Software Platform based Embedded Multiprocessor SoC Prototyping

Benaoumeur Senouci, Ali El Moussaoui, Bernard Goossens
University of Perpignan
52, avenue Paul Alduy, F-66860 Perpignan cedex, France.

Abstract. This paper describes our experience in processor/threads synchronization using the POSIX API standard for MPSoC virtual applications prototyping. Spin-Lock (Binary Semaphore) implementations on general purpose CPU are based on an atomic read and (conditional) write of a shared variable. In modern multiprocessor implementations, these operations occur as dependent pairs of conditional instructions, such as load linked and store conditional. We present and discuss how a hardware semaphore could be a more efficient mechanism for processor/threads synchronisation that is CPU family independent. This mechanism has been implemented for an SMP Operating System with a validation on a top of a multi-ARM software platform.

Keywords: Embedded Operating System, ARM software Platform, Multiprocessor System on Chip (MPSoC) simulation, Hardware/Software co-design

1 Introduction

Embedded system is application-oriented special computer system which is scalable on both software and hardware. It can satisfy the strict requirement of functionality, reliability, cost, volume, and power consumption of the particular application. With rapid development of IC design and manufacture, CPUs became cheap. Lots of consumer electronics have embedded CPU and thus became embedded systems. For example, PDAs, cell phones, point-of-sale devices, VCRs, industrial robot control, or even your toasters can be embedded system. There is more and more demand on the embedded system market. Some report expects that the demand on embedded CPUs is 10 times as large as general purpose PC CPUs (GPP).

The emerging trend for multimedia applications on mobile terminals, combined with a decreasing time-to-market and a multitude of standards have created the need for flexible and scalable computing platforms that are capable of providing considerable (application specific) computational performance at a low cost and a low energy budget.

Hence, in recent years, the first multiprocessor system-on-chip components have emerged. These platforms contain multiple heterogeneous, flexible processing elements, core, a memory hierarchy and I/O components. All these components are linked to each other by a flexible on-chip interconnect structure. These architectures

meet the performance needs of contemporary and future multimedia applications, while limiting the power consumption.

MPSoCs embedded software designers are continually challenged to provide increased computational power for today's multimedia and telecommunication applications to meet tighter system requirements at ever-improving price/performance ratios. In this work we use the MPSoC centralized Operating System (OS) approach, to design a Symmetric Multiprocessor (SMP) operating system based on POSIX API standard. Once the OS architecture is frozen, the designer has some choice of building the different OS services (functions) parts, which could have software based or a hardware-based (hardware) implementation. The main contribution of this work is to validate an SMP (Symmetric MultiProcrssor) Operating System named Mutek on a multiprocessor software platform. The rest of the paper is organized as follows: section 2 details the Prototyping flow using POSIX-based Operating System and the virtual prototyping flow using an SMP OS. Section 3 describes the Mutek Operating System. Section 4 gives an insight of implemented simulation tool MaxSim, while section 5 concludes the paper.

2 Virtual Prototyping Flow using POSIX-Complaint Operating System

The main difficulties when designing multiprocessor system on chip (MPSoC) is the parallelization of sequential code and mapping of functions on the multiprocessor architecture. This is partially true and explains recent studies such as parallel programming models [1].

2.1 POSIX multithreads programming model

POSIX programming model is expedient for describing multi-thread embedded software. The POSIX standard can greatly increase the portability of applications and target several prototyping platforms. However, the desired application to prototype and the configuration architecture of the platform are captured separately; the design procedures adapt the application to port it onto the platform architecture to realize the final implementation [4]. In reality this mind-set process includes an integration step when platform specificities need to be abstracted and embedded software has to be adapted. That is called hardware/software integration. This process requires the development of HAL parts; since it is not obvious that an MPSoC's embedded SW that runs onto one configuration platform architecture will evidently run onto another one. Consequently each MPSoC's embedded software may have its own HAL API implementation, reflecting the specificities of each configuration of the platform architecture (Memory map, processors type, multiprocessor booting, commutation …).

2.2 Hardware Abstraction Layer (HAL)

MPSoC's embedded software is divided in several parts: Application, Middleware, OS, and Hardware Abstraction Layer code (Figure 1). The application is a set of concurrent threads that synchronize, exchange data and will be distributed dynamically/statically on several computing CPUs across a multiprocessor OS. The Middleware consist in communication libraries. In practice, OSs are structured in two parts: SW code which depends on the HW architecture (HAL) and SW code which is independent.

