

Hardware-Software Cosynthesis for Digital Systems

MOST DIGITAL SYSTEMS used for dedicated applications consist of general-purpose processors, memory, and application-specific hardware circuits. Examples of such embedded systems appear in medical instrumentation, process control, automated vehicles, and networking and communication systems. Besides being application specific, such system designs also respect constraints related to the relative timing of their actions. For that reason we call them real-time embedded systems.

Design and analysis of real-time embedded systems pose challenges in performance estimation, selection of appropriate parts for system implementation, and verification of such systems for functional and temporal properties. In practice, designers implement such systems from their specification as a set of loosely defined functionalities by taking a design-oriented approach. For instance, consider the design shown in Figure 1 (next page) of a network processor that is connected to a serial line and memory. This processor receives and sends data over the serial line using a specific communication protocol

RAJESH K. GUPTA
GIOVANNI DE MICHELI
Stanford University

As system design grows increasingly complex, the use of predesigned components, such as general-purpose microprocessors, can simplify synthesized hardware. While the problems in designing systems that contain processors and application-specific integrated circuit chips are not new, computer-aided synthesis of such heterogeneous or mixed systems poses unique challenges. Here, we demonstrate the feasibility of synthesizing heterogeneous systems by using timing constraints to delegate tasks between hardware and software so that performance requirements can be met.

(such as the protocol for Ethernet links). The decision to map functionalities into dedicated hardware or implement them as programs on a processor usual-

ly depends on estimates of achievable performance and the implementation cost of the respective parts. While this division impacts every stage of the design, it is largely based on the designer's experience and takes place early in the design process. As a consequence, portions of a design often are either under- or over-designed with respect to their required performance. More important, due to the ad hoc nature of the overall design process, we have no guarantee that a given implementation meets required system performance (except possibly by over-designing).

In contrast, we can formulate a methodical approach to system implementation as a synthesis-oriented solution, a tactic that has met with enormous success in individual integrated circuit chip design (chip-level synthesis). A synthesis approach for hardware proceeds with systems described at the behavioral level, by means of an appropriate specification language. While the search for a suitable specification language for digital systems is the subject of ongoing research, use of procedural hardware de-

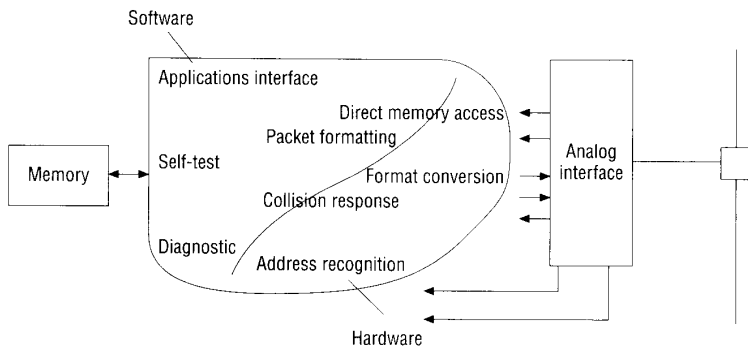


Figure 1. A design-oriented approach to system implementation.

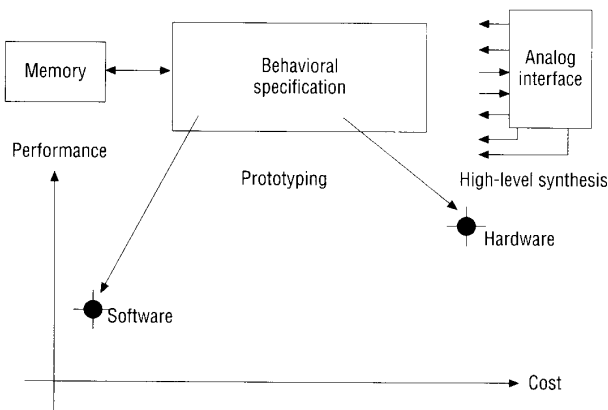


Figure 2. A synthesis-oriented approach to system implementation.

