gantt chart a different schedule of the hardware IP depending on the occupation rate of the DRA. The comparison between DRA charts of Figures 16(c) and 16(d) shows a clear difference in the utilization of the DRA over the time.

In the first case (Figures 16(a) and 16(c)), the DRA is never full and the tasks are configured as soon as the RTOS puts them in the Ready State. Here, the configuration only depends on the data dependencies in the application graph.

In the second case (Figures 16(b) and 16(d)) we consider a smaller DRA composed of 3000 slices. The DRA can not configure all the tasks at the same time. Here configuration depends both on the data dependencies and on the available resources. At level 1 of the DRA model, the hardware scheduler only manages available resources. It searches for sleeping tasks within the DRA to be replaced by a new task asking for resources. Besides, once a hardware task finishes its execution, it is removed (its resources are freed), enabling another task to be implemented. For the estimation of the configuration time we used a metric which depends on the size of the partial bitstream for the targeted DRA technology (about 50 $\mu$s per block of 16 CLBs on a Virtex 5).

As a first conclusion the exploration of the architecture for the robotic vision application leads us to model a complete RSoC platform at a high-level of abstraction. This high-level model focuses on the definition of the RTOS services needed by the identified architectures. For the systems presented in this section, we used as many OS as processors. All these components (Figure 9) are composed of the following services:

(i) a task management service to dynamically create keypoints extraction tasks,

(ii) several shared semaphores and mutex to synchronize the application and to protect image data into the shared memories,

(iii) a priority based scheduler on each processor,

(iv) a time management service for timeouts,

(v) an interrupt manager for the management of the multiprocessor architecture.

Also, another RTOS model is dedicated to the management of hardware tasks. This RTOS model provides several additional services:

(i) at level 1 of the DRA, a specific scheduling service using only the available resources,

(ii) at level 2 of the DRA, a refined scheduling service using also the localization and the shape of the tasks,

(iii) a placement service related to the level of the DRA model,

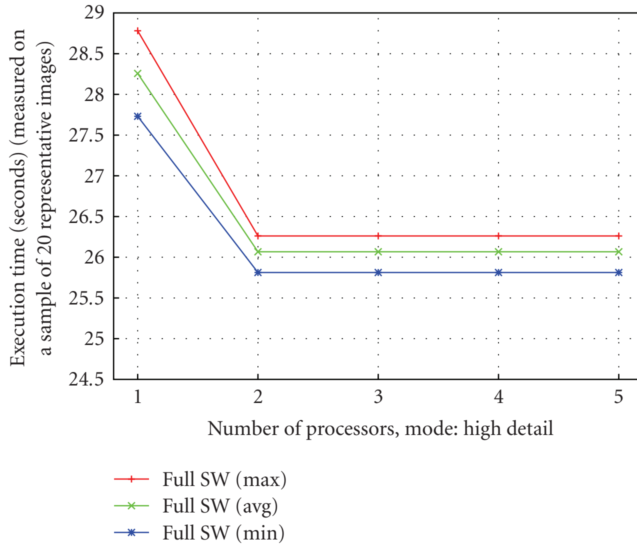
(iv) a communication service using hardware FIFO (results are presented on Figure 17),

(v) several mutex and semaphore proxies for the synchronization with software tasks.

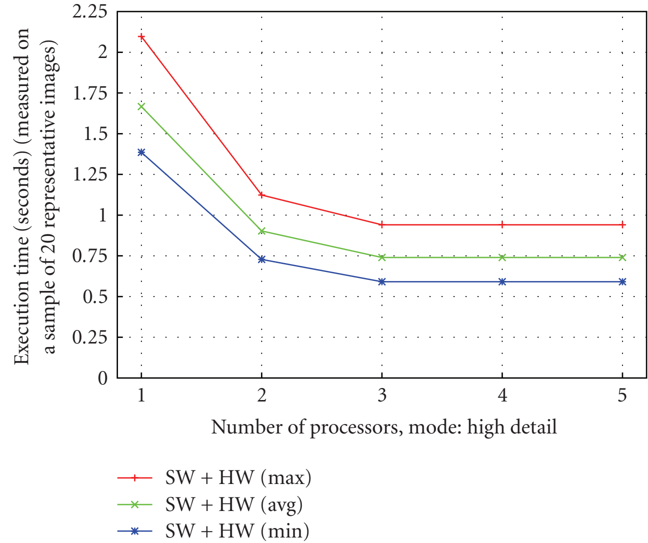
The refinement of the DRA to level 3 allows to test low-level hardware scheduling and placement strategies. We have implemented two simple placement algorithms to manage the DRA resources at a finer grain.

