

A Model and Methodology for Hardware-Software Codesign

IN HARDWARE-SOFTWARE code design, designers consider trade-offs in the way hardware and software components of a system work together to exhibit a specified behavior, given a set of performance goals and an implementation technology. Because of a wide range of possible system structures and design goals, the hardware-software code design problem takes on many forms.

One type of codesign seeks to accelerate application software by extracting portions for implementation in hardware. Programmable hardware may make this type of software acceleration common even in general-purpose computing. In this case, the codesign problem entails characterizing hardware and software performance, identifying a hardware-software partition, transforming the functional description into such a partition, and synthesizing the resulting hardware and software.

The approach we describe in this article addresses these problems in the more general case in which the initial functional specification may consist of both hardware and software. Research in this area of codesign may yield

DONALD E. THOMAS
JAY K. ADAMS
HERMAN SCHMIT
Carnegie Mellon University

This article presents a behavioral model of a class of mixed hardware-software systems and defines a codesign methodology for such systems. The methodology includes hardware-software partitioning, behavioral synthesis, software compilation, and demonstration on a testbed consisting of a commercial CPU, field-programmable gate arrays, and programmable interconnections.

insight and techniques that apply to other forms of hardware-software codesign as well.

Several emerging technologies provide a

starting point for addressing this type of hardware-software codesign. Hardware-software cosimulation is a means of verifying the functionality of mixed hardware-software descriptions. High-level (or behavioral) synthesis can produce hardware implementations for functions described in a high-level software language such as C. Also, recent work in high-level synthesis of multiple-process systems¹ suggests that functionality can be moved from one thread of control to another.

What is needed is a means of applying these techniques to generating hardware-software partitioning alternatives and a formalism that describes the available engineering trade-offs. Toward this end, we have developed a model for system-level simulation and synthesis that provides a detailed understanding of system behavior and a transformation capability that allows generation of design alternatives. We are also carrying out experimental and theoretical work aimed at identifying factors that influence design decisions with respect to a given set of trade-offs and goals. This work, along with a developing

set of CAD tools, is the basis for the code design methodology diagrammed in Figure 1. Parts of the methodology are still in development.

Our methodology focuses on two system design tasks: cosimulation and cosynthesis. An important issue in cosimulation is how to tie behavioral hardware simulation into a software runtime environment. Cosynthesis involves two interrelated design issues: choosing the optimal hardware-software partition and choosing the appropriate level of control concurrency. The hardware-software partition is defined by the set of application-level functions implemented with application-specific hardware. Finding a suitable hardware-software partition requires understanding how best to use limited application-specific hardware resources to meet system design goals. A system's control concurrency is defined by the functional behavior and interaction of its processes. Finding an appropriate level of control concurrency entails redrawing process boundaries by merging or splitting process behaviors, or by moving functions from one process to another, in response to system performance goals.

As Figure 1 shows, our methodology uses a cosimulation environment to develop a mixed hardware-software description, producing one that is functionally correct but may not meet some design goals or may not be realizable with the given implementation technology. Hardware-software cosynthesis modifies the hardware-software partition and control concurrency so that the target system's behavior will meet design goals. Then, we compile the resulting specifications into hardware and software for implementation, using standard techniques for software and behavioral synthesis tools for custom hardware.² Finally, we use a testbed consisting of field-programmable hardware and interconnections residing in the backplane of a general-purpose computer system for experimental measurement.

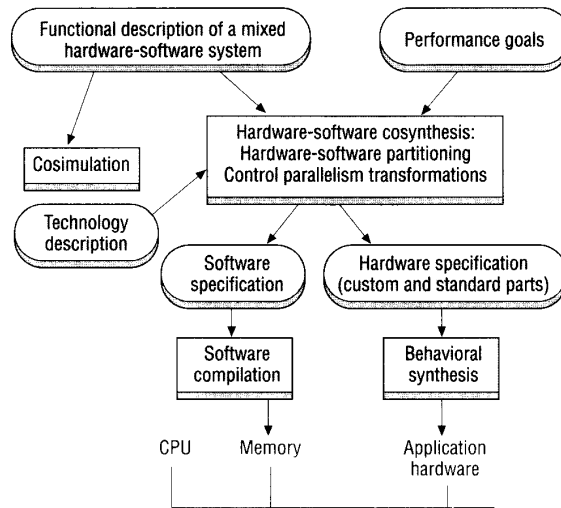


Figure 1. Hardware-software codesign methodology.