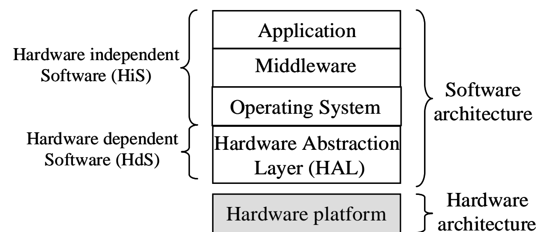

Figure 1: MPSoC's Embedded SW

The Hardware Abstraction Layer (HAL) concept is introduced to exactly talk about the low level programming practice of MPSoC's embedded software [6]. The HAL parts include those "low level" software functionalities whose implementations depend directly upon the underlying hardware platform architecture (Hardware-dependent Software (HdS) in other analogies). This includes also device drivers, DSP-specific algorithms, interrupt management, context related operations, semaphores and so on.

Our paper implements PVP parallel virtual prototype modeling techniques allowing the SW prototype implementation in parallel with HW prototype implementation refinement of the SW VP and HW VP is done by use a standard abstraction for the software that "sits directly on top" of the hardware.

2.3 Hardware/Software integration flow

The conventional design flow is that the HW architecture is designed first, and then the SW design is performed based on the fixed HW architecture. In terms of design cycle, this practice takes a long design cycle since the SW and HW design steps are sequential.

The PVP concept works on the disadvantages of such low-level programming practices by dividing the VP design in to two parallel VP : Hardware VP design and Software VP design having embedded software code in two parts: code that depends on the hardware architecture (the HdS) and code that is implementation independent (the hardware-independent software).

The hardware-independent software comprises application, middle ware, and operating system software. We assume that the application software comprises a set of concurrent tasks and the middle ware software represents dedicated communication libraries. The operating system software provides a useful abstraction interface

between applications and target architectures by simplifying the control code required to coordinate processes.

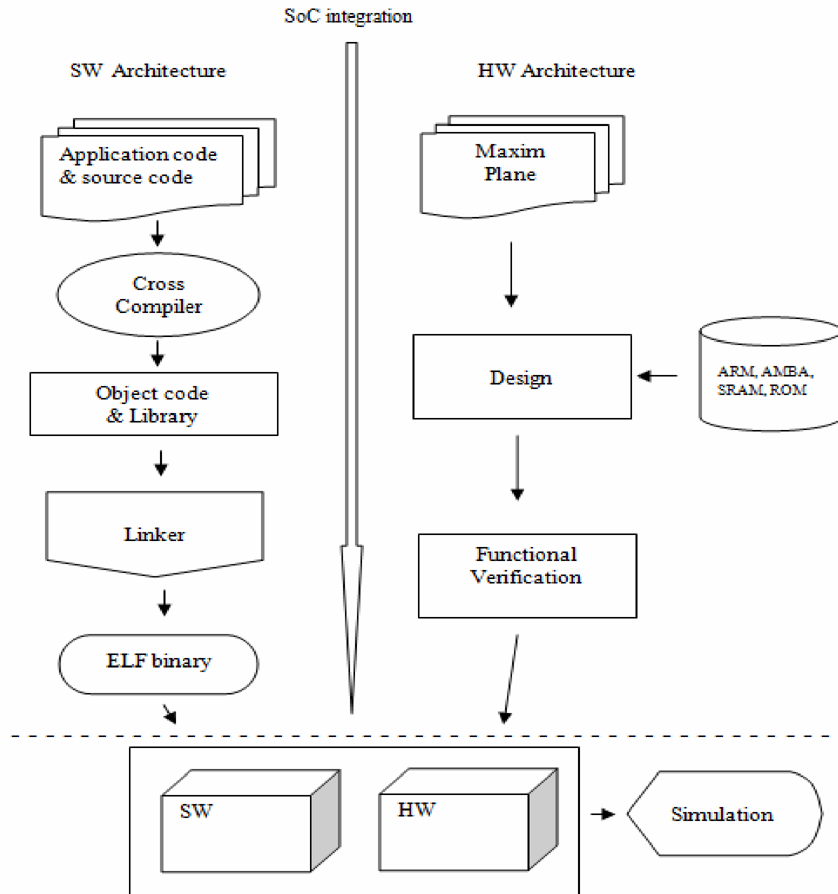

Figure 2: SoC Integration flow

In this section we present the MPSoC design flow from PVP model as show in (Figure 2), The Hardware prototype is designed from a component library composed of processors, memories, and communication structures where processors are modeled by cycle accurate processor model. Also the modeling peripheral components such as timers, interrupt controllers make possible the development of hardware-dependent software (HdS). After hardware designing, Functional validation comes the explicit mapping of the code in the different memories available in the architecture.