*6.4. Accuracy and Simulation Overhead of the Model.* To evaluate the efficiency of our modeling approach, we performed two sets of experiments. First, we evaluated the model accuracy and compared the simulated execution time relative to actual board measurements for multiple implementations. The average application times measured on board is 2926 ms and the simulated time gives 2836 ms. Those results validate our high level model considering the simulation's accuracy is within 3-4% of board measurements.

Then we evaluated the simulation time of the application on top of our RTOS model in comparison with a purely functional description. The deployment of the application tasks was explored and simulated using the Application and Concurrency layers of Figure 2. We vary the number of PEs within the architecture from 1 to 6 OS (Processors or DRA). Tasks execute and communicate in the same way on board and in simulation trough a single shared memory space protected with shared semaphores. Table 2 shows the scalability of our model. It indicates the simulation time $t_n$




(a) Pure software tasks implementation of the application




(b) Mixed hardware and software tasks implementation

Figure 15: Performance gain exploration for several sizes of MPRSOC architectures for case (a) all tasks in software; case (b) partitioned in hardware (on a DRA) and software on multiple processors.

Table 2: Simulation overhead versus number of OS.

| $n$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| simulation time $t_n$ (second) | 5.5 | 6 | 7.4 | 8.6 | 9.8 | 11.1 | 12.8 |
| overhead $s_n$ (%) | −8.9 | 0 | 23.3 | 43.3 | 63.3 | 85 | 113.3 |

of a platform modeled at the Concurrency layer composed of $n$ RTOS and the average simulation overhead $s_n = (t_n - t_1)/t_1$ for different platform sizes. $t_0$ represents the execution time of the pure functional application specification (at the Application layer). Simulations were realized on an Intel DualCore workstation running at 1.66 GHz with 2 GB of RAM.

For monoprocessor platforms, the RTOS model does not impact the simulation time since the overhead is only 8.9% more than the purely functional application description. Results indicate that the simulation time overhead is around 23% more per simulated RTOS. This overhead is due to the SystemC simulation kernel that works for the whole list of SystemC `sc_thread` of the system, which increases with the number of RTOS.

Finally, the framework allows to simulate an application in a functional and non-intrusive debug mode as illustrated in Figure 13.

*6.5. Perspectives.* We are now working on the integration of all the components into a basic and scalable target architecture which is composed of one ISS, a DRA model, a shared bus, a global memory and a distributed OS. The final platform model uses the three layers presented in Figure 2 (Application, Concurrency and Architecture layers) in order to provide a good tradeoff between performance accuracy and simulation overhead. The first experiments show that going down till the cycle-accurate level of the Architecture layer (ISS and CSS models) brings a simulation overhead 500 times longer compared to a timed simulation at the Concurrency layer.