The system model

To discuss our codesign methodology, we assume a system architecture such as that shown in Figure 2. It consists of some application-specific hardware on the system bus of either a general-purpose or an embedded computer system running an appropriate operating system. The application-specific hardware is designed to cooperate with application software running on the CPU. The bus interface works with the operating system device driver to translate into the proper handshakes and data transfers for the application-specific hardware. We assume that the CPU is running an operating system capable of communicating with the hardware device and performing interrupt-driven I/O and that the hardware device includes an appropriate bus interface. The system may include memory and other I/O devices, but we make no assumptions as to their presence or location.

We developed the system behavioral model for system simulation and synthesis with several goals in mind:

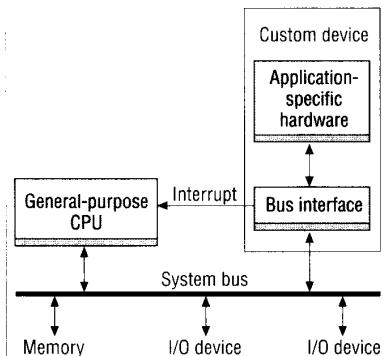


Figure 2. System architecture.

- to describe the behavior of the application-level software and application-specific hardware explicitly
- to hide the details of the operating system and, as much as possible, the hardware architecture
- to be explicit about the level of concurrency and the hardware-software partition
- to facilitate transforming the concurrency level and hardware-software partition

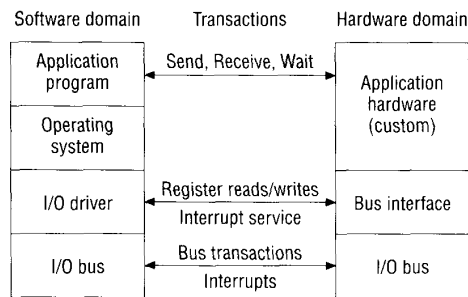


Figure 3. Abstractions for hardware-software interaction.

We model the system behavior as a set of independent, interacting, sequential processes,³ each described behaviorally in a high-level description language. The processes in the system model correspond to the hardware processes (independent state machines) and software processes (as provided by the operating system) that make up the initial system description. The use of communicating sequential processes provides the ability to reason naturally about the system's concurrency level. When each process is designated either hardware or software, the communicating sequential process model also captures the hardware-software partition of system functionality.

A key feature of the model is the abstraction level at which it represents hardware-software interaction. Figure 3 illustrates several abstraction levels at which we could model hardware-software interaction. At the lowest level, we could model bus transactions issued by the CPU, along with how the custom hardware decodes, interprets, and reacts to those transactions. This would necessitate modeling how the bus interface decodes addresses and under what circumstances it can interrupt the CPU.

At a higher level, we could model the device driver as interacting with a bus interface directly. At this level of abstraction, the model can hide address decoding and interrupt behavior of the hardware device, allowing us to consid-

er only data transfer between the driver and the hardware device.

Instead, we model the application program and application hardware interaction at a high level, where the details of the operating system and the device driver, as well as those of the bus interface, are hidden. We make some assumptions about the capabilities of the operating system, its device driver, and the bus interface to gain the ability to formalize the interaction of user-level software and application-specific hardware at the communicating processes level.

A set of interprocess communication primitives captures the abstract interaction of application hardware and software processes, classifying the synchronization and data transfer associated with each interaction. In modeling software, we represent an I/O system call such as a read or write to the hardware device by an appropriate interprocess communication primitive. In the case of a process described in a hardware description language, where data transfer and synchronization are often represented as explicit port operations,⁴ a single interprocess communication primitive represents the set of port operations that perform the data transfer and associated synchronization.

The set of process communication primitives covers the following types of process interaction:

- *Synchronized data transfer.* Syn-

chronized data transfer is data transfer between two processes accompanied by a mechanism ensuring that when the sending process transmits the data, the receiving process is in an appropriate state to receive it. If the receiving process is not in such a state when the sender initiates the transaction, the sender takes appropriate action, either blocking until the receiver becomes ready or continuing with unrelated processing.

- *Unsynchronized (unbuffered) data transfer.* When a data transfer is not buffered or synchronized, a single data value may be received more than once or not at all. Such is often the case with status information. A process may send data to another process regardless of whether that data or any previous data values have actually been received. Subsequent data transmissions will overwrite earlier ones, making it impossible for another process to receive any but the most recent data sent. A process may also receive data from another process regardless of whether the data has already been received or even whether data has been sent.