In the Software prototype (we used Mutek OS) the application and the source code can be parameterized in a number of ways through the use of pre-processor variable definitions within the code. Then the libraries have to compile for specific target

architecture then the application code linked to the software libraries in order to make an ELF file that can be used to boot the platform and run the application code.

3 Implementation

3.1 Software Architecture

POSIX “Portable Operating System Interface" is the collective name of a family of related standards specified by the IEEE to define the application programming interface (API) for software compatible with variants of the Unix operating system, although the standard can apply to any operating system.

An application programming interface (API) is a source code interface that an operating system or library provides to support requests for services to be made of it by computer programs. Mutek operating system is a lightweight kernel that proposes an implementation of the POSIX threads as API for multiprocessor platforms with shared memory. The Mutek operating system is available as part of the DISYDENT Open Embedded System Development Environment [2, 3].

The Mutek kernel uses a monolithic architecture in which both the operating system and application code are statically linked at compile time. The Mutek source code can be parameterized in a number of ways through the use of pre-processor variable definitions within the code. Once the libraries have been compiled for specific target architecture the application code can be linked to the software libraries in order to make an ELF file that can be used to boot the platform and run the application code.

Mutek uses a flat memory model running in physical address mode. The application code is run within the same memory address space as the kernel services thus allowing a complete control over the memory allocation of the objects in exchange for the memory protection mechanism that are usually available on more complex systems. The linker script configuration file (ldscript) must be explicitly given during the linking process. This file defines the memory mapping of all objects and sections used within the compiled code.

The Mutek functionalities are separated into the following libraries:

- libhandler: platform specific code. The library proposes a hardware abstraction layer on top of which all the Mutek functionalities are built. This library contains the platform specific assembly source code.
- libpthread: Posix thread implementation that conforms to the Posix.
- libc: tiny libc library that can be used by application code.
- lcmomo, lmalloc: Disydent Process Network library. This library proposes communication channel abstraction that can be used to hide hardware and software communications behind a unified fifo-based communication framework.

Mutek is an open source project that implements a POSIX compliant multiprocessor kernel [5]. Its architecture is depicted in Figure .

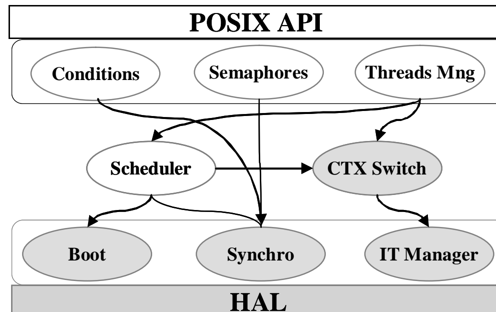

Figure 3: Mutek Architecture

3.1.1 Scheduler

The scheduler manages several lists of threads. It may be shared by all processors (SMP for Symmetrical Multi-Processor). In the SMP scheduler organization, there is a unique scheduler shared by all processors and protected by a lock. The different threads can run on any processor, which leads to task migration.

3.1.2 Kernel protection and thread Migration

The access to the scheduler must be performed through critical section, and under the protection of locks. Lock granularity is one major player in determining the balance between the overhead introduced by the locking mechanism and the opportunity to increase parallelism among different processors.

The SMP version of Mutek allows thread migration. The term migration is related to the ability of an operating system to resume a job on another processor after pre-emption. Intuitively when a CPU finishes the threads currently allocated on it for scheduling, it can resume the execution of a pre-empted thread that was previously executed on another processor. In that way, the system is dynamically balanced, reducing the mean response time of the system.

3.1.3 Thread synchronization

Synchronization is required whenever shared data need to be accessed. In Mutek, this is done using different primitives: Spin-Lock, Mutual exclusion locks, condition variables and semaphores.

The binary semaphore is the lowest level Mutual exclusion object. Thread waiting for a Spin-Lock continually accesses the element and do not return until they grab it. It is an active wait element.

A Mutual exclusion lock (Mutex) allows excusive access to shared resources such as global data. Threads attempting to access an object locked by a Mutex will be blocked until the thread holding the object releases it.

Using condition variables, a thread can wait until (or indicate that) a predicate becomes true. A condition variable requires a Mutex to protect the data associated with the predicate.