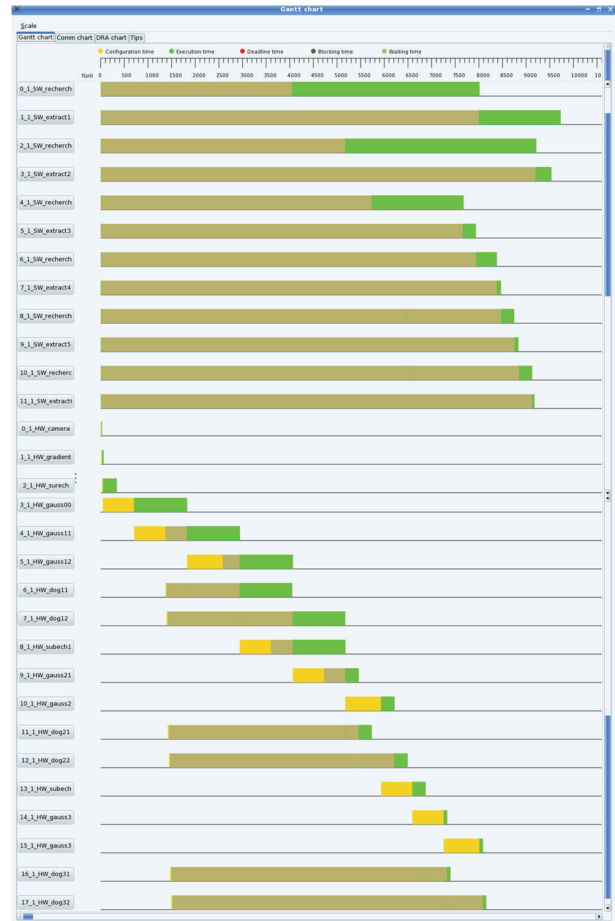
## 7. Conclusion and Perspectives

In this paper, we have presented a modeling framework for the design of a complete RSoC platform including processor(s), Dynamically Reconfigurable Architecture and OS services. The proposed design flow is based on a system level modeling approach which eases the exploration of the RTOS services distribution both onto processors and directly inside a reconfigurable region of the considered hardware unit. The main contribution of this work consists in proposing a unified modeling and refinement methodology for the software and the hardware parts of a dynamically reconfigurable system.

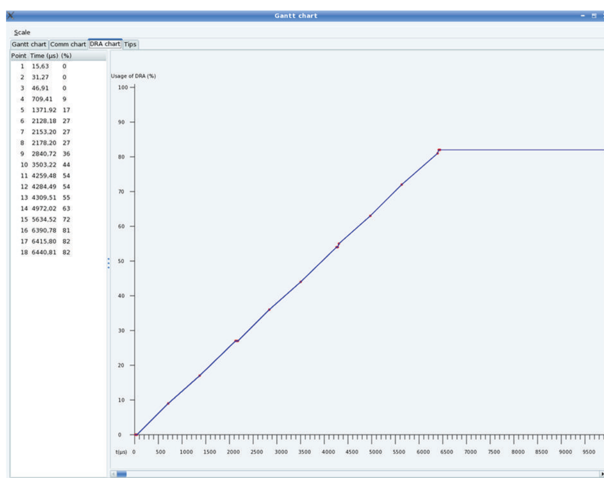
We have also listed the specific services that are needed in the literature for the management of the reconfigurable resources of the architecture. Thanks to a modular and flexible modeling approach we developed a library of generic components for the description of RSoC platforms. Among them, we developed basic hardware services such as hardware task management, hardware/software synchronization and bitstream management at high level of abstraction. The global method and the SystemC models were validated on an image processing application.

Today, the presented results show that the framework allows to define, simulate, and explore the specific services of RTOS for RSoC platforms very early in the design flow.

