A Hardware-Software Real-Time Operating System Framework for SoCs

Vincent J. Mooney III and Douglas M. Blough
Georgia Institute of Technology

The δ framework for RTOS-SoC codesign helps designers simultaneously build a SoC or platform-ASIC architecture and a customized hardware-software RTOS. Examples generated by this prototype tool include RTOS designs that speed up applications by 27% or more, using a small amount of hardware area.

A SYSTEM-ON-A-CHIP (SoC) architecture with reconfigurable logic and multiple processing elements sharing a common memory, like that shown in Figure 1, is likely to become quite common in the near future. For example, programmable-logic companies are producing single chips containing millions of reconfigurable logic gate equivalents and full-custom VLSI processors. Designers can customize such chips by reprogramming the reconfigurable logic or the processors. We envision a scenario in which a few highly reconfigurable, reprogrammable SoCs appear in most embedded applications that demand rapid upgrades, fast time to market, and low cost.

However, changing the SoC architecture and associated software can require significant re-porting or reconfiguration of the real-time operating system. Hence, our work focuses on custom configuration of hardware-software RTOSs for SoCs. This article introduces the δ hardware-software RTOS framework, a prototype hardware-software generation tool that aids in customized RTOS-SoC codesign.

Motivation

Commercial RTOSs available for popular embedded processors provide significant reductions in design time. But they typically don't take advantage of hardware to implement any of their functions, probably because processors and custom hardware accelerators have historically resided on separate chips. Therefore, partitioning functionality between hardware and software has increased system speed only when the dramatic speedup provided by hardware compensates for the chip-to-chip communication cost, as is the case for graphics coprocessors. Large communication costs, however, have limited fine-grained hardware-software partitioning. The advent of SoCs has greatly reduced this communication barrier to splitting logic between hardware and software on the same chip. Therefore, we developed the δ framework.

Analogy

Early computer microarchitecture designers did not fully consider the compiler's effect on performance. With the advent of reduced-instruction-set computing, however, almost all computer architects began to include the compiler early in microarchitecture design. Codesign of the compiler and the computer

0740-7475/02/$17.00 © 2002 IEEE

architecture can be considered an example of hardware-software codesign.[1,2]

Many of today's SoC designs are single-chip implementations of prior multichip PCB designs. The SoC designers make only marginal changes of the PCB design; for example, the SoC might use exactly the same bus structure as the PCB. It is certainly not common practice to codesign the SoC and the RTOS that will run the application code. In this respect, much of today's SoC design is analogous to early computer architecture design performed with no consideration of the compiler.

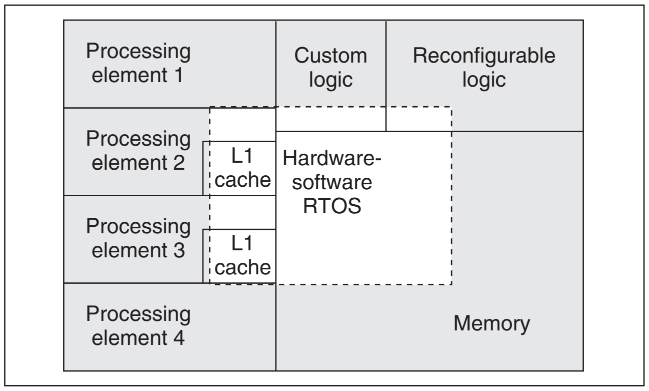


Figure 1. Sample SoC architecture.

Programming model

We focus our research on multiprocessor SoC designs. We assume that these SoCs use current programming models for real-time embedded systems. Specifically, we assume that applications execute with multiple threads of control, using the shared-address-space or message-passing programming models.

Research goal

An important goal of our research is that our RTOS-SoC designs be easy for programmers to use. The 1990s saw the spectacular failure of several large parallel-processing companies such as Thinking Machines and MasPar. Although we cannot point to a single cause of the failures, the general opinion seems to be that these companies' large machines were very difficult to program. To avoid programming difficulty, we deliberately use a few (four to 10) processors with existing compilers such as GCC. Thus, no innovation is required for compiler and processor design in our SoCs. Instead, we leverage processors and compilers available on the market. Specifically, for our simulation engine, we use the Mentor Graphics Seamless Coverification Environment (CVE), which comes with Seamless Processor Support Packages. More than 30 are available, including packages for various Advanced RISC Machine (ARM) processors, IBM PowerPC (PPC) processors, and DSPs (http://www.mentor.com/seamless/).

