A Comparison of the RTU Hardware RTOS with a Hardware/Software RTOS

Jaehwan Lee and Vincent John Mooney III

School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia, U.S.A.
Tel: 404-894-0966
e-mail: {jaehwan, mooney}@ece.gatech.edu

Anders Daleby, Karl Ingström, Tommy Klevin* and Lennart Lindh*

Computer Architecture Laboratory
Mälardalen University (also RealFast*)
Västerås, Sweden
Tel: +46-21-10-1452
e-mail: {ady99002, kim99001, klevin, llh}@mdh.se

Abstract – In this paper, we show the performance comparison and analysis result among three RTOSes: the Real-Time Unit (RTU) hardware RTOS, the pure software Atalanta RTOS and a hardware/software RTOS composed of part of Atalanta interfaced to the System-on-a-Chip Lock Cache (SoCLC) hardware. We also present our RTOS configuration framework that can automatically configure these three RTOSes. The average-case simulation result of a database application example on a three-processor system running thirty tasks with RTU and the same system with SoCLC showed 36% and 19% overall speedups, respectively, as compared to the pure software RTOS system.

I. Introduction

Trends in chip design show rapidly increasing levels of integration. Processors, custom hardware, reconfigurable logic and memory can all be integrated onto a single system-on-a-chip (SoC). Part of future chip designs will certainly include a significant percentage of high performance processors. Another significant portion will be custom application specific integrated circuits (ASICs). However, chips like the Virtex-II Pro from Xilinx [1] which integrate reconfigurable logic and custom processor logic will likely make up an increasing percentage of chip volumes as time progresses.

To fully exploit such chips, we started implementing a framework, which we call δ Hardware/Software RTOS Generation Framework, to help the developers of an SoC codesign both the SoC architecture and the SoC's RTOS at the same time in the beginning of the design phase [2]. The fundamental goal of the δ Framework is to allow the user to more easily configure and explore various RTOS combinations from RTOS components available in a library. Previous work has shown the usefulness of a hardware/software RTOS for SoC [2, 3]. Furthermore, even earlier work has shown the advantage of implementing an RTOS in hardware [4, 5, 6].

In this paper, we present a performance comparison and analysis among three configured systems: (i) a three-processor system with only a pure software RTOS, (ii) a three-processor system with System-on-a-Chip Lock Cache (SoCLC, hardware-supported semaphores [3]), and (iii) a three-processor system with Real-Time Unit (RTU, [4]). We also describe the integration of the RTU into the δ Framework.

This paper is organized as follows: Section II discusses previous work related to this research. Section III gives our motivation and Section IV describes detailed interfaces needed for the implementation. Section V describes the δ Framework which can be used to configure a target system architecture with or without specialized hardware in the RTOS. In Section VI, we describe an application example, three configurations generated from the framework and our simulation environment. In Section VII, we present the performance comparison results of three configured systems and analyze thoroughly where the performance difference comes from so that the user may be able to use the result of the analysis for RTOS design space exploration and also to decide which configuration is most suitable for his or her application(s). Finally, conclusion and future work are addressed in Section VIII.

II. Previous Work

Previously, a hardware RTOS unit called RTU was developed as shown in Figure 1 [4]. The RTU is a hardware operating system that moves the scheduling, inter-process communication (IPC) such as semaphores as well as time management control such as time ticks and delays from the software OS-kernel to hardware. The RTU decreases the system overhead and can improve predictability and response time by an order of magnitude. (This increased performance is due to the reduced CPU load when the RTOS functionality is placed into hardware.) The RTU also dramatically reduces the memory footprint of the OS to consist of only a driver that communicates with the RTU. Thus, less cache misses are seen when the RTU RTOS is used. The RTU also supports task and semaphore creation and deletion, even dynamically at run time.

Another previous work is the Sysem-on-a-Chip Lock Cache (SoCLC, shown in Figure 2) which was introduced as a hardware support to accelerate software locks and semaphores in a software RTOS [3]. Lock variables including binary semaphore functionality are moved into a separate "lock cache" outside of the memory system but in the SoC, thereby improving performance due to reduced delay in accessing a lock variable and reduced bus contention in a shared memory multi-processor SoC.

