Program Implementation Schemes for Hardware-Software Systems

Rajesh K. Gupta, University of Illinois at Urbana-Champaign

Claudionor N. Coelho Jr. and Giovanni De Micheli, Stanford University

Recent advances in the design and synthesis of integrated circuits[1,2] have prompted system architects to investigate computer-aided design methods for systems that contain both application-specific and *predesigned* reprogrammable components. Although a reprogrammable microprocessor like the Mips R3000 can implement most system functionality as a program, dedicated application-specific integrated circuits (ASICs) are needed for performance reasons. In this context, recent advances in ASIC synthesis and the proliferation of advanced and inexpensive processors have stimulated interest in hardware/software codesign.[3]

For the most part, we can apply high-level synthesis techniques to synthesis of systems containing processors by treating the latter as a "generalized resource." However, the problem is more complex, since the software on the processor implements system functionality in an instruction-driven manner with a statically allocated memory space, whereas ASICs operate as data-driven reactive elements. Due to these differences in computational models and primitive operations in hardware and software, a new formulation of the problem of cosynthesis is needed.[4]

An early version of this article was presented at the IEEE/ACM-sponsored First International Workshop on Hardware/Software Codesign, Estes Park, Colorado, September 30-October 2, 1992.

An overview of our cosynthesis approach

Figure 1 illustrates our cosynthesis approach. We specify system behavior using HardwareC,[5] a hardware description language (HDL) that has a C-like syntax and supports timing and resource constraints. It also supports specification of unbounded and unknown delay operations that can arise from data-dependent decisions and external synchronization operations. The particular choice of a HDL to specify system functionality is immaterial for the cosynthesis formulation here, and we could use other HDLs such as Verilog.

The HDL description is compiled into a system graph model based on dataflow graphs.[4,6] The system graph model consists of vertices representing operations and edges representing serialization of those operations (dependencies). The system control flow orders the model's concurrent dataflow sections.

Timing constraints are specified on specific statements in the input HDL description. In particular, constraints on I/O data rates (that is, the rate at which data is produced or consumed) are specified using I/O statements. For an analysis of timing constraints, see R. Gupta.[6]

The system graph model is partitioned based on the feasibility of a hardware-software implementation and the satisfaction of timing constraints. The partitioning scheme relies on identifying unbounded delay operations.[7] After partitioning, we have a set of concurrently executing hardware and software models that consist of hierarchical, acyclic system graph models.

Since data-dependent operations can have unbounded delays, we refer to these operations as *points of synchronization* and schedule them dynamically. This is achieved by using a relative schedule of operations.[5] Briefly, a relative scheduling formulation allows a data-driven dynamic schedule of operations that is feasible under timing constraints. Because hardware and software rates of execution may differ, it is important to allow multiple executions of individual hardware and software modules to achieve an efficient system implementation. Moreover, different execution rates cause variation in rates of communication across hardware and software. This form of distributed computation requires the use of appropriate buffering and handshaking mechanisms.[8]

The software graph models are then serialized to minimize temporary register storage requirements. From the serialized graph models, we generate a corresponding C-code description. Existing software compilers compile the C-code into assembly code for the target processor.

To verify the correctness of implementations, we use Poseidon, a simula-

0018-9162/94/$4.00 © 1994 IEEE

tor that performs concurrent execution of multiple functional models implemented as software or hardware.[8] The hardware component can be simulated either before or after the structural synthesis phase. The former simulation is performed at the system graph level and the latter at the logic level. Poseidon supports simulation of assembly code for the 8051 microcontroller and the DLX microprocessor, a RISC-oriented load/store processor that supports the Mips R3000 instruction-set architecture.

Input to Poseidon consists of model declarations, interconnections, and corresponding interfaces. Each model has an associated clock signal and clock cycle time used for its simulation. The interface is specified using components such as wires, queues, registers, memories, and their protocols.

The rest of this article covers the software components generated by partitioning the graph model shown in the shaded area in Figure 1.

