A Framework for Automatic Generation of Configuration Files for a Custom Hardware/Software RTOS

Jaehwan Lee, Kyeong Keol Ryu and Vincent John Mooney III
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia, U.S.A.

Abstract

In this paper, we propose a framework for user-directed automatic generation of configuration files for a custom hardware/software real-time operating system (RTOS) for System-on-a-Chip. The main goal of this research is to help the user explore which configuration is most suitable for his or her specific application or set of applications. This work leverages three previous innovations in hardware/software RTOS design: System-on-a-Chip Lock Cache (SoCLC) [1], System-on-a-Chip Deadlock Detection Unit (SoCDDU) [2] and System-on-a-Chip Dynamic Memory Management Unit (SoCDMMU) [3]. However, in spite of the excellent performance of these innovations, the user may not need or have the chip space for all three of them. In this case, our framework enables automatic generation of different mixes of hardware and software versions of the SoCLC, SoCDDU and SoCDMMU. We show how to generate configuration files for custom RTOSes for two examples published previously [1, 2]. The average-case simulation result of a database application model with client-server pair of tasks on a four-processor system with SoCLC showed a 27% overall speedup in total execution time than that of the same system but with software synchronization. In the other example, the application execution time to end up in a deadlock state and then detect the deadlock showed a 38% speedup with the SoCDDU as compared to the same system but with a software deadlock detection implementation.

Keywords

Reconfigurable Logic, System-on-a-Chip, RTOS, Embedded Systems, Hardware/Software Partitioning

1. Introduction

A System-on-a-Chip (SoC) architecture with reconfigurable logic and multiple processing elements (PEs), as shown in Figure 1, is likely to become quite common in the near future [4]. Nowadays, programmable logic companies are producing reconfigurable logic chips with millions of reconfigurable logic gate equivalents plus, in some cases, custom layout(s) of processor(s), all available in one chip. These chips may be preferable in embedded systems especially where rapid upgrades are needed or where companies want to take advantage of early time-to-market. These chips may also be useful for system development, where the execution of certain software does not meet the timing constraints of some applications, thereby requiring hardware/ software partitioning tradeoffs. Obviously, one possible solution is moving the slow software processing to faster hardware processing in re-configurable logic. However, changing the target architecture and associated software can require significant re-porting/reconfiguration of the real-time operating system (RTOS), which is where our research fits in.

Our framework is designed to provide automatic hardware/software configurability to support user-directed hardware/software partitioning [5, 6]. To ease the user effort, we made a graphical user interface (GUI) tool with which the user can select necessary RTOS components that are most suitable for his or her application. Here, components of an RTOS are specific function modules that are implemented either in software or in hardware. We focus on real-time systems for our specific application domain. Therefore, we assume that the user knows what kind of applications

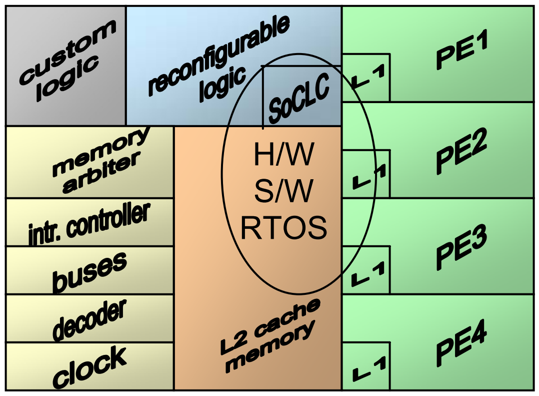


Figure 1. Target SoC Architecture

will be run, how many tasks are in the system, how many methods of inter-process communication (IPC) are used and what the target system is. We also focus our research on a shared memory multiprocessor system, where a shared memory is used for the IPC medium.

The paper is organized as follows: Section 2 discusses the background and motivation of our research. Section 3 gives an overview, requirements and underlying environment of our methodology, describing our target system architecture and RTOS. Section 4 describes our framework in detail and explains how the configuration files can be generated and how these files work together to construct a custom hardware/software RTOS. Section 5 shows the experimental results with two application examples. Finally, conclusion and future work are addressed in Section 6.