Atalanta RTOS

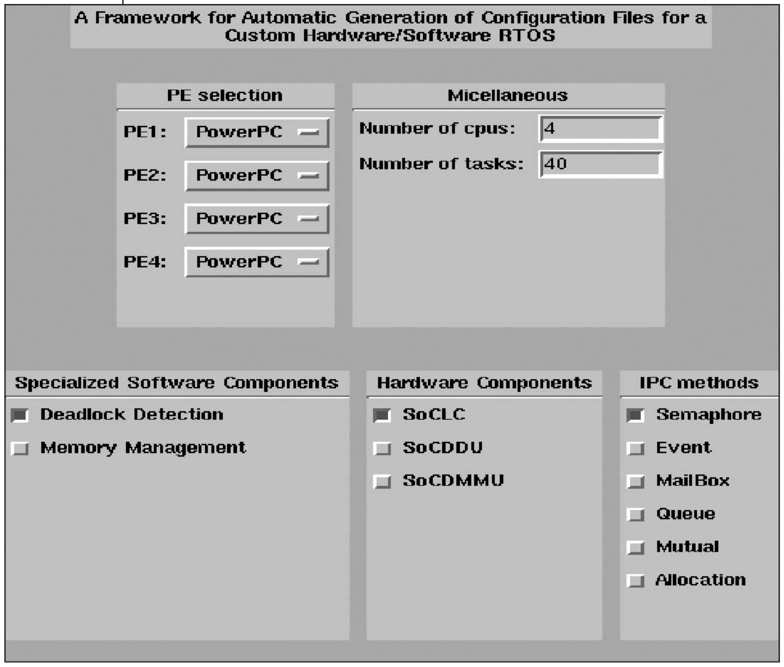
As part of the δ framework, we developed the Atalanta multiprocessor RTOS kernel for SoC architectures. Atalanta provides key RTOS features including multitasking capabilities; event-driven, priority-based preemptive scheduling; and intertask communication and synchronization. Atalanta's small, compact, deterministic, modular, library-based architecture is important for SoC applications. Atalanta also supports special features such as priority inheritance and user configurability.[3]

Atalanta provides various system objects for intertask communication and synchronization. Event groups, mailboxes, queues, semaphores, and mutexes (mutual exclusion objects) are available for tasks executing on the same processor or different processors in a homogeneous architecture. All these objects support the shared-address-space programming model. The mutex, a binary semaphore that implements priority inheritance, enables programmers to prevent priority inversion, thus providing a critical capability for real-time applications. The basic semaphore does not implement priority inheritance. Atalanta also provides message-passing system calls for communication between tasks on different processors in a heterogeneous architecture.

The Atalanta kernel is currently under evaluation in several hardware-software codesign research projects using both ARM and PPC processors.[3]

Approach

The δ framework is designed to provide automatic hardware-software configurability to support user-directed hardware-software partitioning.[1,2,4]



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: 40

Specialized Software Components

- [✓ Deadlock Detection]
- Memory Management

Hardware Components

- [✓ SoCLC]
- SoCDDU
- SoCDMMU

IPC methods

- [✓ Semaphore]
- Event
- MailBox
- Queue
- Mutual
- Allocation

Figure 2. Graphical user interface for the δ framework.

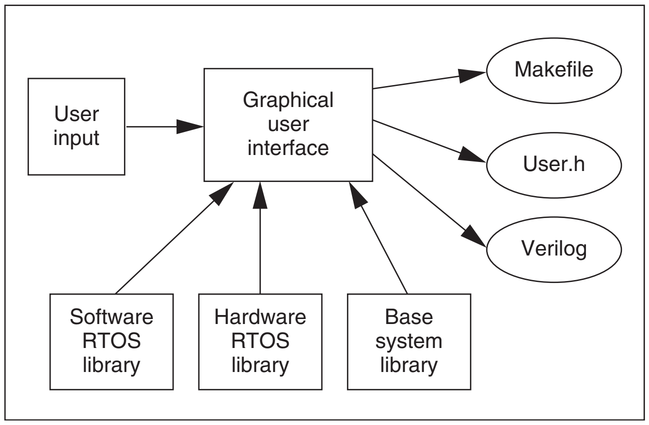


Figure 3. Automatic generation of configuration files.