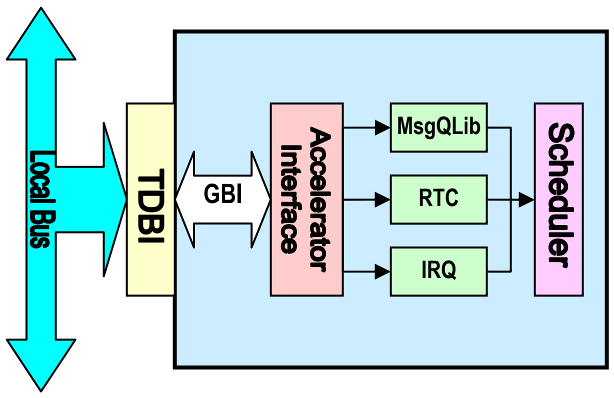
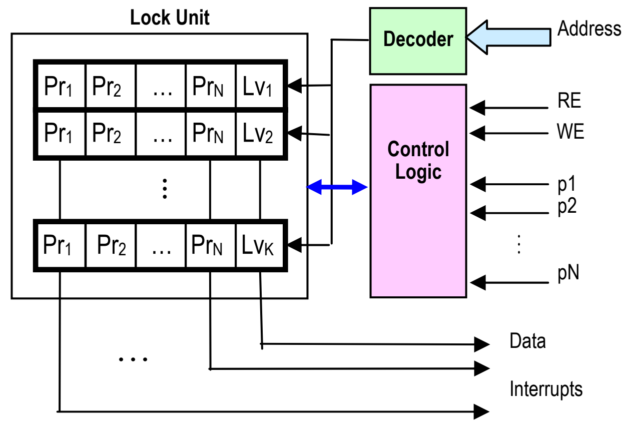


Figure 1. The Real-Time Unit (RTU)


Lock Unit
Pr_1 Pr_2 … Pr_N Lv_1
Pr_1 Pr_2 … Pr_N Lv_2
Pr_1 Pr_2 … Pr_N Lv_K
Decoder
Address
Control Logic
RE
WE
p1
p2
pN
Data
Interrupts

Lv_k : Lock variable $_k$ (k : 1 to K, K : number of locks)
Pr_i : Processor $_i$ (i : 1 to N, N : number of processors)
RE : Read Enable
WE : Write Enable
p1, p2,…,pN: signals designating transactions from PE_1, PE_2,…,PE_N, respectively

Figure 2. The System-on-a-Chip Lock Cache (SoCLC)

Also recently, some research efforts in addition to ours have paid much attention to the field of automatic generation of application specific operating system and architecture [7, 8].

III. Motivation

Suppose an SoC architecture such as Figure 3 is designed and simulated with a pure software RTOS and is found to not meet the timing constraints of applications to be run on the SoC. One reaction could be to wish to change to a larger SoC with more processors. However, suppose we want to investigate the use of logic gates to speed up the application because, for example, we are in a post-fabrication scenario and no larger SoC exists! In this case, we can explore different ways of using the reconfigurable logic to speed up RTOS functionality. Thus, we introduced δ SoC/RTOS Codesign Framework which is designed to provide automatic hardware/software RTOS configurability to support user-directed hardware/

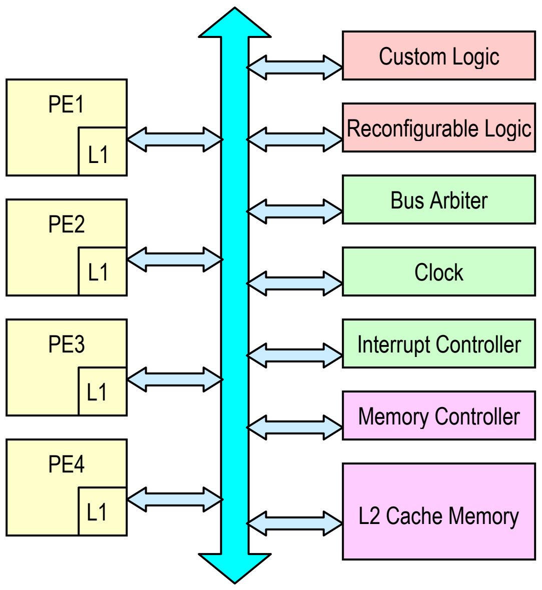


Figure 3. Target SoC architecture (PE: processing element)

software co-design [9, 10]. We previously focused on the configuration of an RTOS which may include parts of the RTOS in hardware. On top of this initial framework, we have included the generation of header files to use the RTU.