2. Background and Motivation

In general, commercial RTOSes available for popular embedded processors provide significant reduction in design time. However, they have to be general and might not be efficient enough for specific applications. To overcome this disadvantage, some previous work has been done in the area of automatic RTOS generation [7, 8]. Using these proposed methodologies, the user can take advantage of many benefits such as a smaller RTOS for embedded systems, rapid generation of the RTOS, easy configuration of the RTOS and a more efficient and faster RTOS due to smaller size than commercial RTOSes. Also, some previous work about automated design of SoC architectures has been done [9, 10]. However, the previous work mainly focuses on one side or the other of automatic generation: either software or hardware. In the methodology proposed in this paper, we focus on the configuration of an RTOS which may include parts of the RTOS in hardware.

3. Methodology

In this section, we propose a novel approach for automating the partitioning of a hardware/software RTOS between a few pre-designed partitions. The flow of automatic generation of configuration files is shown in Figure 2. Specifically, our framework, given the intellectual property (IP) library of processors and RTOS components, translates the user choices into a hardware/software RTOS for SoC. The GUI tool generates configuration files: header files for C pre-processing, a Makefile and some domain specific code files such as Verilog files to glue the system together. To test our tool, we execute our different RTOS configurations in the Mentor Graphics Seamless Co-Verification Environment (CVE) [11]. Within the Seamless framework, Processor Support Packages (PSPs) and Instruction Set Simulators (ISSes) for processors, e.g., ARM920T and PowerPC MPC750, are provided.

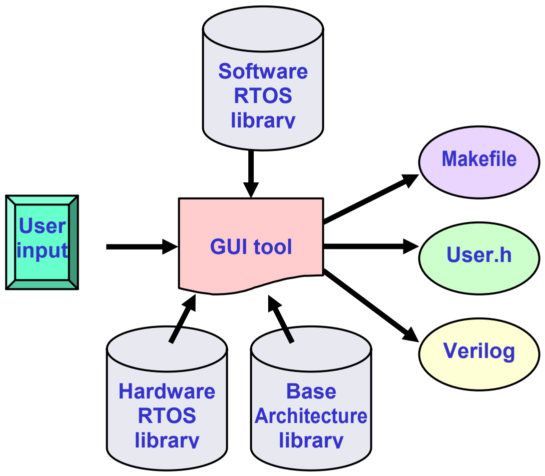


Figure 2. Flow of automatic generation of configuration files

For RTOS hardware IP, we start with an IP library of hardware components consisting of System-on-a-Chip Lock Cache (SoCLC) and System-on-a-Chip Deadlock Detection Unit (SoCDDU) [1, 2]. For the sake of the reader's understanding, we briefly describe these IP components. The SoCLC is a hardware mechanism that resolves the critical section interactions among PEs [1]. Lock variables are moved into a separate "lock cache" outside of the cache memory system but in the SoC, thereby improving the performance criteria in terms of lock latency, lock delay and bandwidth consumption in a shared memory multi-processor SoC. Since each lock variable requires only one bit, the hardware cost is very low. Table 3 (see the last page of this paper) reports an example where 128 lock variables (which are enough for many real-time applications) cost approximately 7000 logic gates of area. The SoCDDU performs a novel parallel hardware deadlock detection based on implementing deadlock searches on the resource allocation graph in hardware [2]. It provides a very fast and very low area way of checking deadlock at run-time with dedicated hardware. The SoCDDU reduces deadlock detection time by 99% as compared to software.

For RTOS software IP, we start with the Atalanta RTOS version 0.3 [12], a shared-memory multiprocessor real-time operating system developed at the Georgia Institute of Technology. This RTOS is specifically designed for supporting multi-processors with large shared memory in which the RTOS is located and is similar to many other small RTOSes. All PEs (currently supported are either all MPC750 processors or all ARM9 processors) execute the same RTOS code and share kernel structures, data and states of all tasks. Each PE, however, runs its own task(s) designated by the user. Almost all software modules are pre-compiled and stored into the Atalanta library. However, some modules have to be linked to the final executable file from the object module itself because some function names have to be the same in order to provide the same application programming interface (API) to the user. For example, if the deadlock detection function could be implemented in the RTOS either in software or in hardware, then the function name that is called by an application should be the same even though the application could use, depending on the particular RTOS instantiated in the SoC, either the software deadlock detection function or a device driver for the SoCDDU. By having the same API, the user application does not need modification whichever method – either software or hardware – is actually implemented in the RTOS and the SoC.