Graphical user interface

Figure 2 illustrates the framework's graphical user interface, which allows the user to select RTOS features. For some features, both hardware and software versions are available. In Figure 2, the user has specified four PPC CPUs (specifically, PPC 750s). Furthermore, the user has chosen to implement deadlock detection in software and the SoC lock cache (SoCLC) in hardware. (The SoCLC, a hardware lock cache to which lock variables are memory-mapped, improves performance and real-time predictability of lock-variable access.) Finally, for interprocedure calls (IPCs), the user has chosen only semaphores under IPC methods.

Partitioning methodology

Figure 3 shows a novel approach to automating the partitioning of a hardware-software RTOS using predesigned hardware and software RTOS components. The δ framework takes the following input:

- *Hardware RTOS library*. This library contains the SoCLC, the SoC deadlock detection unit (SoCDDU), and the SoC dynamic memory management unit (SoCDMMU).
- *Base system library*. This library contains basic elements including bus arbiters and memory elements such as various caches (L1, L2, and so forth). It also holds I/O pin descriptions of all processors our system supports.
- *Software RTOS library*. This consists of the Atalanta kernel.
- *User input*. The user can select the number of processors, the processor type (such as PPC 750 or ARM9TDMI), deadlock detection in hardware (SoCDDU) or software, dynamic memory management in hardware (SoCDMMU) or software, a lock cache in hardware (SoCLC), and different IPC methods. (Although Atalanta implements all the IPC methods in software, when a specific hardware RTOS element is chosen, the IPC method may be altered to depend on hardware support. For example, if the SoCLC is chosen, the lock variables will be memory-mapped to the SoCLC.)

The δ framework outputs the Makefile, User.h, and Verilog header files shown in Figure 3. These are configuration files that combine the hardware-software RTOS IP library components specified by the user.

The RTOS hardware IP available in the δ framework includes the SoCLC, the SoCDDU, and the SoCDMMU. The SoCLC stores lock variables in a separate lock cache outside the memory system, thereby reducing lock latency, lock delay, and bandwidth consumption in a shared-memory multiprocessor SoC.[5,6] Because each lock variable requires only 1 bit, the hardware cost is low. For example, in an experimental SoC, we synthesized a SoCLC with 128 lock variables that cost approximately 7,000 gates.

The SoCDDU performs parallel hardware deadlock detection by searching for deadlocks on the resource allocation graph in hardware.[7] It provides a fast, low-area implementation of runtime deadlock detection. Compared with software, the SoCDDU reduces deadlock detection time by 99%.

The SoCDMMU implements dynamic memory allocation and management in hardware to improve both average- and worst-case memory allocation time.[8,9] For example, in a practical example, the SoCDMMU reduced average-case allocation time by 440% compared with software memory allocation (already optimized for speed).[8]

Target SoC

The δ framework targets a SoC consisting of custom logic, reconfigurable logic, and multiple processing elements, all sharing a common memory, as shown in Figure 1. Figure 4 shows an example of such a SoC, in which four PPC processors communicate with second-level memory and reconfigurable logic over a single bus. However, nothing in the δ framework requires PPC processors or a single bus; the only requirement for additional processors (currently the framework supports only the PPC 750, PPC 755, and ARM9TDMI) is that processor-specific assembly code for the software RTOS must be written for the new processor. All the hardware RTOS components have well-defined interfaces to which any processing element—including a hardware (non–Von Neumann or non-instruction-set) processing element—can connect and thus use the hardware RTOS component's features. In other words, both the custom and reconfigurable logic can contain specialized processing elements that interface to the hardware RTOS components.

Examples and experimental results

To test the δ framework, we simulated various RTOS configurations using the Mentor Graphics Seamless CVE. Figure 5 shows the δ tool flow generating five different hardware-software RTOS instantiations. All five have the

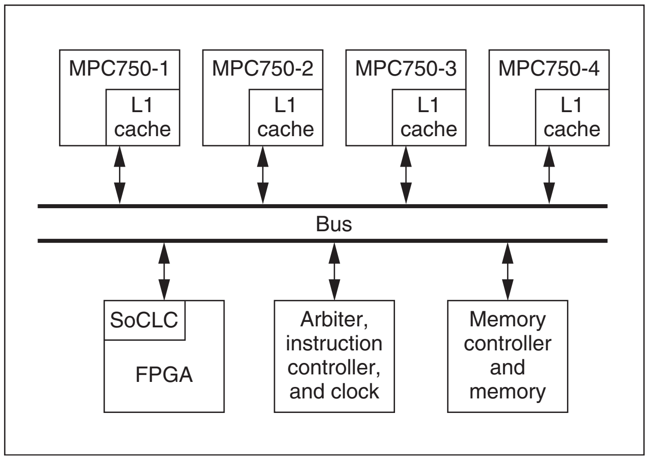


Figure 4. SoC architecture with a SoCLC.