- *Synchronization without data transfer.* A process may synchronize with another, even if no data transfer is needed, by suspending itself until the other process reaches a certain state. A process may use this mechanism either to enable another process to begin a task or to wait for another process to complete a task.

Processes may also communicate by sharing a common memory space. Our model does not represent this situation explicitly; instead, it models the shared memory itself as a process that communicates with other processes using the process communication primitives. In this way, access to shared memory is made explicit, as is the set of processes that ac-

cess a given shared-memory region.

The behavior of a software process may include calls to the operating system that are unrelated to interprocess communication. These calls are represented explicitly in the descriptions of software processes.

System input and output takes place in two ways: hardware processes may refer to external signals, implying some sort of physical, external connection to the application hardware; or input and output may be handled by standard I/O devices (a serial port, for instance), managed by the operating system. In the latter case, inputs and outputs are represented by calls to the operating system from within a software process.

By exposing the behavior of important elements of the system while hiding implementation details, this model facilitates both system simulation and a number of synthesis tasks, such as hardware-software partitioning and control concurrency transformation. Moreover, we believe it will be applicable to a wide variety of systems because of the few assumptions it makes about the underlying hardware architecture.

System simulation

We have implemented a hardware-software cosimulation environment according to the system behavioral model just described. We use the Verilog simulator to perform behavioral simulation of the system hardware processes. The software processes run as separate Unix processes and communicate with the hardware simulator by means of the BSD (Berkeley Software Distribution) Unix socket facility. Becker, Singh, and Tell have used a similar technique.⁵ The difference in our cosimulation environment is that many aspects of the system are hidden by the abstraction used for hardware-software interaction.

The designer must change the application software slightly so that it opens socket connections to the hardware simulator instead of I/O channels to the ac-

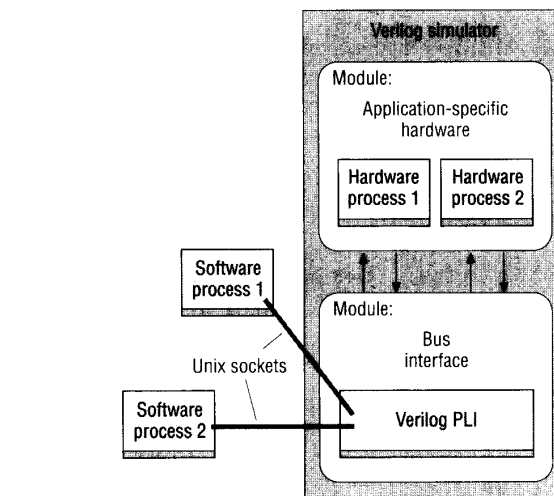


Figure 4. Cosimulation using the Verilog hardware simulator.

tual hardware device. We added routines to the Verilog simulator via the Verilog Programming Language Interface (PLI) to translate socket I/O into simulation events, allowing the hardware simulation models to communicate with the software processes. Figure 4 shows a diagram of the simulation environment.

Because of the similarity between device and socket I/O in Unix, using the socket facility for communication with the hardware simulator makes it possible for the simulator to act largely as a drop-in replacement for the actual hardware device. Also, because the BSD socket facility allows transparent operation over a network, we can run distributed simulations, with the hardware simulator running on one system and the various software processes on others.

In the Verilog simulation environment, one or more modules comprise the application-specific portion of the hardware. A separate module acts as the bus interface. The bus interface module translates the socket activity into the appropriate simulation events. The routines that do this translation are implemented primarily in C and linked

to the Verilog simulation environment through the Verilog PLI.

A major issue in getting separate software processes to communicate with processes in the Verilog behavioral simulation environment is how to enable the socket events (reads or writes pending on a socket) to create simulation events. Ideally, the Verilog simulator would react to socket events as shown in Figure 5 (next page). At the end of each time step, if no future simulation events existed, the simulator would suspend operation pending activity on a socket. If future simulation events did exist, the simulator would check the sockets for activity, create any socket-related simulation events, and continue running. In this way, the simulator would stop running only when simulation could not proceed without the occurrence of a socket event. Checking the sockets for activity at the end of each time step prevents a hardware process from excluding software interaction by looping indefinitely (as might be the case for a clock or a free-running counter).