In this paper, we compare and contrast the previous two solutions (the RTU hardware RTOS and the SoCLC hardware lock cache) with each other and with a pure software solution. Furthermore, we present an upgraded framework able to configure the SoC for all three RTOS configurations: RTU, SoCLC and pure software RTOS.

IV. Implementation

In this section, we describe detailed interfaces needed for the automatic configuration of the RTOS chosen for the SoC under consideration. As shown in Figure 1, the RTU is partitioned into functional units: Accelerator Interface, Scheduler unit, MsgQLib (which handles message passing, semaphores and time-related scheduling such as task delay), IRQ – intelligent interrupt handler – and RTC – real-time control. The interface to the RTU is divided into a generic bus interface (GBI) and a technology dependent bus interface (TDBI). The GBI is independent of external bus while TDBI is dependent on the bus architecture in the system. Thus, this design of RTU makes it easy to interface it towards different busses. All communication (service calls) with the RTU is carried out through a set of registers located in the Accelerator Interface. A service call is decoded in the Accelerator Interface and routed out to the unit that will carry out the supported service call. In order to fully accommodate the RTU, the system needs RTU application programming interfaces (APIs) to handle software hand-shaking between the CPU and the RTU.

The SoCLC, as shown in Figure 2, contains lock variables and Pr (which stands for processor) bit locations associated with each lock variable. The Pr bit locations are kept for every processor and indicate whether or not the processor is waiting for the lock. The waiting processors are sent an interrupt to signal the processor when the lock is released. Also, the SoCLC reports which lock has been released so that the RTOS can search for the correct task corresponding to this lock.

For a pure software RTOS, we use Atalanta RTOS version 0.4 [11], a shared-memory heterogeneous multi-processor RTOS. The Atalanta RTOS is similar in functionality to other small RTOSes such as µC/OS-II [12] or VRTX RTOS [13].

V. Framework

In this section, we briefly describe the δ Framework (please see [2] for more information), its main idea and the environment which makes the δ Framework possible. The flow of automatic generation of configuration files in the δ Framework is shown in Figure 4. From the description library of (i) the SoC architecture, (ii) the Atalanta software RTOS, (iii) the RTU hardware RTOS and (iv) other hardware RTOS components, the δ Framework generates configuration files for a hardware/ software RTOS for SoC directed by the user through the graphical user interface (GUI) tool (written in Tcl&Tk [14]). The configuration files are header files for C pre-processing, makefiles and a hardware description language (HDL) top file to glue the system together. To verify generated configurations from the tool, the HDL top file is compiled, the application(s) and the configured software RTOS are compiled and linked by the GNU gcc compiler for PowerPC. As a next step, to judge performance, we execute various configured RTOS code and SoC hardware in the Mentor Graphics Seamless Co-Verification Environment (CVE) [15] with the Modelsim mixed-VHDL/Verilog simulator. Within the Seamless framework, Processor Support Packages (PSPs) and Instruction Set Simulators (ISSes) for processors, e.g., ARM920T and PowerPC MPC755, are provided.

The information in the libraries shown in Figure 4 includes signals and parameters of the IP (intellectual property) library of the RTU hardware RTOS and other hardware components such as System-on-a-Chip Lock Cache (SoCLC [3]), System-on-a-Chip Deadlock Detection Unit (SoCDDU [16]) and System-on-a-Chip Dynamic Memory Management Unit (SoCDMMU [17]).

Because we precisely described the GUI and the generation and linking process of Makefile and C header files in a previous paper [2], in this section we only address the generation process of an HDL top file and focus on presenting how the RTU can be integrated into the desired system.

Since we use Modelsim as a mixed VHDL/ Verilog simulator, all modules and IP modules (either written in VHDL or written in Verilog) that can possibly be integrated into the target architecture are each pre-compiled and stored to a directory of its own name. Moreover, the tool does not have to extract each module from an HDL library to an HDL file described in the previous paper [2]. Therefore, our tool only needs to manipulate a top-level HDL file so that the configuration and generation process becomes much simplified. Also, a processing element (PE) IP core is wrapped as an independent module so that it can be instantiated multiple times without each instance interfering each other. This rule is applied to all other modules possibly needing to be instantiated multiple times. Therefore, scalability is ensured.