In POSIX.1b, named and unnamed semaphores have been "tuned" specifically for threads. The semaphore is initialized to a certain value and decremented. Threads may wait to acquire a semaphore. If the current value of the semaphore is greater than 0, it is decremented and the wait call returns. If the value is 0 or less, the thread is blocked until the semaphore is available.

3.1.4 Processor identification number

Processors generally provide a specialized register allowing their identification within the system. Each processor is assigned a number at boot time. This identification number is needed, for instance, at boot time, since some start-up actions should be done only once, such as clearing the Blank Static Storage (BSS) and creating the scheduler.

3.2 Hardware Architecture

We used a software architecture model based on MaxSim environment. MaxSim [7] is an environment for System-Level Simulation and Design based in SystemC. The designer can construct a virtual prototype by assembling the system from a component library. This component library is composed of processors, buses, memories, interrupt controllers and others. Therefore, the library can be extended with custom components described by the designer.

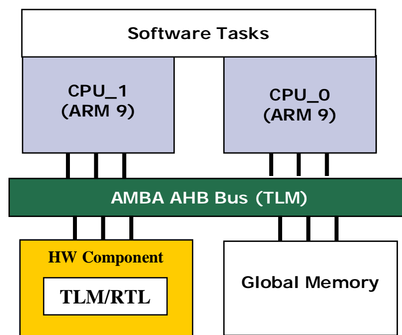

Figure 4: ARM Based Architecture

The computational model of the MaxSim components is based in a cycle-based engine. In this kind of component the behavior is evaluated only in the clock edges. Behavior is described in two methods, communicate and update. In the communicate method all communication between the components are performed, whereas in the update method the communication realized are committed in the shared resources. This way of modeling leads to high simulation speeds, enabling the fast validation of the architecture. Figure 4 shows our multiprocessor architecture model with two

ARM processors, AMBA bus and a global memory; also we can add several HW components written in SystemC at different level of abstraction (RTL or TLM).

3.3. Software Mapping

The boot code sequence provides the system initialization, after resetting processors jump to a specific shared address alias in the global memory address space which is set for starting the first initialization routine inside the operating system kernel (__init) (Figure 5). The binary (.ELF) image is loaded on the global of the multiprocessor architecture, on a specific address.

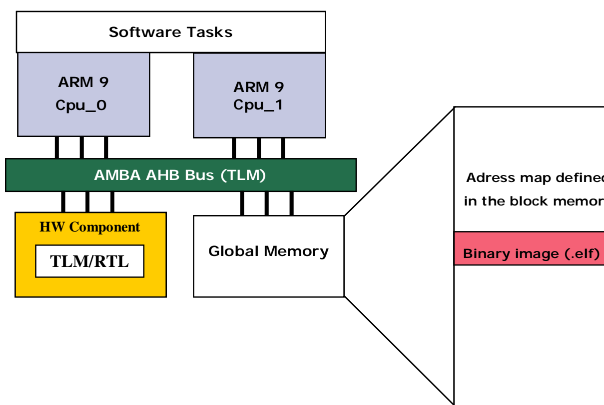

Figure 5: Memory mapping

3.4 Synchronization implementation

3.4.1 CPU-based Spin-Lock implementation

In MPSoC systems, the only non blocking synchronization primitive that is used to ensure atomic access to shared objects is the Spin-Lock. Binary Spin-Lock semaphore implementations on general purpose CPU's are based an atomic read and (conditional) write of a shared variable.

```
// semaddr in R0
void SEM_LOCK (unsigned int semaddr) {
__asm {
            MVN     R2, #0              // load the value (-1)
Tryagain:   SWP     R3, R2, [R0]        // Atomic load and store
            CMN     R3, #1              // did this succeed?
            BEQ     Tryagain            // no – try again
  }; //yes – we have the lock, so Branch in R14
}
void SPIN_UNLOCK (unsigned int semaddr) {
__asm {
            MOV     R2, #0              //load the 'free' value (0)
            STR     R2, [R0]            //open the lock
  };
}
```

Figure 6: CPU-based SEM-LOCK implementation for ARM processor

In modern multiprocessor implementations, these operations occur as dependent pairs of conditional instructions, such as load linked and store conditional [7]. These instructions have evolved from the original test and set, and compare and swap type instructions. These existing mechanisms can be integrated in shared memory multiprocessors (SMP) to provide synchronization between applications running on multiple homogeneous CPUs. Figure 6 shows the implementation of the different functions (SEM_LOCK & SPIN_UNLOCK) using the specific swap (SWP) multiprocessor atomic instruction. SEM_LOCK is used to check the Spin-Lock variable and SPIN_UNLOCK to release it.