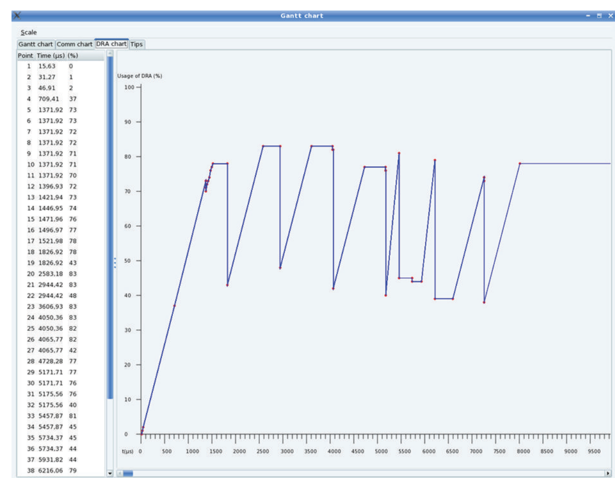
(a) Gantt Chart for large DRA



(b) Gantt Chart for smaller DRA



(c) Occupation rate of large DRA



(d) Occupation rate of smaller DRA

Figure 16: (a) and (b) represent the Gantt diagram for all the application tasks in both software on 3 processors and in hardware on a DRA of 4500 slices on the left and 3000 on the right. (c) and (d) represent the evolution of the DRA occupation over the time.

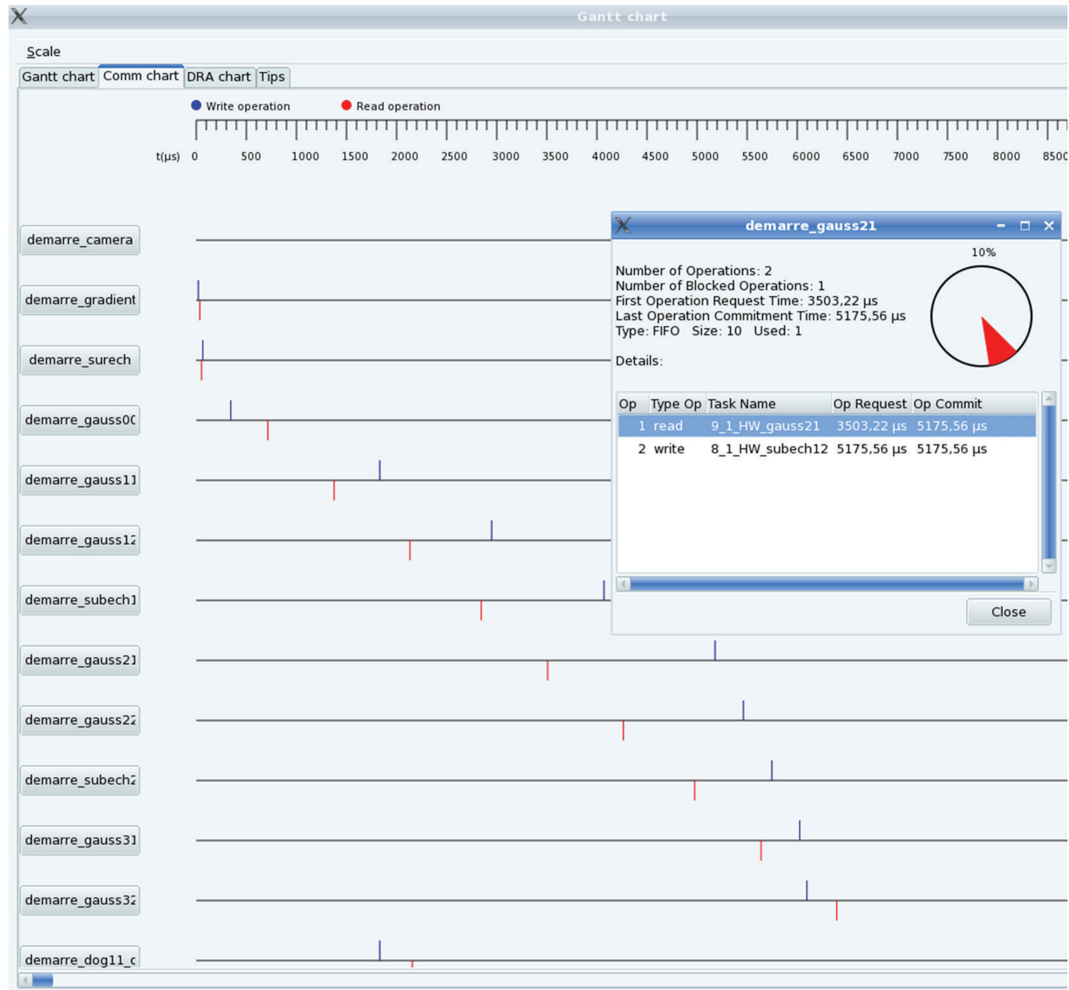



FIGURE 17: The DOGME tool provides several metrics helping the designer to evaluate the simulated design solutions. The window shows communications between tasks over time. It also computes the filling ratio for FIFO based communications.