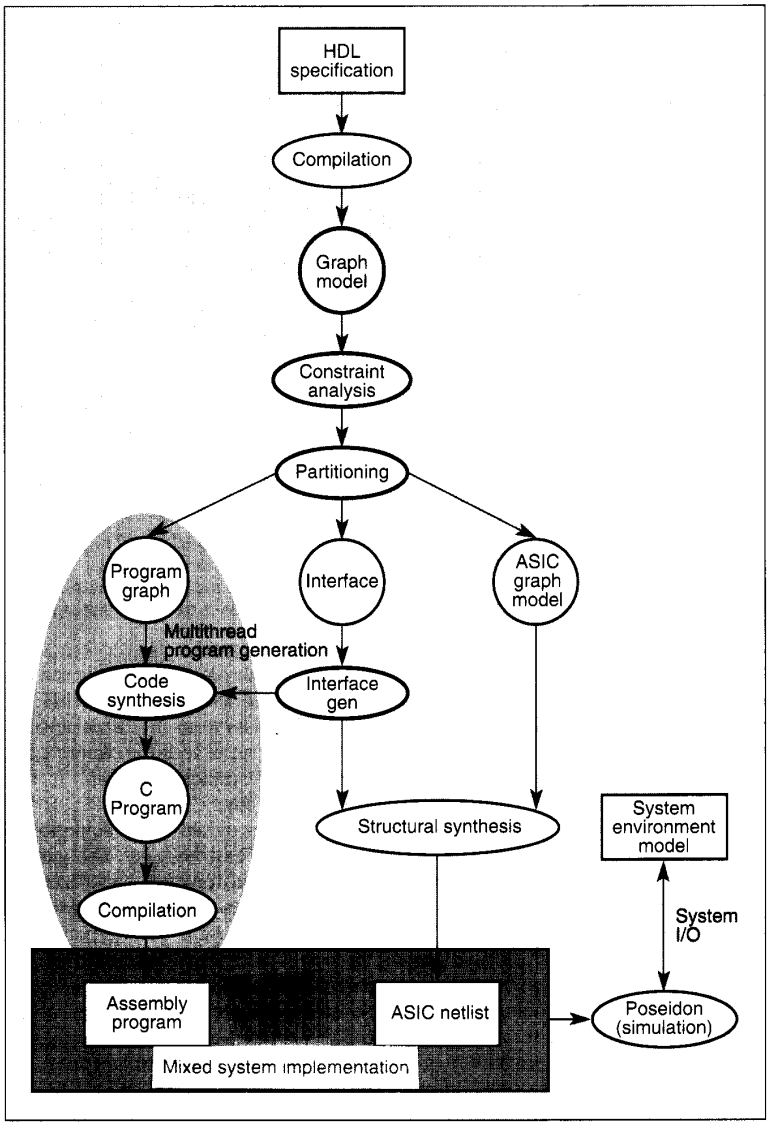

Figure 1. System synthesis procedure.

Target architecture and assumptions

Figure 2 shows our target architecture, a general-purpose processor assisted by application-specific hardware components. To simplify cosynthesis, we make the following assumptions:

- We use a single reprogrammable component because multiple reprogrammable components require additional software synchronization and memory protection to facilitate safe multiprocessing.
- We assume that the memory subsystem has a single level to avoid the complexities that arise when analyzing and synthesizing hierarchical memory subsystems.
- The reprogrammable component is always the bus master. Whereas most processors come with facilities for bus control, inclusion of such functionality on an ASIC would greatly increase total hardware cost.

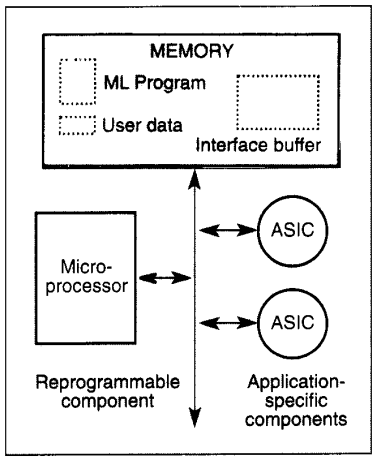

Figure 2. Target system architecture.

```
process example (a, b, c)
  in port a[8];
  in channel b[8];
  out port c;
{
  boolean x[8], y[8], z[8];
  x = read(a);
  y = receive(b);
  if (x > y)
    z = x - y;
else
    z = x * y;
  while (z >= 0) {
    write c = y;
    z = z - 1;}
}
```

Figure 3. Example of a HardwareC process with unbounded delay operations.