In the ARM based architecture we use the swap (SWP) ARM instruction, and "*lock*" the system bus for Spin-Lock implementation. A processor could hold the entire bus until atomic load and store (SWP) instruction completion of the "*semaddr*" global variable, disallowing all other processors. Clearly, implementing Spin-Lock with such mechanisms, which lock the system bus until completion, was offensive for system performance. In order to eliminate the need to lock the system bus for long periods of time, the new ARM instruction set (ex: ARM11) introduced two new instructions load-exclusive (LDREX) and store-exclusive (STREX) which take advantage of an exclusive monitor in memory. These instructions require additional control logic within the CPU that interfaces into the memory coherency policy of the system. While semantically correct, these existing mechanisms introduce significant complexity in the system design that is not easily portable for all CPU family, when creating heterogeneous multiprocessor SoC [8].

3.4.2 Hardware-based synchronization

We can use a hardware component to implement more efficient synchronization mechanisms that are CPU family independent, and require no additional control logic (exclusive monitor). As such, this new mechanisms are easily portable across shared and distributed memory multiprocessor configurations. Thus, the architecture can implement synchronisation that does not lock the system bus that grants other processors or threads access to the memory system. Also this approach is suitable to none centralized interconnect where CPU can not snoop the bus for cache coherency.

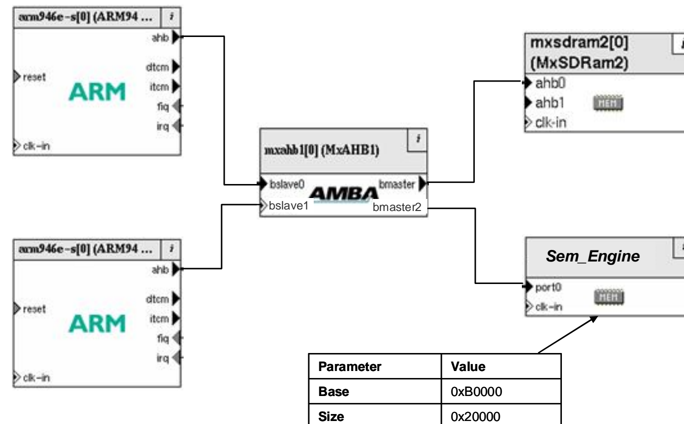

Figure 7: MaxSim based Architecture

3.4.3 Semaphore Engine

The Semaphore Engine defined uses a standard read of a memory mapped register "Sem_addr". We define a simple control structure that updates the register after a read operation. Figure 8 shows the implementation of the semaphore engine component.

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
void SEM_LOCK (unsigned int semaddr) {
while (Sem_addr !=0);
}
void SPIN_UNLOCK (unsigned int semaddr) {
Sem_addr =1;
}
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Figure 8: CPU-independent Spin-Lock implementation

The basic semantics of all SEM_LOCK & SEM_UNLOCK API's for accessing the lock are implemented identically for all system processors (Figure).

We use a finite state machine (FSM) to control the "Sem_addr" register.

Figure 9 shows the four states of the FSM (EM_IDLE, EM_0, EM_1 and EM_2). In *EM_Idle* state we initialize the "*Sem_addr*" register (Sem_addr = 0). To request the Sem-Lock variable, the SEM_LOCK API first reads the "Sem_addr" register (EM_0). When the "*Sem_addr*" value is read, if the Spin-Lock is free (Sem_addr =0), then the control logic implemented as a state machine within the Sem-Lock Engine IP updates the "Sem_addr" register (Sem_addr <=1) (EM_1). If the Spin-Lock is currently locked, then the control logic performs no update. After the first access, the lock is only freed when a SEM_UNLOCK API writes into the "Sem_addr" register (EM_2).

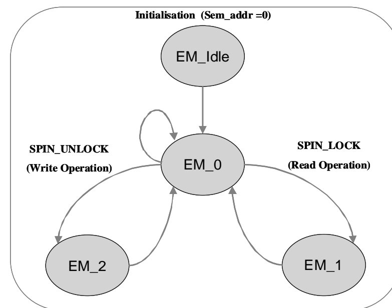

Figure 9: Sem-Engine Engine controller

4 Results and performance analysis

4.1 Discussion

As well to functional tests, tests were performed to quantify the performance of the hardware Semaphore-Engine. This was in response to our concern that the speed of the hardware implementation could guide to more performance than software one. The test sequence was a SEM_LOCK (request) and SEM_UNLOCK (release) (Figure 10).