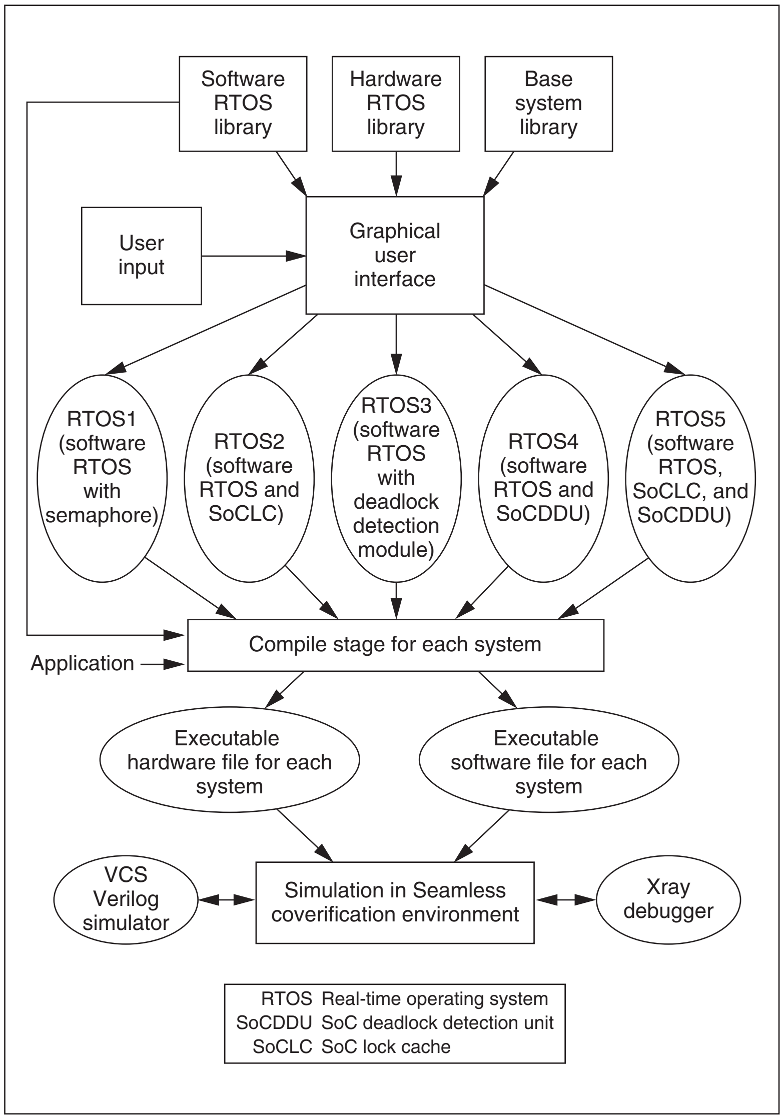

Software RTOS library
Hardware RTOS library
Base system library
User input
Graphical user interface
RTOS1 (software RTOS with semaphore)
RTOS2 (software RTOS and SoCLC)
RTOS3 (software RTOS with deadlock detection module)
RTOS4 (software RTOS and SoCDDU)
RTOS5 (software RTOS, SoCLC, and SoCDDU)
Application
Compile stage for each system
Executable hardware file for each system
Executable software file for each system
VCS Verilog simulator
Simulation in Seamless coverification environment
Xray debugger
RTOS Real-time operating system
SoCDDU SoC deadlock detection unit
SoCLC SoC lock cache

Figure 5. Generating and simulating five custom hardware-software RTOS examples.

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

Configuration	Lock latency (clock cycles)	Lock delay (clock cycles)	Execution time (clock cycles)
*RTOS1 (without SoCLC)	1,200	47,264	36.9 million
RTOS2 (with SoCLC)	908	23,590	29 million
Speedup factor	1.32	2.00	1.27

*RTOS1 used semaphores for long critical sections and spin locks for short critical sections.