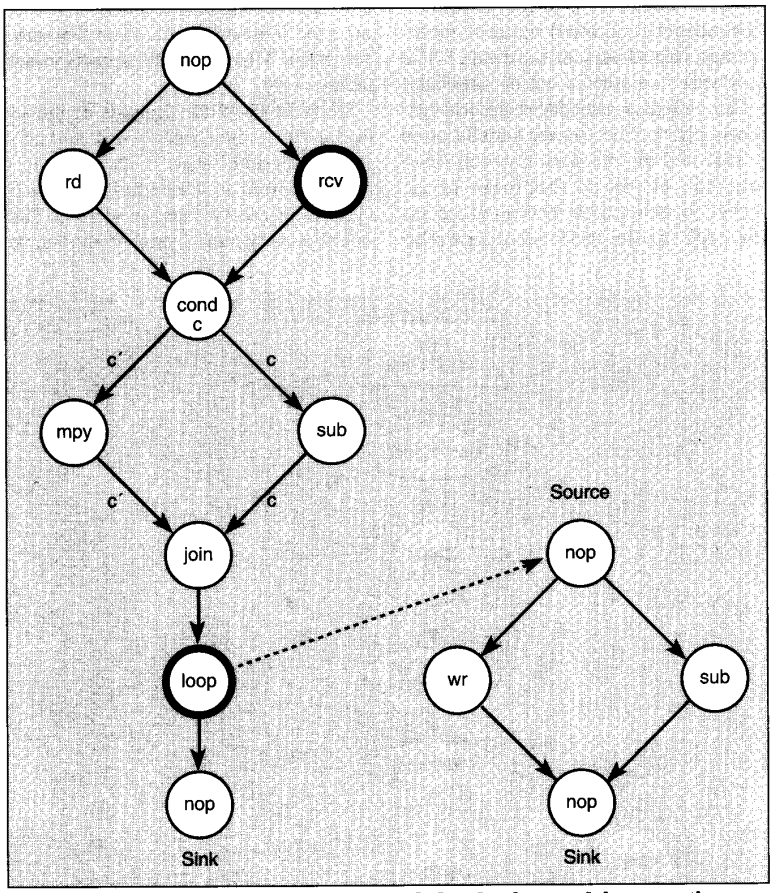

Figure 4. A graph model containing unbounded and unknown delay operations.

- Finally, we assume that the reprogrammable component contains a sufficient number of maskable interrupt input signals. To simplify further, these interrupts are unvectored, and each has a predefined, unique destination address.

The target architecture consists of the most essential elements of hardware-software systems. Although simplified to make the cosynthesis problem tractable, it is general enough to implement a large class of embedded systems.

Implementation of software components

As mentioned above, the software component is described as a set of hierarchical, acyclic graph models. Using hierarchy to describe calls and loops, the graph model pushes the uncertainty of conditional execution paths into the uncertainty of delay. The uncertainty of loop executions reflects the uncertainty as to how many times a called graph model is executed.

Figure 3 shows a HardwareC process description containing message-passing receive, conditional, and loop operations. A process in HardwareC executes concurrently with other processes in the system specification. A process restarts itself upon completion of the last operation in the process body. Thus, there exists an implied outermost loop that contains the body of the process model. The process body can be specified with varying degrees of parallelism, and operations within a process body need not be executed sequentially.

The two graphs in Figure 4, G_0 and G_{loop}, comprise the hierarchical graph model for the process example. The bold circles indicate operations with unbounded or unknown execution delays. Whether a graph can be implemented as single or multiple routines depends on the points of synchronization. A simple graph model that lacks multiple synchronization points can be implemented as a single routine. A hierarchical system model is implemented as a set of routines, where each routine corresponds to a graph in the model hierarchy.

We call a program implementation a program thread because the graph model's operations are serialized in software. Thus, the software component consists of a set of program threads. These threads may be hierarchically related, or some may need to be executed concurrently. Concurrency can be achieved by using an interleaved execution model as we explain later.

A program thread can be initiated by a synchronization operation such as a blocking receive operation (rcv_synch). However, all operations in a thread

have fixed delays, so the (unknown) delay in executing the synchronization operation appears as a delay in scheduling the program thread and is not considered a part of the thread latency. Therefore, an upper bound on the latency of each thread is known statically.

The graph model in Figure 4 has two program threads, T_0 and T_{loop}. T_0 consists of the receive operation followed by the port-read and other operations, while T_{loop} consists of serialized operations in the corresponding graph body (see Figure 5).

Upon synchronization, all the operations in a given graph model will eventually execute. Thus, the corresponding routines can be constructed with known and bounded latencies. As with the graph model, the uncertainty due to data-dependent delay operations is related to the individual routine's invocations. A software implementation with dynamic invocations of fixed-latency program threads simplifies the software's capability to satisfy data-rate constraints. We can determine bounds on software performance even in the presence of unbounded delay operations based on its implementation of program threads.