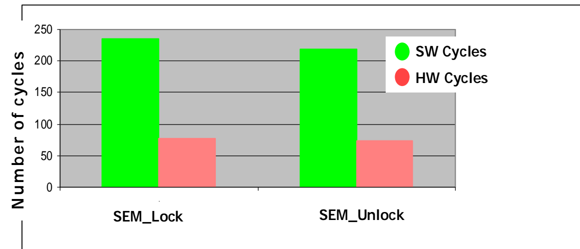

Figure 10: Semaphore performance request and release

The software Spin-Lock implementation average access time was 162 clock cycles, compared to 54 clock cycles for the hardware implementation one, yielding a 3 average performance access ratio.

The operating system is the most fundamental software layer in the MPSoC's embedded software, by simplifying control code required to coordinate processes and abstracting a wide range of high-level system services and device-specific requirements into a generic set of interfaces through which both application and system programs access specific operating- system capabilities. It provides a software adaptation between the application and the hardware architecture. Creation of an OS services based on hardware component, which eliminates the difference between software and hardware from the developer's point of view, requires a hardware-software co-design of portions of the OS to extend system services across the hardware-Software boundary.

One of the most attractive goals of hardware/software co-design of some OS services, in the MPSoC's embedded software design framework, is to augments the portability of applications on several mixed multiprocessor architecture by the based hardware implementation of the entire OS services which are CPU dependent. This hardware-based design of some OS services which is CPU dependent such as synchronization, processor identification, and interrupts controller promise to overcome the Hardware Abstraction Layer (Hardware dependent Software parts) design problems. An extremely important goal for our hardware-software co-design of operating system services is to create an HAL part that will be portable on several platforms and for each platform/configurable architecture couple.

4.2 Conclusion

Advances in technology provide new commercial simulation platform that combine a general purpose CPUs, buses, memories. These new environments are a significant step toward fast validation of MPSoC design. We have used one of these simulation platforms (MaxSim) to study and report the HW/SW interfacing in MPSoC design. This study is a stage to understand the hardware/software integration step in MPSoC system design. We have presented an hardware mechanism for processors/threads synchronization for a multithreaded POSIX compliant kernel named Mutek. This operating system co-design approach proves that there are parts of OS which can be implemented in HW. This HW implementation of some OS services, in the context of Hardware Abstraction Layer (HAL) design is a motivation for us to investigate in using FPGA to implement a unified model for all those parts which tightly dependent on the platform architecture (particularly CPU). When complete, this FPGA-Based

unified model will enable these new reconfigurable single-chip platforms to be accessible by a much broader community of system programmers, and provide increases in MPSoC's embedded software performance.

References

[1] http://www.arm.com

[2] http://www.xilinx.com

[3] K. Keutzer, S. Malik, R. Newton, J. Rabaey and A. Sangiovanni-Vincentelli "System Level Design: Orthogonalization of Concerns and Platform-Based Design" Published in IEEE Transactions on Computer-Aided Design of Circuits and Systems, Vol. 19, No. 12, December 2000.

[4] www-asim.lip6.fr/recherche/disydent.

[5] S. Yoo, A.A. Jerraya, "Introduction to Hardware Abstraction Layers for SoC", pp. 179 - 186, chapter in "Embedded Software for SoC", Ed. by A.A. Jerraya, S. Yoo, D.Verkest, N.Wehn, Kluwer Academic publishers, October 2003.

[6] Andrews, D.L., Neihaus, D., Ashenden, P. " Programming Models for Hybrid FPGA/CPU Computational Components:", IEEE Computer, January 2004.

[7] B.O. Gallmeister and C. Lanier. "Early Experience with POSIX 1003.4 and POSIX 1003.4a". Proceedings of the IEEE Real-Time Systems Symposium, December 1991, pp. 190-198.

[8] T. Rissa and J. Niittylahti, "A hybrid prototyping platform for dynamicallyreconfigurable designs," in Proc. 10th Int. Workshop on Field-Programmable Logic and Applications, Aug. 2000, pp. 371–378.

Jaehwan Lee, Vincent john Mooney "A Comparison of the RTU Hardware RTOS with a Hardware/Software RTOS". Proc ASPDAC 2003.