To ease the user effort required to manually configure the hardware/software RTOS, we made a GUI tool for user inputs. With the GUI tool, the user can select necessary RTOS components that are most suitable for his application. Currently, the user may select the following items in software: IPC methods such as semaphores, mailboxes and queues; schedulers such as priority or round-robin scheduler; and/or a deadlock detection module. The user may also select the following items in hardware: SoCDDU and/or SoCLC.

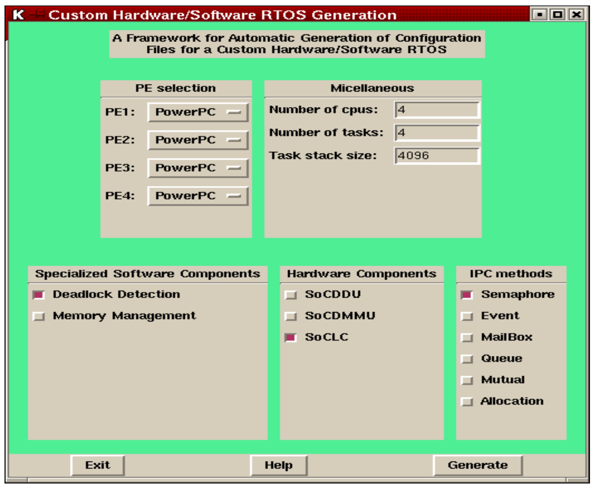
We have specified two important requirements for implementing our tool. The first requirement is that the RTOS configuration process should be easily scalable according to the number of PEs and IPC methods. One example of scalability is that if the user selects the number of PEs, the tool can adaptively generate an appropriate configuration that contains the given number of PE wrappers and the interfaces gluing PEs to the target system. The second requirement is to ensure a high degree of separation of the tool from changes or upgrades of the hardware IP library, which means that our tool is designed so that tool changes caused by hardware module changes in the IP library are minimized.

4. Implementation

In this section, we describe which configuration files are generated from the tool and how these files are used to make a custom hardware/software RTOS.

4.1 The GUI tool for configuration

Figure 3 shows our GUI tool, with which the user can configure an application specific hardware/software RTOS and an SoC for his application. For pre-fabrication design space exploration, different PEs and the number of PEs can be selected. For post-fabrication customization of a platform SoC with reconfigurable logic (e.g., a specific fabricated version of Figure 1), the user can decide whether or not to put parts of the RTOS into the reconfigurable logic. The user can input application specific constraints such as the task stack size and the number of tasks in the system. The source code of the tool is written in the Tcl/Tk language [13].



A Framework for Automatic Generation of Configuration Files for a Custom Hardware/Software RTOS

PE selection
PE1: PowerPC
PE2: PowerPC
PE3: PowerPC
PE4: PowerPC

Micellaneous
Number of cpus: 4
Number of tasks: 4
Task stack size: 4096

Specialized Software Components
[✓ Deadlock Detection]
Memory Management

Hardware Components
SoCDDU
SoCDMMU
[✓ SoCLC]

IPC methods
[✓ Semaphore]
Event
MailBox
Queue
Mutual
Allocation

Figure 3. A GUI tool for a RTOS component selection

4.2 *Makefile* generation example

The generation of a *Makefile* is shown in Example 1.

Example 1: *Makefile* generation. After the input data shown in Figure 3 is entered by the user in the GUI, the user clicks the *Generate* button, and the tool generates a *Makefile* containing assignments saying software deadlock detection object module and a device driver for SoCLC are included.