In the following sections, we describe a code-level transformation of the data-dependent loop operations that makes it possible to observe imposed I/O rate constraints. In cases where such transformations are not possible, we use processor interrupts, along with bounds on the number of interrupts and interrupt latencies, to ensure satisfaction of rate constraints.

Rate constraints and software performance. We derive the data-rate constraints on the software component's I/O from the corresponding constraints on system I/Os. A data-rate constraint on an input (output) specifies a lower bound on the samples per second that the I/O data can consume (produce). In a deterministic software component, that is, a software component with known and bounded execution delays, precise data rates can be computed and checked against corresponding data-rate constraints. However, the presence of an unbounded delay operation between consecutive read (write) operations requires computation of statistical measures — such as the distribution of input data value and interarrival time — to determine the rate of data production and consumption.

Data-dependent loop operations are major contributors to the variability of data rates, since the delay due to these operations consists of active execution times rather than idle-wait-type delays encountered by the external synchronization operations. In some cases, we can avoid statistical measures by transforming the dynamic loop execution model into a corresponding *pseudostatic* loop execution model, as follows.

Figure 6 shows a software component that reads a value and then performs the data-dependent delay operation shown in Figure 7. The ASIC sends data to the processor on port x at an input rate constraint of ρ samples per second. The processor's function is modeled by the HardwareC process fragment in Figure 7a, wherein x is a Boolean array that represents an integer. In its software implementation, this behavior is translated into the two program threads in 7b, one performing the read operations and the other performing operations in the body of the loop. For each execution of thread T1, there are x executions of thread T2.

For the HardwareC process in Figure 7a, the overall execution time of the While statement determines the interval between the read operation's successive executions. Due to the variable-delay loop operation, the input data rate at port x is variable and is dependent upon the value of x as a function of time. For each invocation of thread T1, there are x invocations of thread T2 before thread T1 can be resumed. In the absence of any other data-dependency to operations in the loop body, thread T1 can be restarted before completing all invocations of thread T2 by buffering the data transfer from thread T1 to T2. Further, if variable x is used only for indexing the loop iterations,

T_0	T_{loop}
rcv_synch	loop_synch
read	write
cond_eval	op sub
cond_jump	detach
op add	
op mpy	
detach	

Figure 5. T_0 and T_{loop} program threads.

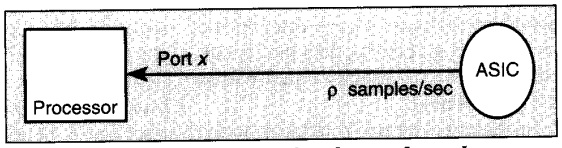

Figure 6. A mixed implementation that reads a value followed by a data-dependent delay operation.

```
process test(x, ...)
  in port x [SIZE];
{
  ...
  read x;
  while (x >= 0)
  {
    <loop-body>
    x = x - 1;
  }
}
```

(a)

Thread T1	Thread T2
read	loop_synch
detach	<loop_body>
	x = x - 1
	detach

(b)

Figure 7. (a) HardwareC process fragment; (b) program threads.