Here, we take a system utilizing the RTU, shown in Figure 5, as an example target to describe the hardware configuration process. The RTU system of Figure 5 consists of more than ten modules such as a clock generator, three MPC755s, a memory controller, L2 memory, a bus arbiter, an interrupt controller and an RTU. We store the module information in the hardware system

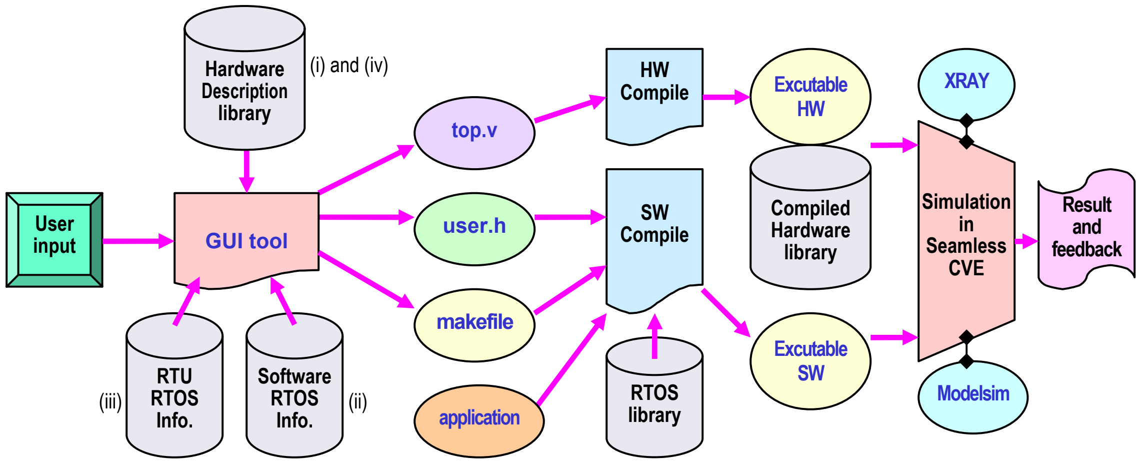


Figure 4. δ Framework for customized hardware/software RTOS design

description library shown in Figure 4. Also, the RTU interface information and Atalanta software RTOS information are stored in separate files. If the user selects the RTU in order for the system to be as fast as possible and most easily proven to be deterministic, then the GUI tool executes the Archi_gen program with the arguments specified by the user. Archi_gen generates an application specific HDL top file utilizing the RTU. The final output of the configured SoC is a Verilog top module, which contains all module instantiation code for PE wrappers, memory, a memory controller, bus, a bus arbiter and an RTU.

VI. Experimental Setup

In this section, we describe our database application example, three configured systems and experimental setup. As an application, we implemented a database system model which has many different task level synchronization scenarios [18]. As illustrated in Figure 6, a database system may have several transactions. Each task must acquire a lock before initiating a transaction in order for the transaction to be atomic. A transaction is a process of accessing a database object (labeled O_i in Figure 6), which is equivalent to a critical section (CS) in our example. In Figure 6, any thick or dotted arrow labeled "Ti" indicates an object copying action from source to destination, which is a job assigned to each task.

We have used δ Framework to generate three hardware/software RTOS configurations for three systems. Here, a "system" is defined as an SoC architecture with an associated RTOS. We also define a "base architecture" as an SoC architecture that contains only essential hardware components in an SoC (needed for almost all systems) such as PE wrappers with PE instantiation description, an arbiter, an address decoder, a memory and a clock. Each SoC architecture we simulated contained three MPC755s and additional hardware modules (as shown in Figure 5 for the case of placing the RTU in the on-chip reconfigurable logic). The clock speed of each system is 125 MHz (8 ns clock period), and the size of the L2 memory is 16 Mbytes. The MPC755 has separate instruction and data caches each of size 32KB.

The first configuration for the system illustrated in Figure 7 uses only the Base architecture with a pure software RTOS, and, therefore, the synchronization is performed with software semaphores and spin-locks in the system. Note that in this first system, all of the reconfigurable logic is available for other uses as needed. For comparison with the first system, we generated a second configuration for the system, illustrated in Figure 8, which utilizes SoCLC for the synchronization in the same database example. The third configuration includes the RTU as shown in Figure 5, exploiting the reconfigurable logic for scheduling, synchronization and even time-related services. In short, in the system shown in Figure 5, the hardware RTU handles most of the RTOS services.