For effective hardware-software co-

```

while (sockets are open or simulation event queue is not empty)
{
    while (simulation events present in current time step)
    {
        process simulation event
    }
    if (future simulation events exist)
    {
        advance simulation time
        if (socket events present)
        {
            produce simulation events for socket events
        }
    }
    else
    {
        wait for socket event
        produce simulation events for socket events
    }
}

```

Figure 5. Ideal interaction of simulator and socket events.

simulation, it is important to tie the hardware simulation environment to the natural software runtime environment. Our work on cosimulation illustrates one way of achieving that tie. Unfortunately, restrictions on the way the PLI interacts with the simulation event scheduler in Verilog prevent us from implementing that technique directly. Instead, we implemented two compromise solutions: one that works efficiently with a restricted class of hardware descriptions and one that works in all cases but is fairly inefficient.

System synthesis

The general cosynthesis problem addressed by our methodology is how to meet a system performance goal, or improve system performance, with a minimum of hardware resources. Beginning with a behavioral system description in terms of interacting processes, we must extract a set of hardware and software processes that will comprise the implementation.

Since the process boundaries in the initial system description may not represent ideal hardware-software boundaries, we first decompose the processes into nontrivial sequences of operations called tasks. How operations should be grouped into tasks is one consideration. Once this grouping has been done, the problem for cosynthesis is to find the subset of tasks that should be implemented in hardware and to determine how tasks should be grouped into processes. A key feature of our cosynthesis methodology is that hardware-software partitioning takes place at the task level, rather than at the operation level as in Gupta and De Micheli's approach.⁶

Any cosynthesis decision is evaluated according to how it affects the system's performance and cost characteristics. Cost and performance requirements depend on the particular system. For example, an embedded controller may have real-time deadlines to meet. In that case, design decisions that move toward this performance requirement take pri-

ority over other decisions. Minimizing the cost of such a system is also desirable, but secondary to meeting the performance requirement. On the other hand, a system consisting of a fixed set of resources must meet cost constraints. In this case, design decisions are evaluated primarily on their ability to satisfy the cost constraints.

Given some basis for evaluating system performance, we can decide which tasks should be implemented as hardware and which as software. For some tasks, the decision may be clear: If a task interacts closely with the operating system (for example, makes many OS calls or relies on virtual memory), software may be the only feasible implementation. Likewise, if a task interacts closely with external signals, implementing it in hardware may be the only feasible solution. For the remaining tasks, either implementation is possible. We can determine which to pursue according to the following criteria:

1. Dynamic properties of the system: a characterization of how a task's execution time impacts system performance
2. Static properties of the task: the difference in execution times between hardware and software implementations of the task
3. Hardware costs: the amount of custom hardware required to realize a hardware implementation of the task

The first consideration takes into account how much system performance depends on the execution time of each task, which in turn depends on the criterion by which system performance is measured. In a system for which maximum throughput is the design goal, we may measure the dependence of system performance on task performance simply by counting the average number of times the task must execute for each sample of input data. In a system with hard real-time constraints, the measure

of performance might be how well, if at all, the software tasks can follow some real-time scheduling discipline. How a task's execution time impacts system performance in this case depends on such factors as the priority of the task, the periodicity of the task, and scheduling overheads.

We must consider the second criterion, static properties of task behaviors, because some tasks are inherently much better suited for hardware implementation than others. Tasks that exhibit a high degree of data parallelism or that would benefit from a custom memory architecture, for instance, would be better suited for custom hardware implementation than serial tasks or tasks that closely fit a general-purpose architecture. To quantify these differences, we must identify properties of a task's behavior that indicate how software and hardware implementations of the task will perform. The properties we have identified so far rely on unique capabilities of custom hardware. These properties are the following:

- The need for *arbitrary arithmetic operations*. Some operations are expensive or clumsy in software because they are not common enough to be included in the function of a general-purpose ALU (bit reversal, for example). In a hardware implementation, where functional units can be customized for the application, such operations can be included as primitive ALU functions.
- The ability to exploit a high degree of *data parallelism*. Although VLIW (very long instruction word) and superscalar techniques are being used to increase the data parallelism of general-purpose processors, arbitrarily high data parallelism can be achieved only in a hardware implementation.
- The ability to use *multiple threads of control*. Multiple threads of control

are possible in a software implementation, but at high cost. If the threads run on a single processor, the operating system must perform task switches, greatly impacting performance. Using multiple CPUs to implement concurrent software threads adds costs to the system even if the additional tasks are small. In hardware, however, a separate thread of control requires only an additional state machine, making fine-grained control parallelism much more practical.