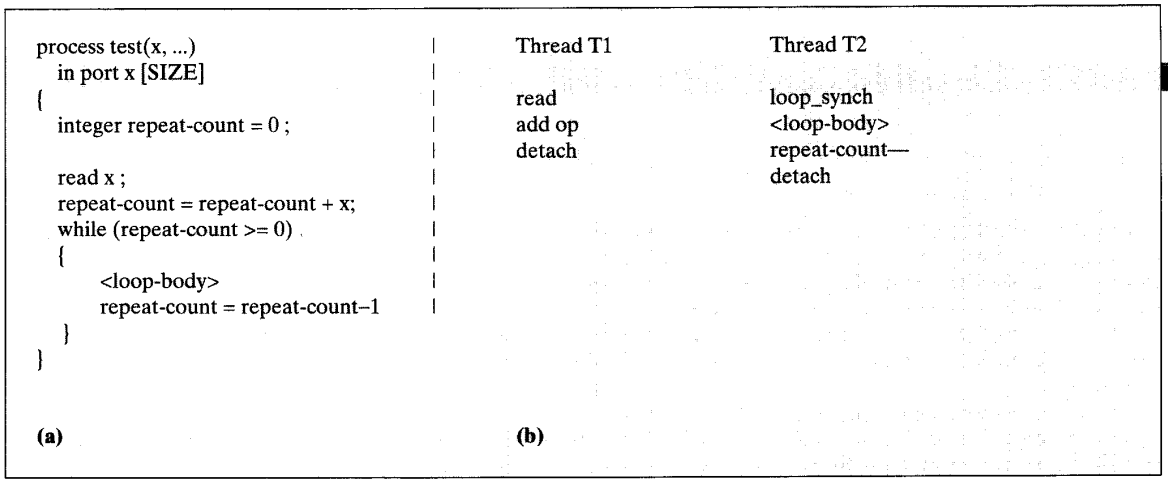

Figure 8. Transformation of (a) data-dependent loop into (b) a pseudostatic loop.

the need for an interthread buffer can be obviated by accumulating the value of x in a separate loop counter as shown in Figure 8. We call this loop construct a pseudostatic loop because data-rate constraints on I/Os affected by the data-dependent loop operation statically determine an upper bound on the loop body's number of iterations.

A pseudostatic loop implementation assumes that there is a repeat-count counter associated with every loop and that a loop body must execute as long as its repeat-count is a nonzero number. Additionally, the corresponding loop body does not use the repeat-count for any purpose other than keeping a count of the remaining iterations. Under such conditions, the above component can be transformed into two program threads, where one thread reads port x and increments the repeat-count for the loop body contained in the other thread.

In this case, we can provide a bound on the rate at which port x is read by scheduling the read thread T1 after m iterations of the loop body. (Care must be taken to avoid overflow of this counter. Generally, overflow can be avoided if m is greater than or equal to the average value of x. In the extreme, we can guarantee that it will not overflow if m is greater than the maximum of x, which is equivalent to assigning worst case delay to the loop operation.)

Next we consider the problem of software synchronization and scheduling mechanisms that make a hardware/software design feasible.

Control flow in the software component

Our model for software components relies on sequential execution of each thread. Concurrency is achieved through interleaved execution of the threads. Multiple program threads starting with unbounded delay operations can also be created out of a graph model. Therefore, software synchronization is needed to ensure correct ordering of operations within and between the different program threads.

With multiple threads, we represent the control flow with a directed flow graph in which the nodes represent individual program threads and the edges indicate control dependencies (Figure 9a). Since the total number of program threads and their dependencies are known statically, we construct the program thread so that it observes these dependencies.

The threads are identified by unique tags. A tag can be, for example, its entry point into the code memory. A runtime FIFO (first in, first out) called *control FIFO* maintains the identifier tags of the threads that are ready to run based on control flow (but may still be waiting for data). Before detaching, each thread performs one or more *enqueue* operations to the FIFO for its successor threads. In Figure 9b, Body refers to the linearized set of operations from the corresponding graph models. Control dependency from thread T1 to T2 is built into the code of T1 by the enqueue operation on the control FIFO (Figure 9c).

A thread dependency on more than one predecessor thread (that is, a multiple indegree or fanin node in the flow graph) is satisfied by ensuring multiple enqueue operations for the thread by means of a counter. For example, a thread node with an indegree of two would contain a synchronization preamble code as indicated by the While statement shown in Figure 10a. T1 is enqueued by T2 and T3, since there are dependencies from T2 and T3 into T1. Therefore, T1 must wait for control to be transfered from T2 and T3. This control transfer is achieved by counting the number of times T1 is enqueued in the control FIFO.

Control transfer for multiple fanin nodes entails program overheads that

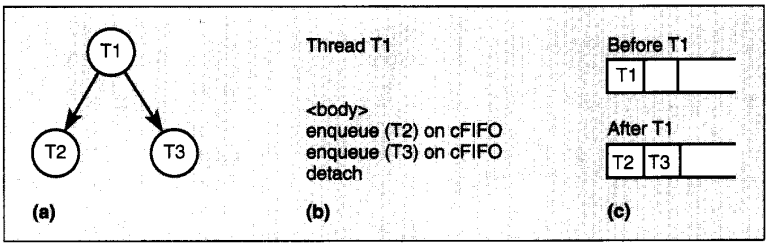

Figure 9. (a) Flow graph with control dependencies; (b) linearized set of operations; (c) enqueue operation on control FIFO.