Simulations of two database examples were carried out on each of these three systems using Seamless CVE [15], as illustrated on the far right-hand side of Figure 4. We used Modelsim from Mentor Graphics for mixed VHDL/Verilog simulation and XRAY^{TM} debugger from Mentor Graphics for application code debugging. To simulate each configured system, both the software part including application and the hardware part of the Verilog top module were compiled. Then the executable application and the multi-processor hardware setup consisting of three MPC755's were connected in Seamless CVE.

VII. Experimental Results

Experimental results in Table 1 present the total execution time of (i) simulation with software semaphores, (ii) simulation with SoCLC (hardware-supported semaphores) and (iii) simulation with RTU. As seen in the table, the RTU system achieves a 50% speedup over case (i) in the total execution time of the 6-task database application. On the other hand, the SoCLC system showed a 41% speedup over case (i). We also simulated these systems with a 30-task database application, where the RTU system and the SoCLC system showed 36% and 19% speedups, respectively, compared to the pure software

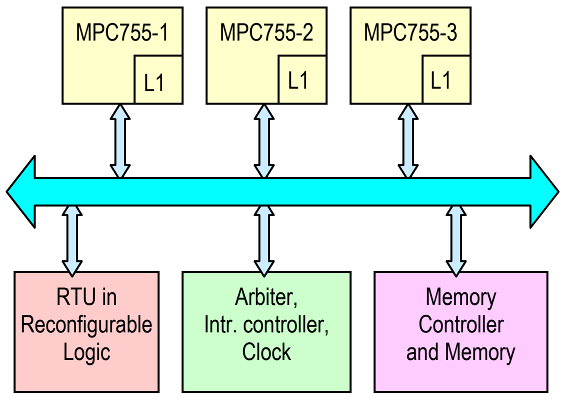

MPC755-1
L1
MPC755-2
L1
MPC755-3
L1
RTU in Reconfigurable Logic
Arbiter, Intr. controller, Clock
Memory Controller and Memory

Figure 5. An SoC architecture with RTU

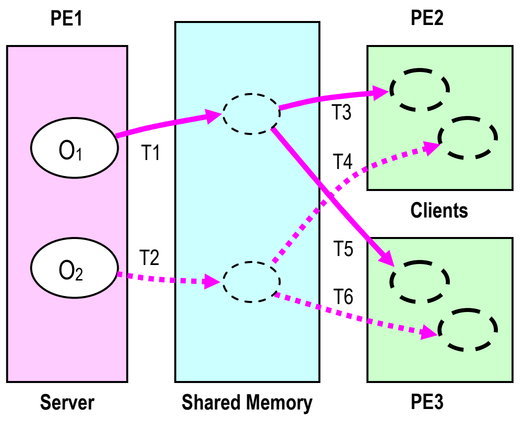


Figure 6. A database transaction application

RTOS system of case (i). The reason why smaller execution time reductions are seen when comparing to the pure software system in the 30-task case is that, when compared to the 6-task case, software for the 30-task case was able to take much more advantage of the caches.

In order to gain some insight to explain the performance differences observed, we looked in more detail at the different scenarios and RTOS interactions. Table 2 shows the total number of interactions including semaphore interactions and context switches while executing the applications. Table 3 shows in which of three broad areas – communication using the bus, context switching and computation – PEs have spent their clock cycles. The numbers for communication shown in Table 3 indicate the time period between just before posting a semaphore and just after acquiring the semaphore in a task that waits the semaphore and has the highest priority for the semaphore. For example, if Task 1 in PE1 releases a semaphore for Task 2 in PE2 (which is waiting for the semaphore), the communication time period would be between just before a post_semaphore statement (semaphore release call) of Task 1 in PE1 and just after a pend_semaphore statement (semaphore acquisition call) of Task 2 in PE2. In the similar way, the numbers for context switch were measured. The time spent on communication in the pure software RTOS case is prominent because the pure software RTOS does not have any hardware notifying mechanism for semaphores, while the RTU and the SoCLC system exploit an interrupt notification mechanism for semaphores.

We also noted that the context switch time when using the RTU is not much less than others. To explain why, recall that a context switch consists of three steps: (i) pushing all PE registers to the current task stack, (ii) selecting (scheduling) the next task to be run, and (iii) popping all PE registers from the stack of the next task. While Step (ii) can be done by hardware, Steps (i) and (iii) cannot be done by hardware in general PEs because all registers inside a PE must be stored to or restored from the memory by the PE itself. That is why the context switch time of the RTU cannot be reduced significantly (as evidenced in Table 3).