Now, we have to refine some existing services such as the hardware scheduler at lower levels of abstraction in order to manage and estimate more accurately the resources used by an application on a real FPGA. We also have to extend the library of models: processing units, refined communication media and services such as placement algorithms from the literature. The OveRSoc framework could then be used as a comparison environment for upcoming methods in the context of DRA management.

## Acknowledgments

We would like to thank Sylvain Viateur for his help on the ISS SystemC model. The work presented in this paper was performed in the OveRSoC project which is supported by the french ANR funding.

## References

[1] O. Diessel and G. Wigley, “Opportunities for operating systems research in reconfigurable computing,” Technical Report ACRC-99-018, Advanced Computing Research Centre, School of Computer and Information Science, University of South Australia, Mawson Lakes, South Australia, 1999.

[2] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, “Designing an operating system for a heterogeneous reconfigurable SoC,” in *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS ’03)*, p. 7, April 2003.

[3] 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.

[4] J. Becker, M. Hubner, G. Hettich, R. Constapel, J. Eisenmann, and J. Luka, “Dynamic and partial FPGA exploitation,” *Proceedings of the IEEE*, vol. 95, no. 2, pp. 438–452, 2007.

[5] B. Miramond and J.-M. Delosme, “Design space exploration for dynamically reconfigurable architectures,” in *Proceedings of Design, Automation and Test in Europe (DATE ’05)*, pp. 366–371, Munich, Germany, March 2005.

[6] M. Yuan, X. He, and Z. Gu, “Hardware/software partitioning and static task scheduling on runtime reconfigurable FPGAs using a SMT solver,” in *Proceedings of the 14th IEEE*

*Real-Time and Embedded Technology and Applications Symposium (RTAS '08)*, pp. 295–304, St. Louis, Mo, USA, April 2008.

[7] K. Ramamritham and J. A. Stankovic, "Scheduling algorithms and operating systems support for real-time systems," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 55–67, 1994.

[8] G. Wigley and D. Kearney, "The first real operating system for reconfigurable computers," in *Proceedings of Australasian Conference on Computer Systems Architecture (ACSAC '01)*, pp. 130–137, IEEE Computer Society, 2001.

[9] F. Engel, I. Kuz, S. Petters, and S. Ruocco, "Operating systems on SoCs: a good idea?" in *Proceedings of IEEE Embedded Real-Time Systems Implementation Workshop (ERTSI '04)*, December 2004.

[10] I. Benkhermi, M. E. A. Benkhelifa, D. Chillet, S. Pillement, J.-C. Prevotet, and F. Verdier, "System-level modelling for reconfigurable SoCs," in *Proceedings of the 20th Conference on Design of Circuits and Integrated Systems (DCIS '05)*, Lisboa, Portugal, November 2005.

[11] K. Bazargan, R. Kastner, and M. Sarrafzadeh, "Fast template placement for reconfigurable computing systems," *IEEE Design and Test of Computers*, vol. 17, no. 1, pp. 68–83, 2000.

[12] A. Kuhn, F. Madlener, and S. Huss, "Resource management for dynamic reconfigurable hardware structures," in *Proceedings of Reconfigurable Communication Centric System-on-Chips (ReCoSoC '06)*, 2006.

[13] J. C. Van Der Veen, S. P. Fekete, M. Majer, et al., "Defragmenting the module layout of a partially reconfigurable device," in *Proceedings of the 5th International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '05)*, pp. 92–101, Las Vegas, Nev, USA, June 2005.