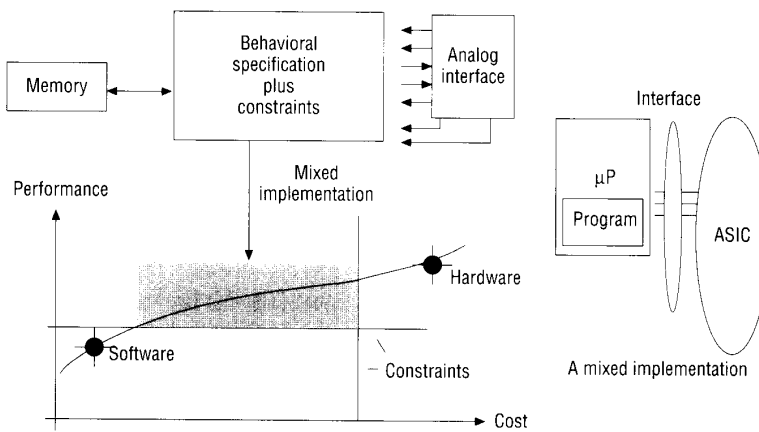


Figure 3. Proposed approach to system implementation.

scription languages (HDLs) to describe integrated circuits has been gaining wide acceptance in recent years.

A synthesis-oriented approach to digital circuit design starts with a behavioral description of circuit functionality. From that, it attempts to generate a gate-level implementation that can be characterized as a purely hardware implementation (Figure 2). Recent strides in high-level synthesis allow us to synthesize digital circuits from high-level specifications; several such systems are available from industry and academia. Gajski¹ and Camposano and Wolf² provide surveys of these. Synthesis produces a gate-level or geometric-level description that is implemented as single or multiple chips. As the number of gates (or logic cells) increases, such a solution requires semicustom or custom design technologies, which then leads to associated increases in cost and design turnaround time. For large system designs, synthesized hardware solutions consequently tend to be fairly expensive, depending upon the technology chosen to implement the chip.

On the other end of the system development cost and performance spectrum, one can also create a software prototype, amenable to simulation, of a system using a general-purpose programming language. (See Figure 2.) The Rapide prototyping system³ is one example. Designers can build such software prototypes rather quickly and often use them for verifying system functionality. However, software prototype performance very often falls short of what time-constrained system designs require.

Practical experience tells us that cost-effective designs use a mixture of hardware and software to accomplish their overall goals (Figure 1). This provides sufficient motivation for attempting a synthesis-oriented approach to achieve system implementations having both hardware and software components. Such an approach would benefit from a systematic analysis of design trade-offs that is com-

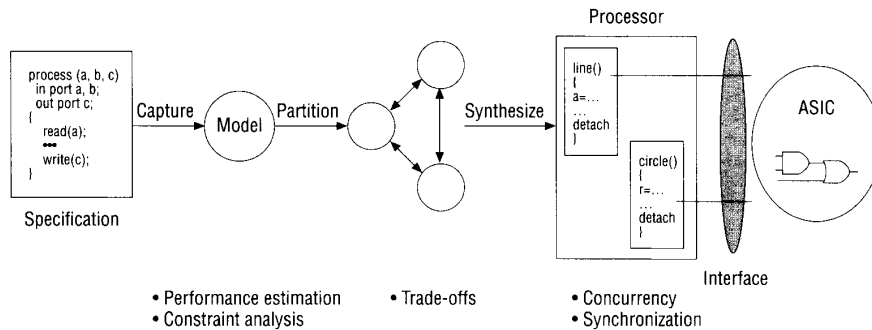


Figure 4. Synthesis approach to embedded systems.

mon in synthesis while also creating cost-effective systems.

One way to accomplish this task is to specify constraints on cost and performance of the resulting implementation (Figure 3). We present an approach to systematic exploration of system designs that is driven by such constraints. Our work builds upon high-level synthesis techniques for digital hardware⁴ by extending the concept of a resource needed for implementation.