Table 2. SoCLC and SoCDDU hardware area.

	Total area	
Chip technology	SoCLC	SoCDDU
Semicustom VLSI	7,435 gates using a TSMC* 0.25-μm standard cell library from LEDA**	364 gates using an AMI Semiconductors 0.3-μm standard cell library
Xilinx XC4000E 4003EPC84 FPGA	532 sequential logic gates 9,036 other gates	10 sequential logic gates 559 other gates

*Taiwan Semiconductor Manufacturing Corp.

**Library of Efficient Datatypes and Algorithms

same chip architecture: four PPC processors with a single bus and shared memory.

The first configuration, RTOS1, is for a system implementing a database transaction example with software semaphores and spin locks.[6] For comparison with RTOS1, we generated RTOS2, which uses a SoCLC along with the same database example. Figure 4 shows the SoC architecture for this case.

The third configuration, RTOS3, is a software RTOS with deadlock detection in software. It is designed for the shared-memory SoC already described. RTOS4 is for a system using the SoCDDU hardware. Finally, RTOS5 is for a system using both a SoCLC and a SoCDDU.

We simulated these systems in the Seamless framework, using the Synopsys VCS Verilog simulator and Mentor Graphics Xray for application code debugging. We compiled the software application code and the software RTOS code for execution on the PPC processors, and we instantiated the hardware part of the configured RTOS in a top-level Verilog file that could execute in the Seamless framework on the Synopsys VCS simulator. We interfaced the processors with the shared memory and other hardware RTOS components through an address decoder, an arbiter, and a memory controller.

To verify that the generated RTOS1 and RTOS2 configurations are correct, we used the same database example mentioned previously in connection with the SoCLC.[6] The example includes accesses to both short and long critical sections. Long critical sections are actual database-copying actions, whereas short critical sections are synchronization actions among server tasks and client tasks before a long data transaction begins.

Table 1 presents our experimental simulation results for the database example. It compares lock delay, lock latency, and total execution time for RTOS1 (with software locks) and RTOS2 (with SoCLC). Both configurations ran with 40 tasks. RTOS2 achieves a speedup of 27% over RTOS1 on the same architecture.

To estimate the added area cost of RTOS2, we synthesized the SoCLC to a semicustom library and to reconfigurable logic (using a Xilinx 4000 series FPGA as an example for reconfigurable logic area). Table 2 shows the hardware area we used for the SoCLC with support for up to 64 short and 64 long critical-section locks. Table 2 also shows the SoCDDU hardware area for an example we discuss later.

On the basis of these results, δ framework users can choose between two tradeoffs: Gain a speedup using the SoCLC with chip space overhead, or use software locking and save the hardware for other uses (or simply reduce overall chip size).

To verify that the generated configurations for RTOS3 and RTOS4 are correct, we considered how the SoCDDU would perform in an example based on a Jini lookup service application[10] in which client applications can request services through intermediate layers (lookup, discovery, and join). This system has four clients (four PPC processors) and four services: peripheral component interconnect (PCI), MPEG, fast Fourier transform (FFT), and wireless interface hardware units. The first processor, MCP750-1, processes video streams. The second, MPC750-2, completes signal-processing algorithms. The third, MPC750-3, handles fax, voice, and e-mail. The fourth, MPC750-4, handles communication functions. Because the system has multiple requesters and resources, deadlock is possible. Therefore, the system could benefit from the SoCDDU.

Figure 6 illustrates the example. The sequence of requests and grants leads to a deadlock caused by event e_5 at time t_5 (a previous article[7] presents the sequence in detail). We did not obtain the processing times shown from an actual industrial product; rather, we estimated them to exemplify an application running setup code in C to dynamically change (at runtime) hardware resource use among the processors. We assumed that if a processor 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.