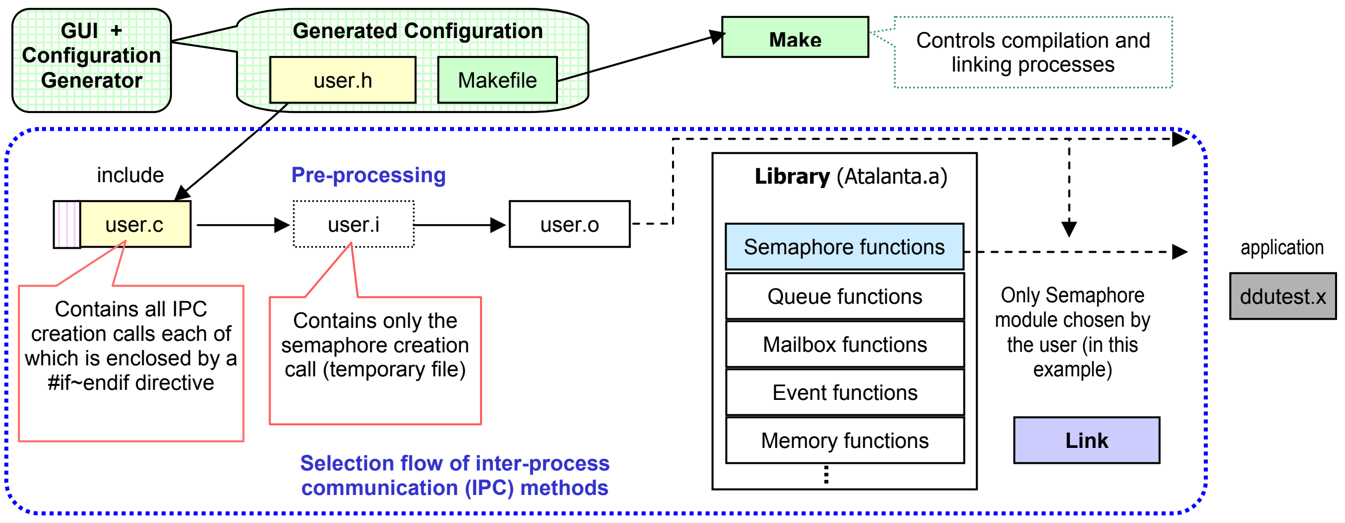
4.3 The linking process of specialized software components for a function with different implementation

To generate a smaller RTOS from the given library, it is necessary that only needed components be included in the final executable file. One of the methods to achieve this is very straightforward. For example, when the user selects the software deadlock detection component, then the tool generates a *Makefile* that includes the software deadlock detection object. On the contrary, when the user selects the hardware deadlock detection unit, then the tool generates a different *Makefile* that includes only the software device driver object containing APIs that manipulate the hardware deadlock detection unit. Therefore, the final executable file will have either the software module or the device driver module.

4.4 IPC module linking process

On the other hand, when the user selects IPC methods, the inclusion process is more complicated than the linking process of specialized software components. IPC modules that implement IPC methods (such as queue, mailbox, semaphore and event) are provided as separate files. The linking process for IPC modules is shown in Figure 4. From the GUI of the tool, the user can choose one or more IPC methods according to his

Figure 4. Example of the IPC module linking process



GUI + Configuration Generator
Generated Configuration
user.h
Makefile
Make
Controls compilation and linking processes
include
Pre-processing
user.c
user.i
user.o
Contains all IPC creation calls each of which is enclosed by a #if~endif directive
Contains only the semaphore creation call (temporary file)
Library (Atalanta.a)
Semaphore functions
Queue functions
Mailbox functions
Event functions
Memory functions
Only Semaphore module chosen by the user (in this example)
Link
application
ddutest.x
Selection flow of inter-process communication (IPC) methods

application requirements. For example, as shown in Figure 3, when the user selects only the semaphore component among IPC methods, the tool generates a *user.h* file which has a semaphore definition and is used by the C pre-processor. Without automation, *user.h* must be written by hand. This *user.h* file will then be included into *user.c,* which contains all the call routines of IPC creation functions. Each of the call routines is enclosed by a *#if~#endif* compiler directive and also corresponds to a C source file. During C pre-processing, the compiler will include the semaphore call routine. Thus, during linking, the linker will include the semaphore module from the Atalanta library in the final executable file because the semaphore creation function is needed (the creation and management functions for semaphores are contained in *semaphore.c* in Atalanta [12]).

4.5 Assumption of our simulation framework

As described in the first paragraph of Section 3, we use PSPs from Mentor Graphics; therefore, we only generate PE wrappers instead of IP cores themselves.