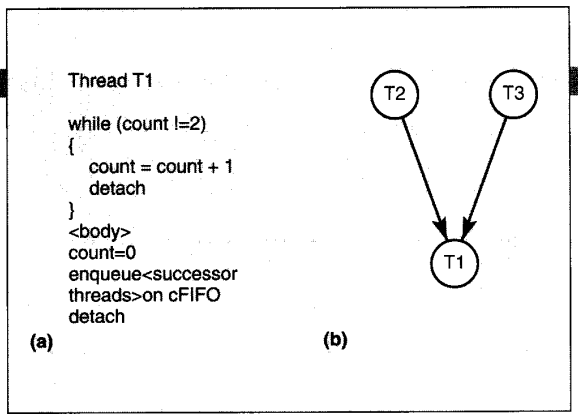

Figure 10. (a) Thread with multiple input control dependencies; (b) directed flow graph.

```
#include "transfer_to.h"

int lastPC[MAXCOROUTINES] = {scheduler, circle,
  line, main};
int current = MAIN;

int *controlFIFO_out = (int *) 0xaa0000;
int *controlFIFO  = (int *) 0xab0000;
int *controlFIFO_outak = (int *) 0xac0000;

#include "line.c"
#include "circle.c"

void main() {
     resume(SCHEDULER);
};

int nextCoroutine;

void scheduler() {
     resume(LINE);
     resume(CIRCLE);
     while(!RESET) {
       do {
            nextCoroutine = *controlFIFO;
       } while ((nextCoroutine & 0x4) != 0x4);
       resume(nextCoroutine & 0x3); } }
```

Figure 11. Main program of the graphics controller software component.

add to the corresponding threads' latency. For this reason, an attempt should be made to reduce multiple dependencies for a program thread through careful dependency analysis. In case of multiple outdegree nodes in the flow graph, we serialize the enabling of successor threads.

We have addressed the problem of achieving concurrency through interleaved execution in software.[8] The different program threads can be implemented as program subroutines that operate under a global task scheduler (or the main program). We can significantly reduce the overhead or number of cycles for each interthread control transfer operation by placing program subroutines in a cooperative, rather than hierarchical, relationship and implementing them as coroutines.[10] For a DLX microprocessor,[9] a coroutine implementation of program threads constitutes an overhead of 19 cycles. We achieve this low overhead by determining coroutine transfers in our model at compilation time, and thus an efficient use of registers prevents the high overhead costs of more general transfer schemes.

Another scheme for software implementation is to construct a program routine using description by cases.[11] In this method, we construct a single program with a unique case assignment for each thread (in a rather large conditional in the final program). A set of global state registers is used to store the state of execution of each thread. The overhead due to this scheme depends strongly upon the processor's instruction-set architecture. For the DLX microprocessor, the overhead amounts to 35 cycles for each interthread transfer operation. With so-called CISC processors, this scheme reduces the overhead by reducing the amount of ALU operations and slightly increasing memory I/O operations.

Hardware-software synchronization

Given the software component's serial execution, a data transfer from hardware to software must be explicitly synchronized. Using a polling strategy, we can design the software to perform premeditated transfers from hardware based on its data requirements. This requires static scheduling of the hardware component. In cases where software is communication limited, that is, where the processor is busy-waiting for an I/O operation most of the time, such a scheme would be sufficient. With no unbounded delay operations, the software component in this scheme can be a single program thread and data channel, since all data transfers are serialized. However, this would not support any branching or reordering of data arrivals, since dynamic scheduling of operations in hardware would not be supported.

To accommodate different rates of software/hardware execution with unbounded delay operations, we use dynamic scheduling of different threads based on availability of data. One scheduling mechanism is the above-mentioned control FIFO that attempts to ensure that data items are consumed in the order produced. The hardware-software interface consists of data queues on each channel and a control FIFO that receives and holds the enabled program threads' identifiers in the order they arrive. The control FIFO depth is sized with the number of threads of execution, since a thread execution is stalled pending availability of the requested data.

The control FIFO and associated control logic can be implemented either in hardware as a part of the ASIC or in software. If the control FIFO is implemented in software, the FIFO control logic is no longer needed, since the control flow is already in software. In this case, the data-available lines from data queues are connected to the processor's unvectored interrupt lines, where the respective interrupt service routines are used to enqueue the thread identifier tags into the control FIFO. During enqueue operations, the interrupts are disabled to preserve the software control flow's integrity.

Table 1. A comparison of control FIFO implementation schemes.