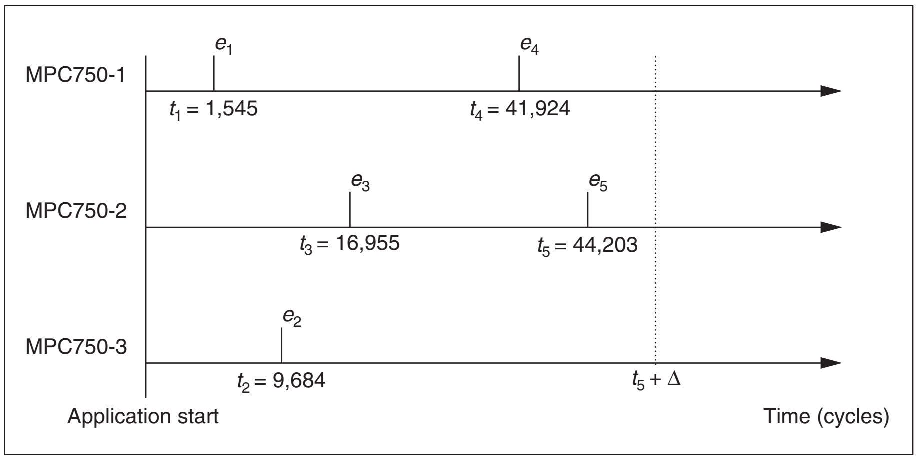


Figure 6. Deadlock event sequence.

Table 3. Software versus SoCDDU in performance time.

Deadlock detection method	Detection time Δ (clock cycles)	Total execution time $t_5 + \Delta$ (clock cycles)
Software algorithm	16,928	61,131
SoCDDU	2	44,205

Suppose that we start deadlock detection at time t_5 and that the time needed for deadlock detection is Δ. We compared the Δ values required with two deadlock detection methods: software versus SoCDDU. We compared the two methods' effects on deadlock detection time and total execution time. Table 3 shows the results. Total execution time speeds up 38% in this example. Table 2 shows the hardware area used by the SoCDDU.

Both the SoCLC and the SoCDDU are scalable according to the number of processors, lock variables, or hardware resources. Thus, δ framework users can consider various predefined hardware-software RTOS partitions and SoC architectures.

Incorporating the δ framework in platform-based design

How might a designer use the δ framework in platform-based design? That depends on whether the user is designing a custom platform ASIC (also known as a SoC) for eventual fabrication or using an existing platform ASIC.

In the former case, the δ framework helps in design space exploration of the SoC architecture. For example, if fast locking is important for the kinds of applications to be run, the designer might want a SoCLC in custom hardware. On the basis of this application analysis, the designer would use the framework to narrow down the set of RTOS choices and corresponding SoC architecture choices. Then, the designer can simulate each customized RTOS and SoC architecture in a cycle-accurate simulator such as the Seamless CVE to obtain sample code traces for the applications to be run on the SoC. Seamless CVE also supports non-cycle-accurate simulation modes for faster design space exploration, and the δ framework works with these modes as well.

Designers should also consider additional factors such as power consumption. Then, they can make a final choice as to which SoC to fabricate. After fabrication, the processors, the custom hardware RTOS services, and the amount of reconfigurable logic included on the SoC will limit the RTOS choices available.

In the postfabrication scenario, a particular real-time application might have a rare deadlock condition, which, if encountered, should trigger system reset as soon as possible. An already fabricated SoC with on-chip reconfigurable logic could implement the hardware SoCDDU in reconfigurable logic for this application. Potentially, any of the hardware RTOS services could be placed in reconfigurable logic. Therefore, to the extent possible (which is limited by area), the δ framework can generate a hardware-software RTOS customized for a particular set of application requirements to be run on a platform ASIC with reconfigurable logic. Even without reconfigurable logic or custom RTOS logic included on the SoC, the framework is useful, but perhaps no more useful than an existing software RTOS available commercially.

Finally, the δ framework can be extended to include additional hardware and software components to generate a wider range of customized RTOSs. For example, a designer might notice that the application software would run much faster and consume less energy if a particular RTOS function were implemented completely or partly in hardware. Although retooling the RTOS generation framework requires effort (much as integrating new instructions into a compiler does), a user with access to and understanding of the source code could add such functionality, or the company providing the framework could make these extensions. In fact, the existing three hardware RTOS IP components resulted from our trying to run application programs in a multiprocessor SoC and then asking, How can we speed up these programs dramatically with a very small hardware addition (less than 20,000 gates) to the RTOS? In short, we consider the δ framework to be an early conceptual prototype of the RTOS side of the equation in RTOS-SoC or RTOS-platform ASIC codesign.