[14] K. Puma and D. Bhatia, "Temporal partitioning and scheduling data flow graphs for re-configurable computers," *IEEE Transactions on Computers*, vol. 48, no. 6, pp. 579–590, 1999.

[15] J. Resano, D. Mozos, D. Verkest, F. Catthoor, and S. Vernalde, "Specific scheduling support to minimize the reconfiguration overhead of dynamically reconfigurable hardware," in *Proceedings of the 41st Annual Design Automation Conference (DAC '04)*, pp. 119–124, San Diego, Calif, USA, 2004.

[16] L. Levinson, R. Manner, M. Sessler, and H. Simmler, "Preemptive multitasking on FPGAs," in *Proceedings of the 8th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '00)*, pp. 301–302, 2000.

[17] H. Simmler, L. Levinson, and R. Männer, "Multitasking on FPGA coprocessors," in *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*, vol. 1896 of *Lecture Notes in Computer Science*, pp. 121–130, Springer, Berlin, Germany, 2000.

[18] D. Koch, A. Ahmadinia, C. Bobda, and H. Kalte, "FPGA architecture extensions for preemptive multitasking and hardware defragmentation," in *Proceedings of IEEE International Conference on Field-Programmable Technology (FPT '04)*, vol. 3203 of *Lecture Notes in Computer Science*, pp. 433–436, Brisbane, Australia, December 2004.

[19] T. Marescox, A. Bartic, B. Verkest, S. Vernalde, and R. Lauwereins, "Interconnection networks enable fine-grain dynamic multitasking on FPGAs," in *Proceedings of the 12th Field Programmable Logic and Applications Conference*, vol. 2438, pp. 795–804, September 2002.

[20] Symbad, "SYMBAD—Formal Verification in SYsteM Level Based Design," 2002, http://www.setnet.org/Research/SYMBAD.htm.

[21] M. Borgatti, A. Capello, U. Rossi, et al., "An integrated design and verification methodology for reconfigurable multimedia systems," in *Proceedings of Design, Automation and Test in Europe (DATE '05)*, pp. 266–271, Munich, Germany, March 2005.

[22] P. Hsiung, C. Liao, C. Tseng, S. Lin, Y. Chen, and K. Chiu, "Hardware-software codesign and coverification methodology for dynamically reconfigurable system-on-chips," in *Proceedings of Workshop on Object-Oriented Technology and Applications (OOTA '04)*, 2004.

[23] Y. Qu, K. Tiensyrjä, and K. Masselos, "System-level modeling of dynamically reconfigurable co-processors," in *Field Programmable Logic and Application*, vol. 3203 of *Lecture Notes in Computer Science*, pp. 881–885, Springer, Berlin, Germany, 2004.

[24] P.-A. Hsiung, C.-H. Huang, and C.-F. Liao, "Perfecto: a systemc-based performance evaluation framework for dynamically partially reconfigurable systems," in *Proceedings of International Conference on Field Programmable Logic and Applications (FPL '06)*, Lecture Notes in Computer Science, pp. 190–195, Madrid, Spain, August 2006.

[25] A. Raabe, P. A. Hartmann, and J. K. Anlauf, "ReChannel: describing and simulating reconfigurable hardware in systemC," *ACM Transactions on Design Automation of Electronic Systems*, vol. 13, no. 1, pp. 112–120, 2008.

[26] K. Benkrid, A. Benkrid, and S. Belkacemi, "Efficient FPGA hardware development: a multi-language approach," *Journal of Systems Architecture*, vol. 53, no. 4, pp. 184–209, 2007.

[27] A. Herrholz, F. Oppenheimer, P. A. Hartmann, et al., "The ANDRES project: analysis and design of run-time reconfigurable, heterogeneous systems," in *Proceedings of International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 396–401, August 2007.

[28] J. J. Lee and V. J. Mooney III, "Hardware/software partitioning of operating systems: focus on deadlock detection and avoidance," *IEE Proceedings: Computers and Digital Techniques*, vol. 152, no. 2, pp. 167–182, 2005.