4.6 The configuration of the hardware modules of an RTOS

In this section, we describe how a user-selected hardware RTOS component is integrated to the target architecture. The final output of the tool is a Verilog hardware description language (HDL) header file that contains hardware IP modules such as PE wrapper, memory, bus, SoCLC and/or SoCDDU. Here, we define the "Base" system as an SoC architecture that contains only essential components (needed for almost all systems) such as PE wrappers, an arbiter, an address decoder, a memory and a clock, which are stored in the Base Architecture Library shown in Figure 2. Optional hardware RTOS components such as SoCLC and SOCDDU are stored in the Hardware RTOS Library. Optional hardware RTOS components chosen by the user are integrated together with the Base system. The generation of a target SoC Verilog header file (which contains the Base architecture together with configured hardware RTOS components, if any) starts with an architecture description that is a data structure describing one of the target SoC architectures. From the tool, if the user does not select any hardware RTOS components, a Verilog header file containing only the Base system will be generated. On the other hand, if the user wants to use SoCLC to support fast synchronization among PEs (e.g., to meet hard timing deadlines of an application), he can select *SoCLC* in the hardware component selection. Then the tool will automatically generate an application specific hardware file containing SoCLC. This hardware file generation is performed by a C program (to be explained in the next section), which is compiled with a TCL wrapper.

4.7 A HDL file generation example

Throughout this section, we take an SoC system utilizing the SoCLC, shown in Figure 5, as an example target to describe the hardware configuration process. The SoCLC system of Figure 5 consists of more than ten modules such as a clock generator, MPC750s, an address decoder, L2 cache memory, a memory arbiter, interrupt controller and SoCLC. Of course, in this system, a device driver software module is also needed to support SoCLC hardware; such device driver software is included in the final executable file by the aforementioned IPC module linking process described in Section 4.4. Our tool takes the user input via the GUI shown in Figure 3 and uses the hardware IP libraries to generate the code for each module and for the instantiation of all Verilog files in appropriate link files (usually a single "top" module in a "top" Verilog file).

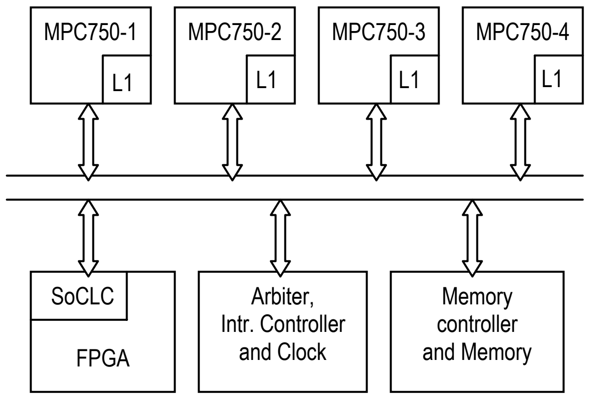

MPC750-1
L1
MPC750-2
L1
MPC750-3
L1
MPC750-4
L1
SoCLC
FPGA
Arbiter, Intr. Controller and Clock
Memory controller and Memory

Figure 5. An SoC architecture with SoCLC

Example 2: Verilog file generation. Here, we describe in more detail the generation process of a top-level Verilog file for the SoCLC system step by step. When the user selects *`SoCLC'* in the tool, as shown in Figure 3, and clicks the *`Generate'* button, the tool calls a hardware generation program, *Archi_gen* (written in C), with architecture parameters. Then, the tool makes a compile script for compiling the generated Verilog file. As a next step, Archi_gen makes the hardware Verilog file as follows (see Figure 6). First, Archi_gen extracts (i) all needed modules according to the SoCLC system description of Figure 5. Second, Archi_gen generates (ii) the code for wiring up the SoCLC system (including all buses). Third, Archi_gen generates (iii, see Figure 6) the instantiation code according to the instantiation type of the hardware modules chosen. In this step, Archi_gen also generates instantiation code for the appropriate number of PEs according to the user selection of the number of PEs in the tool. In our example, Archi_gen generates the code for PE instantiation four times each for one MPC750. Fourth, Archi_gen extracts the initialization code (needed for test) for necessary signals according to the SoCLC initialization description. Finally, a Verilog header file containing an SoCLC system hardware architecture is made.