Scheme	Program Size (bytes)	Synchronization Overhead Delay (% cycles)	Input Data Rate^{-1} (cycles/coordinate)	Output Data Rate^{-1} (cycles/coordinate)			
				line		circle	
				average	peak	average	peak
Hardware CFIFO	5,972	0	81	535.2	330	76.4	30
Software CFIFO	6,588	50.0	95	749.5	502	106.8	31
Optimized software CFIFO	6,360	29.4	95	651.0	407	94.0	31

Results

To illustrate the effectiveness of hardware/software interface implementation, we present a portion of a design for a graphics controller that outputs pixel coordinates for lines and circles given the end coordinates or radius. The system design's final implementation consists of line- and circle-drawing routines in the software component and an ASIC that performs initial coordinate generation and coordinate transfer to the video RAM. The software component has two execution threads corresponding to the line- and circle-drawing routines. Both program threads generate coordinates used by the ASIC. The data-driven dynamic scheduling of program threads is achieved with a three-deep control FIFO. The circle- and line-drawing program threads are identified by identifier numbers 1 and 2, respectively. The program threads are implemented by using the coroutine scheme described above. Figure 11 shows the main program of a hardware control FIFO implementation.

Table 1 compares the performance of different program implementations using control FIFO in hardware and software. The hardware implementation of a control FIFO with a fanin of three, when synthesized and mapped to LSI Logic 10K Library of gates costs 228 gates. An equivalent software implementation adds 388 bytes to the software component's overall program size. Note that the cost of hardware control FIFO increases as the number of data queues increases. On the other hand, software implementation of control FIFO using interrupt routines to perform enqueue operations offers lower implementation costs with a 50-percent increase in the thread latencies. For software implementation of control FIFO, the enqueue and dequeue operations are described in C and are then compiled for DLX assembly. The overhead for enqueue and dequeue operations is reduced further by manually optimizing the assembly code for these operations (see Optimized software CFIFO in Table 1). This onetime optimization does not affect the program threads' C-code description, and it reduces program size and program thread overhead (to 29.4 percent) and improves the data output rate. Note that we express data I/O rates in terms of the number of cycles it takes to input or output a coordinate. Due to the data-dependent behavior of program threads, the actual data I/O rates would vary with the value of the input data. In this simulation, the input rate is for a simultaneous drawing of a line and a five-pixel radius with a width of 1 pixel each; the results are accurate to one pixel. An input rate of 81 cycles per coordinate corresponds to approximately 0.25 million samples per second for a processor running at 20 MHz. Similarly, a peak circle output rate of 30 cycles per coordinate corresponds to a rate of 0.67 million samples per second. An implementation of line- and circle-drawing program threads for the graphics controller that uses interthread buffering has a total program size of 5,788 bytes for a 62-percent overhead delay per program thread.

Though instructive, the line- and circle-drawing algorithms do not fully exploit the potential of a mixed implementation. However, a more computationally intensive operation like spline generation or operations involving floating-point arithmetic would greatly benefit by mixed program implementations. As an example of complex system design, consider an Ethernet-based network coprocessor. This processor is modeled as a set of 13 concurrently executing processes that interact by means of 24 send and 40 receive operations. The total HardwareC description consists of 1,036 lines of code. A mixed implementation with a software component containing 17 cases using description by cases takes 8,572 bytes of program and data storage, but it

Table 2. Software component for system examples.

Example	Program implementation	Program Size bytes	Max. Delay cycles
Graphics controller	Coroutines, Hardware CFIFO	5,972	806, 859
Network coprocessor	Description by cases, Hardware CFIFO	8,572	56

reduces the overall cost of the ASIC by 23 percent from 10,882 gates to 8,394 gates using LSI Logic 10K Library of gates. The execution unit resides in the software component and executes most of the frame assembly and disassembly operations. The mixed implementation meets the following performance requirements: a maximum propagation delay of 46.4 microseconds (μs), a maximum jam time of 4.8 μs, a minimum interframe spacing of 67.2 μs, and an input bit rate for a 10-Mbit/second operation using Ethernet protocol. Table 2 lists the software characteristics for the graphics controller and the network coprocessor.

Reprogrammable processors offer a promising means to realize low-cost embedded applications. Where possible, portions of system functionality can be delegated to the software component instead of to ASICs.

Software component design for such systems poses interesting problems because serial program execution must interact with concurrent hardware operations. In our approach to system synthesis, we implement the software component as a set of program routines called threads. We preserve the concurrency inherent in the system model by interleaving the execution of threads. Further, dynamic scheduling of fixed latency threads avoids unnecessary serialization of operations in a graph model.