- The need for *customized memory architectures*. In a software implementation, memory bandwidth is limited to that supported by the CPU and memory subsystem. A hardware implementation can achieve high memory bandwidth at low cost by employing a memory architecture tailored to the application. Independent static RAMs, for instance, can be used to store a set of arrays to be accessed in parallel.

Finally, we must consider the amount of custom hardware necessary to reduce a task's execution time. For some tasks, custom hardware implementations might perform well but be impractical due to high gate counts or memory requirements. For others, there may be a range of achievable performance gains, depending on how much of the custom hardware is devoted to the task. Furthermore, the hardware implementation cost for a given set of tasks may vary according to how the tasks are grouped into processes. Along with deciding which set of tasks to implement in hardware, we must also decide how the limited amount of custom hardware should be allocated to the various hardware tasks.

Examples

Two examples illustrate the cosynthesis considerations discussed in the previous section. The first example demonstrates the extent to which a task's static properties

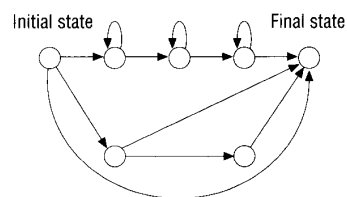


Figure 6. Hidden Markov model for phonemes.

determine its suitability for hardware implementation. The second shows how a system's dynamic properties, as well as hardware costs, affect the hardware-software partitioning trade-off for a complete, though simple, system.

Example 1: speech phoneme recognition.

The Sphinx speech recognition system⁷ uses hidden Markov models (HMMs) to represent three levels of speech knowledge: phonemes, words, and sentences. An HMM is a set of transitions connecting a set of states. The set of states includes an initial and final state. There are two probability functions for each transition: The *transition probability function* determines the probability that a particular transition will be taken. The *output probability density function* determines each alphabetic output symbol's conditional probability of being emitted, given that the transition is taken. The system performs speech recognition by determining the HMM that best matches a given input at the phoneme, word, and phrase level. The HMMs for each phoneme share the same topology, shown in Figure 6.

The Sphinx system's front end converts a continuous speech signal into a stream of 8-bit vectors. The phoneme recognizer receives one of these vectors every 10 milliseconds and determines which of the 48 phonemes modeled is most likely to have emitted the past sequence of input vectors. The recognizer makes this determination by using the *forward algorithm*, which computes, for

Table 1. Dataflow graph (DFG) depth for the Sphinx phoneme recognizer.

Behavior	DFG depth
Serial operations and serial memory accesses	149
Parallel operations and serial memory accesses	87
Parallel operations and parallel memory accesses	22

each state in an HMM, the probability that the past sequence of vectors would have been emitted from this HMM if it were in that state at that moment. The probability computed for a particular HMM's final state is the probability that the entire HMM emitted the past sequence of vectors. The phoneme recognizer determines the most likely phoneme by selecting the final state with the highest probability.

Because the probability computations for each transition require a large number of multiplications, designers frequently accelerate the implementation of the forward algorithm by transforming all the probabilities into logarithms, thus changing multiplications into additions. The probability of a state is the sum of the probabilities of the transitions incident upon it. The use of logarithmic probabilities, however, makes this addition difficult and time-consuming to compute exactly. Most implementations approximate the sum by subtracting the smaller logarithmic probability from the larger and using this difference to address a lookup table containing a correction factor, which is added to the larger logarithm. This method avoids all intermediate multiplications and logarithm conversions.

The evaluation of the probability at each state depends only on the probability of its ancestors during the last computation and the probability functions of the

transitions incident upon it. Therefore, all the state computations can be concurrent. Five types of operations are used in computing the HMM: additions, subtractions, comparisons, overflow checks, and array accesses. In total, there are 149 operations in one iteration of the phoneme recognition task. The critical path through the task in terms of data dependencies, however, is only 22 operations. This level of concurrency allows an average of seven simultaneous operations. Through the exploitation of data parallelism, a custom hardware implementation will probably outperform a scalar CPU, even if the CPU runs at a higher clock rate.