5. Experimental Results

We have used our framework to generate five

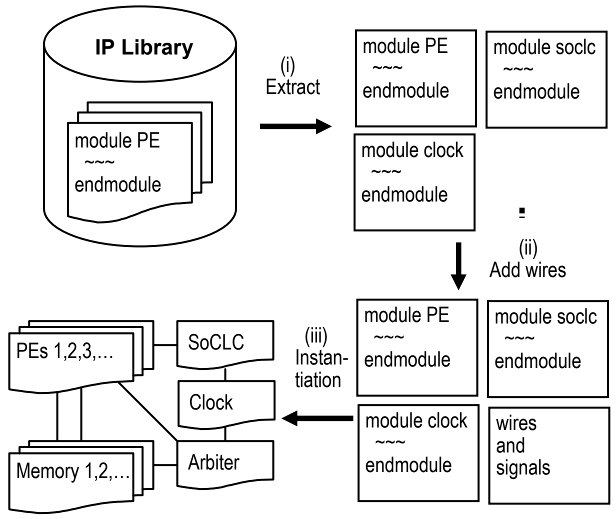

IP Library
module PE
~~~
endmodule
(i)
Extract
module PE
~~~
endmodule
module soclc
~~~
endmodule
module clock
~~~
endmodule
(ii)
Add wires
module PE
~~~
endmodule
module soclc
~~~
endmodule
module clock
~~~
endmodule
wires and signals
(iii)
Instan-
tiation
PEs 1,2,3,...
SoCLC
Clock
Arbiter
Memory 1,2,...

**Figure 6. HDL file generation process**

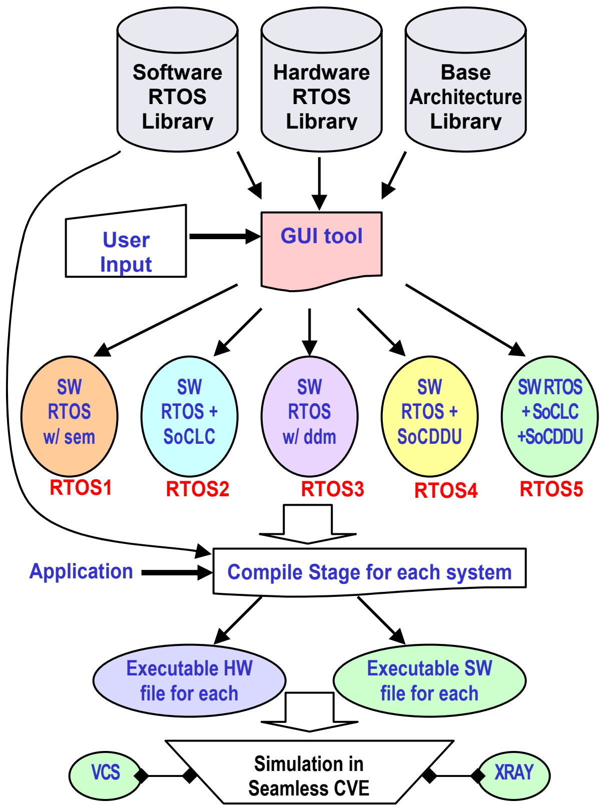


**Figure 7. Five custom hardware/software RTOS instantiations and their verification setup**

hardware/software RTOS configurations for five systems as shown in Figure 7. Here, we use the word "system" to refer to an SoC architecture with an RTOS. Each SoC architecture has four MPC750s and additional hardware modules -- for example, the modules shown in Figure 5. The first configuration in Figure 7, marked as "SW RTOS w/ sem" (with semaphore), is for the system implementing a database transaction example [14] with semaphores and spin-locks. For comparison with the first system, we generated the second configuration, "SW RTOS + SoCLC", which utilizes SoCLC along with the same database example. The third configuration, "SW RTOS w/ ddm" (with a software deadlock detection module), is for the system implementing an example motivated by a Jini lookup service application [15]. The fourth configuration, "SW RTOS + SoCDDU", is for the system utilizing SoCDDU for comparison with the third system. The fifth configuration, "SW RTOS + SoCLC + SoCDDU", is for the system utilizing both SoCLC and SoCDDU.

To make RTOS hardware instantiation scalable and to support user-directed automatic RTOS configuration, we changed the given structure of the Verilog code of the previously implemented systems [1, 2] to a hierarchical structure so that the instantiation process can be done more easily. In a hierarchical structure, a
~~~

hardware IP core is wrapped as an independent module, and it can be instantiated multiple times without interfering one another. Therefore, the scalability (mentioned in Section 1) can be ensured.