WE BELIEVE that platform-based design must pay more attention to software in the platform definition process. Bearing this out, our results with the δ framework show that SoC architectures can benefit greatly from early codesign with a hardware-software RTOS. In particular, we have shown several examples in which a small hardware area (less than 10,000 gates) results in speedups of 27% or more in likely application scenarios. In our future work, we plan to more fully automate RTOS generation by integrating a wider variety of RTOS hardware components in the δ framework. ■

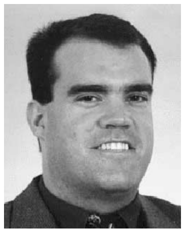
Acknowledgments

National Science Foundation grants CCR-9984808, CCR-0082164, and INT-9973120 funded this research. The State of Georgia under the Yamacraw initiative provided additional funding. We also acknowledge donations from Denali, Hewlett-Packard, Intel, LEDA, Mentor Graphics, Sun Microsystems, and Synopsys. Finally, we acknowledge graduate students and researchers Bilge S. Akgul, Jaehwan Lee, Kyoung-Keol Ryu, Mohamed Shalan, Pun Shiu, Eung Shin, Di-shi Sun, and Yudong Tan, who did much of the work reported in this article.

■ References

1. G. De Micheli and M. Sami, eds., *Hardware/Software Co-Design*, Kluwer Academic, Boston, 1996.
2. *Proc. IEEE*, Special Issue on Hardware/Software Co-Design, IEEE Press, Piscataway, N.J., Mar. 1997.
3. D. Sun et al., *Atalanta: A New Multiprocessor RTOS Kernel for System-on-a-Chip Applications*, tech. report GIT-CC-02-19, Georgia Inst. of Technology, Atlanta, 2002; http://www.cc.gatech.edu/pubs.html.
4. R.K. Gupta, *Co-Synthesis of Hardware and Software for Digital Embedded Systems*, Kluwer Academic, Boston, 1995.
5. B.S. Akgul and V. Mooney, "System-on-a-Chip Processor Support in Hardware," *Proc. Design, Automation, and Test in Europe* (DATE 01), IEEE CS Press, Los Alamitos, Calif., 2001, pp. 633-639.
6. B.S. Akgul, J. Lee, and V. Mooney, "System-on-a-Chip Processor Synchronization Hardware Unit with Task Preemption Support," *Proc. Int'l Conf. Compilers, Architecture, and Synthesis for Embedded Systems* (CASES 01), ACM Press, New York, 2001, pp. 149-157.
7. P. Shiu, Y. Tan, and V. Mooney, "A Novel Parallel Deadlock Detection Algorithm and Architecture," *Proc. Int'l Symp. Hardware/Software Codesign* (CODES 01), ACM Press, New York, 2001, pp. 30-36.

8. M. Shalan and V. Mooney, "A Dynamic Memory Management Unit for Embedded Real-Time System-on-a-Chip," *Proc. Int'l Conf. Compilers, Architecture, and Synthesis for Embedded Systems* (CASES 00), ACM Press, New York, 2000, pp. 180-186.
9. M. Shalan and V. Mooney, "Hardware Support for Real-Time Embedded Multiprocessor System-on-a-Chip Memory Management," *Proc. Int'l Symp. Hardware/Software Codesign* (CODES 02), ACM Press, New York, 2002, pp. 79-84.
10. S. Morgan, "Jini to the Rescue," *IEEE Spectrum*, vol. 37, no. 4, Apr. 2000, pp. 44-49.



Vincent J. Mooney III is an assistant professor in the School of Electrical and Computer Engineering and an adjunct assistant professor in the College of Computing, both at the Georgia Institute of Technology in Atlanta. His research interests include computer-aided design of integrated circuits with emphasis on hardware-software codesign, real-time operating systems, and low-power architectures and associated compilers. Mooney has a BS in electrical engineering and computer science from Yale University, an MS in electrical engineering and an MA in philosophy from Stanford University, and a PhD in electrical engineering from Stanford. He is a member of the IEEE and the ACM.

Douglas M. Blough is a professor of electrical and computer engineering at the Georgia Institute of Technology, where he also holds a joint appointment in the College of Computing. His research interests include system dependability, sensor networks, and multicomputer architecture and operating systems. Blough has a BS in electrical engineering and an MS and a PhD in computer science from Johns Hopkins University. He is a senior member of the IEEE and a member of Tau Beta Pi and Eta Kappa Nu.

■ Direct questions and comments about this article to Vincent J. Mooney III, School of Electrical and Computer Engineering, Georgia Inst. of Technology, Atlanta, GA 30332-0250; mooney@ece.gatech.edu.