The custom hardware implementation of the forward algorithm has other advantages over a software implementation, even if a tightly coupled multiprocessor is used to exploit fine-grained parallelism. First, the 8-bit functional units in the custom hardware solution meet the task's requirements more efficiently than a general-purpose processor, which ordinarily has large bit widths and many functions not needed by this task. Unutilized bits and operations could result in lower performance of the software implementation, despite the processor's possible technology advantages. Second, the functional units' interconnection matches the task's requirements. Therefore, the custom hardware implementation has much less communication and synchronization overhead than a general-purpose processor solution.

Finally, the monolithic memory architecture used by most general-purpose processors prevents independent table lookups from being carried out in parallel. The ability to perform memory operations in parallel can improve the performance of this task significantly. As Table 1 shows, performing nonmemory operations in parallel reduces the task's critical path from 149 to 87 operations. Performing memory operations in parallel further reduces the critical path to 22 operations, indicating that mem-

ory parallelism is an important component in the total parallelism of the task. A custom hardware implementation can allocate a dedicated storage unit for each array, allowing concurrent memory accesses and avoiding the inefficiencies and overheads of the general-purpose memory hierarchy, such as cache misses, cache cold-start, and coherency overhead.

The phoneme recognizer exhibits three of the four task properties described earlier: reliance on arbitrary arithmetic operations, a high degree of data parallelism, and the need for a customized memory architecture. This task has the added advantage of a relatively low communication bandwidth with other procedures in the speech recognition system. In 10 milliseconds, the phoneme recognizer receives a new 8-bit vector and returns 48 updated phoneme probabilities, for a total communication rate of 4,900 bytes per second. The cost of a hardware implementation of this task is also reasonable; initial estimates indicate that the phoneme recognizer will fit into the three FPGAs (field-programmable gate arrays) present on the Rasa board described later in this article.

Example 2: data compression/encryption. This example shows how the system synthesis considerations outlined earlier might affect hardware-software partitioning for the simple system diagrammed in Figure 7. For two of the tasks, reading data from the disk and transmitting it over the local area network, there are few implementation alternatives. Implementing these tasks in hardware would be difficult to justify, since they require a high degree of interaction with the operating system. The other tasks (data compression, frame assembly, and data encryption) are much less predisposed to either hardware or software and may be implemented as either. Assuming that the system design goal is to maximize throughput using a limited amount of custom hardware, the cosynthesis problem is to determine

which subset of tasks should be implemented with custom hardware and which with software to achieve that goal.

First, consider how much a task's execution time impacts system throughput. For the system shown in Figure 7, we determined the number of executions per input byte for a sample data set. Table 2 shows how often each task is performed for every byte of data read from the disk. The data compression task, for instance, is executed 10 times as frequently as the encryption task.

Table 3 addresses the second consideration, how the execution time of a hardware implementation compares to that of a software implementation. The table lists the number of clock cycles per task invocation for software and hardware implementations (assuming a simple model of software execution in which each instruction takes one cycle). These results show that custom hardware can be up to six times as fast as software, depending on the task.

By combining the information from Table 2 and Table 3, we calculate the normalized execution time (cycles of execution per byte of input) of each task for hardware and software implementations (Table 4). To group the tasks into two independent processes, we should partition them so that the sums of the normalized execution times of the tasks are roughly equal for both processes.

Finally, we must consider the amount of hardware required to achieve the desired performance gain. The execution time listed for the encryption task in Table 3 represents only one possible hardware implementation, one using eight independent memories. Table 5 shows how the number of independent memories in a hardware implementation of the encryption task affects execution time. By incorporating this information into an analysis like that shown in Table 4, we can make a decision about the necessary amount of hardware resources, as well as hardware-software process

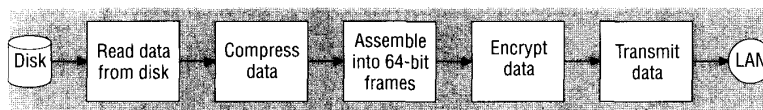


Figure 7. Data compression/encryption system.

partitioning.

On the basis of this sort of analysis, we might decide to implement the compression task with a single software process and to combine the frame assembly and encryption tasks into a single hardware process. Since the compression process is the throughput bottleneck in this case, a single-memory implementation of the encryption task

Table 2. Task invocations as determined by system simulation.

Task	Calls	Calls per input byte
Data compression	1,431	1.00
Frame assembly	753	0.53
Encryption	141	0.10

Table 3. Task execution time for software and hardware implementations.