However, even with the simplified target architecture, accurately characterizing and synthesizing the software component is challenging, and our work represents only a first step toward system software synthesis. We are now working to extend this model to include the effects of hierarchical memory schemes and the reduction of communication overheads by using channel-sharing and data-encoding schemes. ■

Acknowledgments

This work has benefitted enormously from discussions with Martin Freeman of Philips Research. This research was sponsored by NSF-ARPA, under grant No. MIP 9115432, and by a fellowship provided by the Center for Integrated Systems and Philips. The second author was supported by the fellowship 200212/90.7 from CNPq-Brazil.

References

1. D. Gajski, *Silicon Compilation*, Addison-Wesley Publishing, Reading, Mass., 1988.
2. R. Camposano and W. Wolf, *High-Level VLSI Synthesis*, Kluwer Academic Publishers, Norwell, Mass., 1991.
3. Presentations at the *Int'l Workshop on Hardware/Software Codesign*, 1992, 1993.
4. R.K. Gupta and G. De Micheli, "Hardware-Software Cosynthesis for Digital Systems," *IEEE Design & Test of Computers*, Vol. 10, No. 3, Sept. 1993, pp. 37-53.
5. G. De Micheli et al., "The Olympus Synthesis System for Digital Design," *IEEE Design & Test of Computers*, Vol. 7, No. 3, Oct. 1990, pp. 37-53.
6. R.K. Gupta, *Cosynthesis of Hardware and Software for Digital Embedded Systems*, doctoral dissertation, Stanford Univ., Palo Alto, Calif., 1993.
7. R.K. Gupta and G. De Micheli, "System-Level Synthesis Using Reprogrammable Components," in *Proc. European Design Automation Conf.*, IEEE CS Press, Los Alamitos, Calif., Order No. 2645-02T, 1992, pp. 2-7.
8. R.K. Gupta, C. Coelho, and G. De Micheli, "Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components," in *Proc. 29th Design Automation Conf.*, IEEE CS Press, Los Alamitos, Order No. 2822-02, 1992, pp. 225-230.
9. J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan-Kaufmann, Menlo Park, 1990, pp. 89-137.
10. M.E. Conway, "Design of a Separate Transition-Diagram Compiler," *Comm. ACM*, Vol. 6, No. 7, July, 1963, pp. 396-408.
11. P.J.H. King, "Decision Tables," *The Computer Journal*, Vol. 10, No. 2, Aug. 1967.

Rajesh K. Gupta is an assistant professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign. His primary research interest is the design and synthesis of VLSI circuits and systems. He also maintains an active interest in computer architecture and communication systems.

Gupta received BTech in electrical engineering from the Indian Institute of Technology, Kanpur, India in 1984, an MS in electrical engineering and computer science from the University of California, at Berkeley, in 1986, and a PhD in electrical engineering from Stanford University in 1993. From 1986 to 1989 he was employed at Intel Corporation, where he worked on VLSI design as a member of the design teams for the 80386-SX, 80486, and Pentium microprocessor devices. He is coauthor of a patent for a PLL-based clock circuit.

Claudionor N. Coelho Jr. is a doctoral student in the Department of Electrical Engineering at Stanford University. His primary research interests are high-level synthesis, system-level synthesis, and verification. He received a BS degree in electrical engineering and an MS degree in computer science from the Universidade Federal de Minas Gerais, Brazil in 1988 and 1990, respectively.

Giovanni De Micheli is an associate professor of electrical engineering and, by courtesy, of computer science at Stanford University. His research interests include computer-aided design of integrated circuits, with particular emphasis on automated synthesis, optimization, and verification of VLSI circuits. From 1984 to 1986 he worked at IBM's T.J. Watson Research Center, where he was project leader of the Design Automation Workstation group.

He received a Dr. Eng. degree in nuclear engineering from the Politecnico di Milano, Italy, in 1979, MS and PhD degrees in electrical engineering and computer science from the University of California, at Berkeley, in 1980 and 1983, respectively. De Micheli has published extensively and serves as associate editor of *Proceedings of the IEEE* and *IEEE Transactions on VLSI Systems* and of *Integration: the VLSI Journal*. He is a fellow of the IEEE, and he was granted a Presidential Young Investigator award in 1988.

Readers can contact the authors through R. K. Gupta at the Digital Computer Laboratory, University of Illinois at Urbana-Champaign, 1304 West Springfield Ave., Room 2214, Urbana, IL 61801; e-mail rgupta@cs.uiuc.edu.