Simulations of these five systems, as shown in Figure 7, were carried out using Seamless CVE [11]. We use Synopsys VCS^{TM} for the Verilog simulator and XRAY^{TM} from Mentor Graphics for application code debugging. To simulate the aforementioned examples, both the software part including application and the hardware part of the configured RTOS were compiled. Next, the executable application and the multi-processor hardware setups consisting of four MPC750's were connected in Seamless CVE. The interfacing of the processors with the shared memory and other hardware RTOS components were established through an address decoder, an arbiter and a memory controller unit.

To verify that the generated configurations for RTOS1 and RTOS2 (shown in Figure 7) are correct, we have used the same database example that the authors of the SoCLC implemented previously. The example includes accesses to short critical sections (CSes) as well as long CSes. Long CSes are the actual database object copying actions, whereas the short CSes are the synchronization actions among the server tasks and the client tasks [14].

Experimental results in Table 1 present the lock latency, lock delay and total execution time for two cases: (i) simulation with software semaphores and (ii) simulation with SoCLC (hardware-supported semaphores); both were run with 40 tasks [1]. As seen in the table, the SoCLC mechanism achieves 27% overall speedup in the total execution time of the database example.

Table 1. Average-case simulation results of the database example

	* Without SoCLC	With SoCLC	Speedup
Lock Latency (clock cycles)	1200	908	1.32x
Lock Delay (clock cycles)	47264	23590	2.00x
Execution Time (clock cycles)	36.9M	29M	1.27x

* Semaphores for long CSes and spin-locks for short CSes are used instead of SoCLC.

We used FPGA Express from Synopsys to synthesize the SoCLC in Xilinx XC4000E 4003EPC84 [16]. The total elements for an SoCLC with 64 short CS locks and 64 long CS locks are 532 sequential logics and 9036 other gates (as shown in Table 3). From these results, the user can choose between two tradeoffs: gain a speedup using the SoCLC with chip space overhead versus keep the available reconfigurable logic in the SoC available for other uses and instead use the slower software semaphore synchronization method.

To verify that the generated configurations for RTOS3 and RTOS4, shown in Figure 7, are correct, we consider how the SoCDDU would perform in a real-life example. We devised an example motivated by a Jini lookup service application [15], where client applications can request services through intermediate layers (i.e., lookup, discovery and join). This example has four clients and four services. Here, clients are PEs on which application tasks request resources; the resources are PCI, MPEG, FFT and wireless interface hardware units. The first PE processes video streams. The second PE completes some signal processing algorithms. The third PE handles services such as fax, voice and email. The fourth PE handles communication functions. Because this is a system with multiple PEs and multiple resources, it is possible that a deadlock may occur. Therefore, this is a good example that can exploit the SoCDDU. We devised a sequence of requests and grants that leads to a deadlock at time t5 in Figure 8 (for the exact detailed sequence, please see [2]). The processing times shown in Figure 8 are not from an actual industrial product but instead are estimates intended to exemplify an application with short execution times. We measured the execution time in CPU clock cycles, where the CPU, MPC750, has an 83.3MHz clock, with the SoCDDU on the same clock. We assume that if a PE needs two resources, it cannot proceed in its computation until it acquires both resources. This assumption is valid in typical multimedia applications processing streamed data.

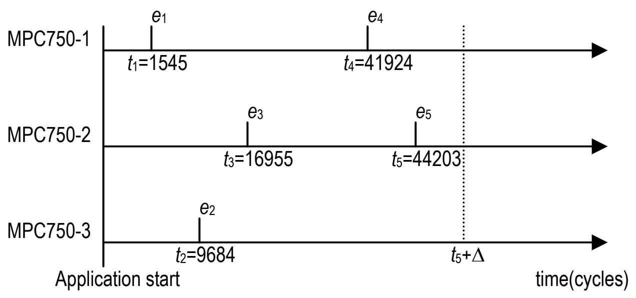


Figure 8. Event sequence

Suppose we start deadlock detection after event *e5* occurs, and the deadlock detection time is Δ. What we are interested in is the values of Δ with different deadlock detection methods. We compare the impact on deadlock detection time and total execution time of both methods: software deadlock detection versus SoCDDU. The result is shown in Table 2. The total execution time is reduced by 16926 clock cycles, a speedup of 38% up to deadlock discovery.