[29] G. Smit, E. Schüler, J. Becker, J. Quévremont, and W. Brugger, "Overview of the 4S project," in *Proceedings of International Symposium on System-on-Chip*, pp. 70–73, Tampere, Finland, November 2005.

[30] OSCI, "IEEE 1666$^{TM}$ Standard SystemC Language," http://www.systemc.org/.

[31] E. A. Lee and S. Neuendorffer, "Concurrent models of computation for embedded software," *IEE Proceedings: Computers and Digital Techniques*, vol. 152, no. 2, pp. 239–250, 2005.

[32] K. Keutzer, "System-level design: orthogonalization of concerns and platform-based design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 12, pp. 1523–1543, 2000.

[33] P. Kohout, B. Ganesh, and B. Jacob, "Hardware support for real-time operating systems," in *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '03)*, pp. 45–51, Newport Beach, Calif, USA, October 2003.

[34] P. Gerin, H. Shen, A. Chureau, A. Bouchhima, and A. A. Jerraya, "Flexible and executable hardware/software interface modeling for multiprocessor SoC design using systemC," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '07)*, pp. 390–395, IEEE Computer Society, Washington, DC, USA, 2007.

[35] A. Donlin, "Transaction level modeling: flows and use models," in *Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and Systems Synthesis (CODES+ISSS '04)*, pp. 75–80, Stockholm, sweden, 2004.

[36] OCP-IP, "Open core protocol international partnership," http://www.ocpip.org/.

[37] E. Huck, B. Miramond, and F. Verdier, "A modular systemC RTOS model for embedded services explorations," in *Proceedings of Conference on Design and Architectures for Signal and Image Processing (DASIP '07)*, Grenoble, France, 2007.

[38] B. Miramond, F. Verdier, and M. Aichouch, "DOGME distributed operating system graphical modeling environment," http://oversoc.ensea.fr/ oversoc-graphical-modeling-environment-1.

[39] "Eclipse rich client platform," http://eclipsercp.org/.

[40] D. Desmet, D. Verkest, and H. De Man, "Operating system based software generation for systems-on-chip," in *Proceedings of Design Automation Conference*, pp. 396–401, 2000.

[41] P. Hastono, S. Klaus, and S. A. Huss, "Real-time operating system services for realistic systemc simulation models of embedded systems," in *Proceedings of Forum on Specification and Design Languages (FDL '04)*, pp. 380–392, Lille, France, September 2004.

[42] H. Posadas, J. A. Adamez, E. Villar, F. Blasco, and F. Escuder, "RTOS modeling in SystemC for real-time embedded SW simulation: a POSIX model," *Design Automation for Embedded Systems*, vol. 10, no. 4, pp. 209–227, 2005.

[43] Z. He, A. Mok, and C. Peng, "Timed RTOS modeling for embedded system design," in *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '05)*, pp. 448–457, San Francisco, Calif, USA, March 2005.

[44] P.-A. Hsiung, C.-H. Huang, and Y.-H. Chen, "Hardware task scheduling and placement in operating systems for dynamically reconfigurable SoC," *Journal of Embedded Computing*, vol. 3, no. 1, pp. 53–62, 2009.

[45] M. Ullmann, M. Hübner, and J. Becker, "On-demand FPGA run-time system for flexible and dynamical reconfiguration," *International Journal of Engineering Simulation*, vol. 1, no. 3-4, pp. 193–204, 2005.

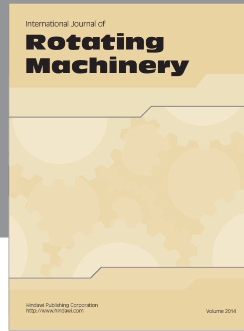
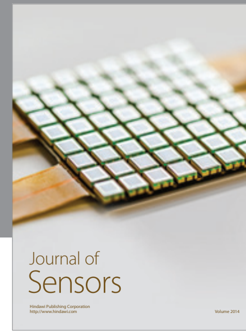
[46] F. Verdier, B. Miramond, M. Maillard, E. Huck, and T. Lefebvre, "Using high-level RTOS models for HW/SW embedded architecture exploration: case study on mobile robotic vision," *EURASIP Journal on Embedded Systems*, vol. 2008, no. 1, Article ID 349465, 17 pages, 2008.