We synthesized and measured the hardware area of the SoCLC and RTU with TSMC 0.25um standard cell library from LEDA [19]. The number of total gates for an SoCLC with 64 short CS locks and 64 long CS locks was 7435 and the number of total gates for the RTU was approximately 250,000 as shown in Table 4.

In conclusion, from the information about (i) the size of a specific hardware RTOS component, (ii) the simulation results and (iii) available reconfigurable logic, the user can choose which configuration is most suitable for his or her application or set of applications.

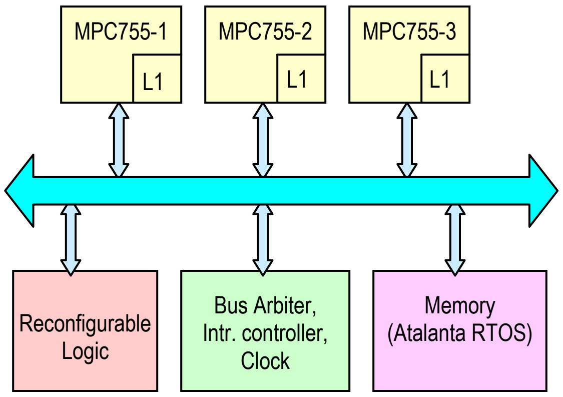


Figure 7. An SoC architecture with no hardware RTOS components

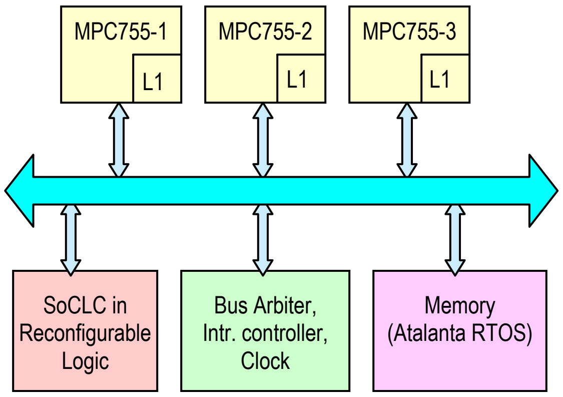

MPC755-1
L1
MPC755-2
L1
MPC755-3
L1
SoCLC in Reconfigurable Logic
Bus Arbiter, Intr. controller, Clock
Memory (Atalanta RTOS)

Figure 8. An SoC architecture with SoCLC

Table 1. Average-case simulation results of the examples

Total Execution Time		Pure SW*	With SoCLC	With RTU
6 tasks	(in cycles)	100398	71365	67038
	Speedup	0%	41%	50%
30 tasks	(in cycles)	379440	317916	279480
	Speedup	0%	19%	36%

* Semaphores are used in pure software while a hardware mechanism is used in SoCLC and RTU.

Table 2. Number of interactions

Times	6 tasks	30 tasks
Number of semaphore interactions	12	60
Number of context switches	3	30
Number of short locks	10	58

Table 3. Average time spent on (6 task case)

cycles	Pure SW	With SoCLC	With RTU
communication	18944	3730	2075
context switch	3218	3231	2835
computation	8523	8577	8421

VIII. Conclusion

In this paper, we presented the average-case simulation result of two database application examples, an example of two client-server triad of tasks and an example of ten client-server triad of tasks. We simulated the total execution time of the examples on (i) a three-processor system with only a software synchronization implementation, (ii) a three-processor system with System-on-a-chip Lock Cache (SoCLC, hardware-supported semaphores [3]) and (iii) a three-processor system with the RTU. As seen in Table 1, the RTU system achieves a 50% speedup over case (i) in the total execution time of the 6-task database application. On the other hand, the SoCLC system showed a 41% speedup over case (i). In case of a 30-task database application, the RTU system and the SoCLC system showed 36% and 19% speedups, respectively, compared to the pure software RTOS system of case (i).

We showed the total number of interactions including semaphore interactions and context switches while executing the applications. We also presented in which of three broad areas – communication using the bus, context switching and computation – PEs have spent their clock cycles. The time spent on communication in the pure software RTOS case took almost an order of magnitude longer than the others because the pure software RTOS does not have any hardware notifying mechanism for semaphores between PEs. The context switching time of the RTU case is not much less than the others because of the time spent in storing and restoring all registers of the PE, which cannot be done by hardware because all registers inside a PE must be stored to or restored from the memory by the PE itself.