We also synthesized the SoCDDU using FPGA Express targeted to XC4000E 4003EPC84. The total elements of the SoCDDU for our example are 10 sequential logics and 559 other gates. Table 3 summarizes the area overhead of our specific instantiations of the SoCLC and SoCDDU both in terms of a semi-custom VLSI implementation and an implementation in Xilinx reconfigurable logic. Please note that the SoCLC and SoCDDU are scalable with a variable area based on the number of lock variables or the number of requestors/resources [1, 2]. From these results, the user can also consider tradeoffs between the SoCDDU with chip space overhead in reconfigurable logic versus the software deadlock detection method with slower execution time but no hardware area overhead.

Table 2. Deadlock detection time and total execution time

Method of Deadlock Detection	Detection Time Δ (Cycles)	T5+Δ
Software Algorithm	16928	61131
SoCDDU	2	44205

Table 3. Hardware area

Total area in		SoCLC (64 short CS locks + 64 long CS locks)	SoCDDU (5 Processors x 5 Resources)
Semi-custom VLSI		7435 gates using TSMC 0.25μm standard cell library	364 gates using AMI 0.3μm standard cell library
Xilinx XC4000E 4003EPC84	Seq. logics	532	10
	Other gates	9036	559

6. Conclusion

We have presented a framework for user-directed automatic generation of configuration files for a custom hardware/software RTOS for SoC. Our framework is implemented in Tcl/Tk and in C, enabling automatic generation of different mixes of hardware and software versions of the SoCLC [1] and SoCDDU [2] RTOS components. The framework helps the user explore which configuration is most suitable for his specific application or set of applications. This exploration also helps the user examine different SoC architectures prior to fabrication. Our framework is especially useful in post-fabrication customization of an SoC with large amounts of reconfigurable logic where the user wants to efficiently explore the RTOS design space afforded by placing parts of the RTOS in reconfigurable logic.

We have verified the correctness of our framework through the simulation of two examples. We have five different instantiations of a custom hardware/software RTOS. Two of the instantiations were simulated with the example of client-server pair interactions [1]. Two more instantiations were simulated with the example implementing a novel deadlock detection algorithm [2]. The hardware part of the fifth instantiation (with both SoCLC and SoCDDU hardware units) was also simulated for correct operation (but without a full application example). Thus, from our framework, the user can explore RTOS design space by configuring various tradeoffs between software and hardware modules and by simulating different RTOS configurations in a co-design simulator. Therefore, our tool can be considered to be an aid to user-directed hardware/software partitioning [5, 6].

We are currently working on the configuration of a system with SoCDMMU [3]. We would like to extend our research to the configuration of heterogeneous multi-processor systems each with a custom hardware/ software RTOS.

7. Acknowledgements

This research is funded by NSF under INT-9973120, CCR-9984808 and CCR-0082164. We acknowledge donations received from Denali, Hewlett-Packard, Intel, LEDA, Mentor Graphics, SUN and Synopsys.

8. References

[1] 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.

[2] 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.

[3] 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.

[4] The international technology roadmap for semiconductors, edited by Semiconductor Industry Association, November 2001, http://semichips.org.

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

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

[7] F. Balarin, M. Chiodo, A. Jurecska and L. Lavagno, “Automatic generation of real-time operating system for embedded systems,” *Proceedings of the 5th International Workshop on Hardware/Software Co-Design (CODES/CACHE ’97)*, 1997.

[8] 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.

[9] 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.

[10] S. Vercauteren, B. Lin and H. Man, “Constructing application-specific heterogeneous embedded architectures from custom hardware/software applications,” *ACM/IEEE Design Automation Conference*, pp. 521-526, June 1996.

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

[12] D. Sun et al., “Atalanta: A new multiprocessor RTOS kernel for System-on-a-Chip Applications,” GIT-CC-02-19, http://www.cc.gatech.edu/pubs.html.

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

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

[15] S. Morgan, “Jini to the rescue,” *IEEE Spectrum*, 37(4), pp 44-49, April 2000.

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