Task	Software (cycles)	Hardware (cycles)	Ratio
Data compression	39	21.5	1.81
Frame assembly	29	7.2	4.03
Encryption	576	94.0	6.15

Table 4. Normalized execution time for software and hardware implementations.

Task	Software (cycles per input byte)	Hardware (cycles per input byte)
Data compression	39.0	21.5
Frame assembly	15.4	3.8
Encryption	57.6	9.4

Table 5. Execution time for hardware implementations of encryption task.

Number of independent memories	Task runtime (cycles)	Normalized execution time (cycles per input byte)
8	94	9.4
4	110	11.0
2	142	14.2
1	206	20.6

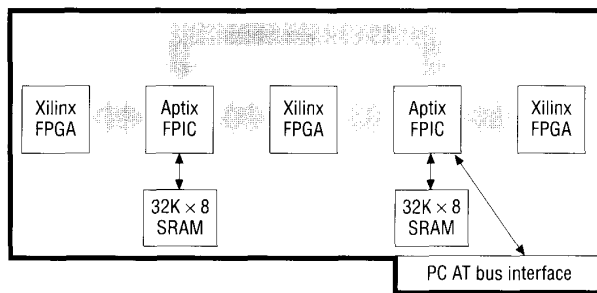


Figure 8. The Rasa Board.

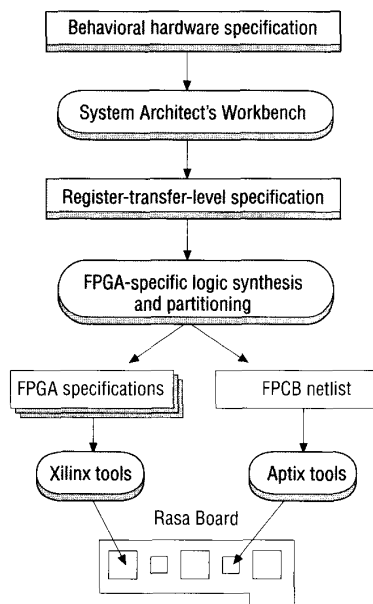


Figure 9. Hardware compilation design flow for the Rasa Board.

would suffice. The result would be a system with two processes: a software process with a normalized cycle time of 39, and a hardware process with a normalized cycle time of 3.8 for frame assembly and 20.6 for encryption, totaling 24.4.

The encryption task of this example is much better suited to hardware than to software. Further examination of the task's behavior reveals the main reason: The task uses operations that are time-

consuming implemented as software but simple and efficient implemented in hardware, where arbitrary sets of operations can be incorporated in a functional unit. The encryption task can also benefit from the use of multiple, independent memories to store arrays. Such considerations, along with the task's role in the system, help us draw process boundaries and select a suitable hardware-software partition.

A prototype system

To test and validate our codesign methodology and tools, we have designed a prototype hardware-software system. It consists of an Intel-486-based PC and the Rasa Board (from *tabula rasa*, meaning blank tablet, referring to the human mind in its initial state, before receiving impressions from the external world). The Rasa Board, shown in Figure 8, is a field-programmable platform for application-specific hardware. Like the Splash board⁸ and the Any-Board,⁹ the Rasa Board consists of Xilinx FPGAs, memories, and a microcomputer interface. Unlike the other boards, the Rasa Board's components interconnect via two Aptix field-programmable interconnect chips (FPICs) on a field-programmable circuit board (FPCB). Because the interconnection network is programmable, the task of partitioning the design into separate physical components is not further complicated by a requirement to map the interchip nets


onto a fixed interconnection scheme. This fact simplifies partitioning and improves the results of FPGA placement, because I/O pins can be placed according to internal placement considerations rather than external interconnection requirements.

To create hardware designs for this board from a behavioral specification, we have coupled a behavioral synthesis tool, the System Architect's Workbench, to a set of logic synthesis and partitioning tools. Figure 9 shows the design flow. The system output is a set of FPGA specifications and a netlist for the FPCB. Using Xilinx and Aptix tools, we convert these specifications to the configuration bit streams for the FPGAs and the FPCB.

The Rasa Board is presently under construction in our research group. When the board is complete, we will use it to implement hardware processes in the prototype hardware-software system. On the basis of our initial estimates, we believe the Rasa Board has sufficient hardware resources to build implementations of the phoneme recognizer and the encryption tasks that will outperform software implementations on the Intel 486.