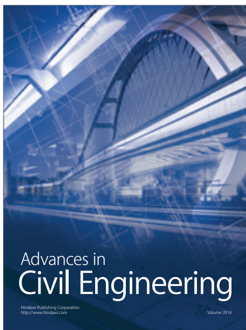
[47] J. Labrosse, "MicroC/OS-II: the real-time kernel," CMP Media, 2002, http://www.micrium.com/page/support/bookstore.

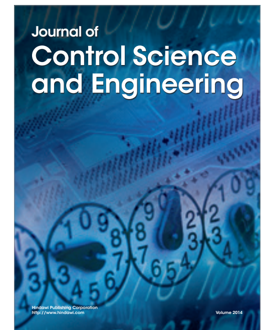
[48] "Altera," http://www.altera.com/.

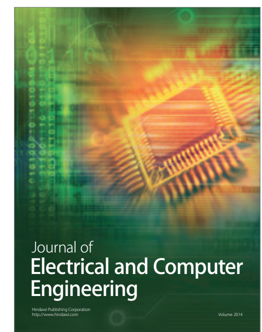
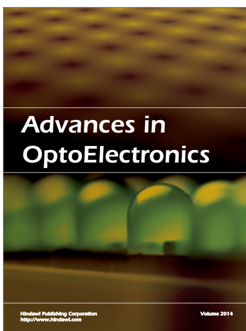
[49] Xilinx, "Virtex 5 familly overview," http://www.xilinx.com/.

Journal of
Engineering

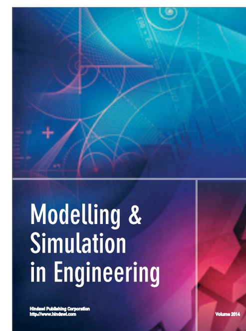
The Scientific
World Journal

International Journal of
Rotating
Machinery

Journal of
Sensors

International Journal of
Distributed
Sensor Networks

Advances in
Civil Engineering

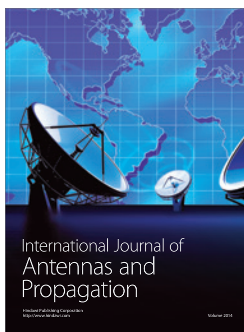
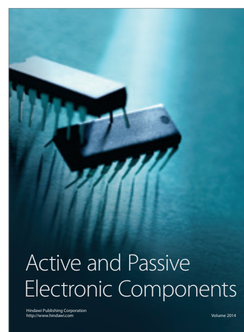
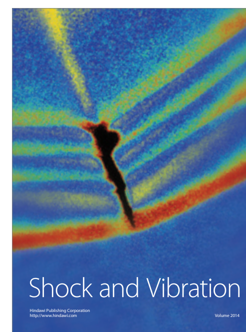
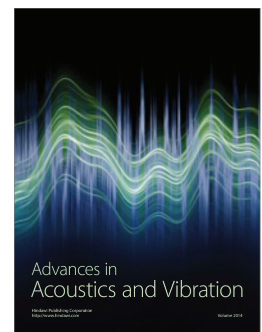
Hindawi
Submit your manuscripts at
http://www.hindawi.com

Journal of
Control Science
and Engineering

Journal of
Robotics

Journal of
Electrical and Computer
Engineering

Advances in
OptoElectronics

VLSI Design

International Journal of
Navigation and
Observation

Modelling &
Simulation
in Engineering

International Journal of
Aerospace
Engineering

International Journal of
Chemical Engineering

International Journal of
Antennas and
Propagation

Active and Passive
Electronic Components

Shock and Vibration

Advances in
Acoustics and Vibration