We also showed automatic generation of configuration files for a multi-processor system with the RTU hardware RTOS. To compare this system having an RTU with others, we configured two more multi-processor systems with or without the SoCLC hardware RTOS component with the aid of the δ Framework. To measure the performance difference among the configured systems, we compiled and simulated them with two examples within the Seamless Co-Verification Environment [15].

We also showed that, with the δ Framework, the user may be able to use the result of the analysis for RTOS design space exploration and also to decide which configuration is most suitable for his or her application or set of applications. Thus, the δ Framework helps evaluate different SoC/RTOS architectures that employ a hardware RTOS, hardware/software RTOS or a pure software RTOS. In other words, the δ Framework helps in user-directed hardware/software codesign [9, 10].

We plan to extend our research to the configuration of heterogeneous multi-processor systems each with a custom hardware/software RTOS.

Table 4. Hardware area in total gates

Total area	SoCLC (64 short CS locks + 64 long CS locks)	RTU for 3 processors
TSMC 0.25μm library from LEDA	7435 gates	About 250000 gates

Acknowledgements

The two researchers in the U.S. were funded by NSF under INT-9973120, CCR-9984808 and CCR-0082164. Funding for the Swedish researchers was provided by the KK-foundation and ABB. The U.S. researchers also acknowledge donations received from Denali, Hewlett-Packard, Intel, LEDA, Mentor Graphics, National Semiconductor, Sun and Synopsys.

References

[1] Xilinx, http://www.xilinx.com.

[2] J. Lee, K. Ryu and V. Mooney, "A Framework for Automatic Generation of Configuration Files for a Custom Hardware/Software RTOS," *Proc. of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'02),* pp. 31-37, June 2002.

[3] B. S. Akgul, J. Lee and V. Mooney, "System-on-a-Chip processor synchronization hardware unit with task preemption support," *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '01)*, pp.149-157, November 2001.

[4] A. Daleby and K. Ingström, Technical Reference Manual for RTU Operating System Accelerator, Västerås, Sweden, 2002.

[5] RealFast, http://www.realfast.se.

[6] T. Nakano, Y. Komatsudaira, A. Shiomi and M. Imai, "VLSI Implementation of a Real-time Operating System," *Proc. of ASP-DAC '97*, pp. 679-680, January 1997.

[7] L. Gauthier, S. Yoo and A. Jerraya, "Automatic generation and targeting of application-specific operating systems and embedded systems software," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(11), pp.1293-1301, November 2001.

[8] D. Lyonnard, S. Yoo, A. Baghdadi and A. Jerraya, "Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip," *38th Design Automation Conference (DAC 2001)*, June 2001.

[9] G. D. Micheli and M. Sami, editors, *Hardware/Software Co-Design*, Kluwer Academic Publishers, Norwell, MA, 1996.

[10] R. K. Gupta, *Co-synthesis of Hardware and Software for Digital Embedded Systems*, Kluwer Academic Publishers, Boston, MA, 1995.

[11] D. Sun et al., *Atalanta: A new multiprocessor RTOS kernel for System-on-a-Chip Applications*, Technical Report GIT-CC-02-19, http://www.cc.gatech.edu/pubs.html, Atlanta, GA, 2002.

[12] μC/OS-II, http://www.ucos-ii.com/

[13] VRTX RTOS, http://www.mentor.com/embedded/vrtxos/

[14] J. Ousterhout, Tcl/Tk, http://home.pacbell.net/ouster/.

[15] Mentor Graphics, Hardware/Software Co-Verification: Seamless, http://www.mentor.com/seamless/.

[16] P. H. Shiu, Y. Tan and V. Mooney, "A novel parallel deadlock detection algorithm and architecture," *9th International Workshop on Hardware/Software Co-Design (CODES '01)*, pp.30-36, April 2001.

[17] M. Shalan and V. Mooney, "A dynamic memory management unit for embedded real-time System-on-a-Chip," *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '00),* November 2000, pp. 180-186.

[18] M. A. Olson, "Selecting and implementing an embedded database system," *IEEE Computer*, pp.27-34, September 2000.

[19] LEDA Systems, inc. http://www.ledasys.com/