THE CODESIGN METHODOLOGY presented here defines a mixed hardware-software system model that facilitates cosimulation and cosynthesis. The abstract level of the communicating sequential process model allows the designer and the design tools to reason about system functions at a level appropriate for codesign. Partitioning and transformation merge and/or split processes to meet performance requirements or to fit physical constraints. Further, the methodology ties into the detailed hardware design process through behavioral synthesis, allowing functions originally conceived as software to be implemented as hardware when performance constraints dictate.

The design examples we presented illustrate how certain characteristics of sys-

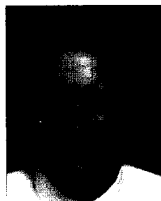
tem behavior and constraints suggest hardware or software implementation. As the development of the partitioning tool proceeds, these characteristics will play an important role in the decision-making process. Finally, the Rasa Board project will enable us to measure and characterize the effects of the cosynthesis tools in actual implementations. 

Acknowledgments

We acknowledge the Semiconductor Research Corporation, the National Science Foundation (under contract MIP-9112930), General Motors Research, the Xilinx Corporation, and the Aptix Corporation for the funding of the projects described here.

References

1. J.W. Hagerman and D.E. Thomas, *Process Transformation for System-Level Synthesis*, Tech. Report CMUCAD-93-08, Carnegie Mellon Univ., Pittsburgh, 1993.
2. D.E. Thomas et al., *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench*, Kluwer, Boston, 1990.
3. C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, N.J., 1985.
4. L.F. Arnstein, *Describing Systems for High-Level Synthesis in the Verilog Language*, Tech. Report CMUCAD-90-51, Carnegie Mellon Univ., 1990.
5. D. Becker, R.K. Singh, and S.G. Tell, "An Engineering Environment for Hardware/Software Co-Simulation," *Proc. 29th Design Automation Conf.*, IEEE Computer Society Press, Los Alamitos, Calif., 1992, pp. 129-134.
6. R.K. Gupta and G. De Micheli, "System-Level Synthesis Using Re-programmable Components," *Proc. Third European Conf. Design Automation*, IEEE CS Press, 1992, pp. 2-7.
7. K.F. Lee and H.W. Hon, "Large Vocabulary Speaker-Independent Continuous Speech Recognition," *Int'l Conf. Acoustics, Speech, and Signal Processing* (Vol. 1), IEEE, New York, 1988, pp. 123-126.
8. M. Gokhale et al., *SPLASH: A Reconfigurable Linear Logic Array*, Tech. Report SRC-TR-90-012, Supercomputer Research Center, Institute for Defense Analyses, Bowie, Md., 1990.
9. D.E. Van den Bout et al., "AnyBoard: An FPGA-Based, Reconfigurable System," *IEEE Design & Test of Computers*, Vol. 9, No. 3, Sept. 1992, pp. 21-30.



Donald E. Thomas is a professor of electrical and computer engineering at Carnegie Mellon University, where he works on automatic design of digital systems and hardware-software codesign. In 1985-86 he was a visiting scientist at the IBM T.J. Watson Research Center. Thomas was elected a fellow of the IEEE for his contributions to automatic design of integrated circuits and systems and to computer engineering education. He was the 1989 Design Automation Conference chair and served on the IEEE Computer Society Board of Governors in 1989 and 1990. He received his PhD from Carnegie Mellon University.



Jay K. Adams is a PhD student in the Electrical and Computer Engineering Department of Carnegie Mellon University. His research interests include high-level synthesis and hardware-software codesign. Previously, he was a member of the technical staff at Hewlett-Packard's Engineering Systems Laboratory. He received a BS degree from MIT and an MS degree from Carnegie Mellon. He is a member of the IEEE, the IEEE Computer Society, Tau Beta Pi, and Eta Kappa Nu.



Herman Schmit is a PhD student in the Electrical and Computer Engineering Department of Carnegie Mellon University, performing research on the high-level synthesis of memory architecture. Earlier, he worked in Data General Corporation's High-End Systems Development Department. He received a BSE degree from the University of Pennsylvania and an MSEE degree from Carnegie Mellon.

Send correspondence about this article to Donald E. Thomas, ECE Dept., Carnegie Mellon University, Pittsburgh, PA 15213; thomas@ece.cmu.edu.