As shown in Figure 4, this approach captures a behavioral specification into a system model that is partitioned for implementation into hardware and software. We then synthesize the partitioned model into interacting hardware and software components for the target architecture shown in Figure 5. The target architecture uses one processor that is embedded with an application-specific hardware component. The processor uses only one level of memory and address space for its instructions and data. Currently, to simplify the synthesis and performance estimation for the hardware component, we do not pipeline the application-specific hardware. Even with its relative simplicity, the target architecture can apply to a wide class of applications in embedded systems.

Among the related work, Woo, Wolf, and Dunlop⁵ investigate implementing hardware or software from a cospecifi-

cation. Chou, Ortega, and Borriello⁶ describe synthesis of hardware or software for interface circuits. Chiodo et al.⁷ discuss a methodology for generating hardware and software based on a unified finite-state-machine-based model. Given a system specification as a C-program, Henkel and Ernst⁸ identify portions of the program that can be implemented into hardware to achieve a speedup of overall execution times. Srivastava and Brodersen⁹ and Buck et al.¹⁰ present frameworks for generating hardware and software components of a system. Investigators have proposed several new architectures that use field-programmable gate arrays to create special-purpose coprocessors to speed up applications (PAM¹¹, MoM¹²) or to create prototypes (QuickTum¹³).

Capturing specification of system functionality and constraints

We capture system functionality using a hardware description language, *HardwareC*.¹⁴ The cosynthesis approach formulated here does not depend upon the particular choice of the HDL, and could use other HDLs such as VHDL or Verilog. However, the use of *HardwareC* leverages the use of Olympus tools developed for chip-level synthesis.⁴

HardwareC follows much of the syntax and semantics of the programming language, with modifications necessary for correct and unambiguous hardware

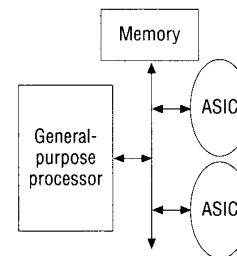


Figure 5. Target architecture.

modeling. *HardwareC* description consists of a set of interacting processes that are instantiated into blocks using a declarative semantics. A process model executes concurrently with other processes in the system specification. A process restarts itself on completion. Operations within a process body allow for nested concurrent and sequential operations.

Figure 6 shows an example of an HDL functionality specification. This example performs two data input operations, followed by a conditional in which a counter index is generated. The specification uses counter index *z* to seed a down-counter indicated by the while loop. A graph-based representation as shown captures this HDL specification.

In general, the system model consists of a set of hierarchically related sequencing graphs. Within a graph, vertices represent language-level operations and edges represent dependencies be-

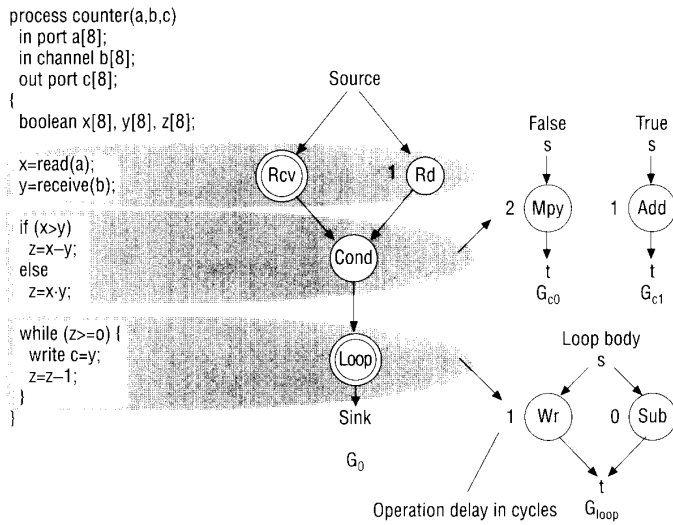


Figure 6. Example of input specification and capture.

tween the operations. Such a representation makes explicit the concurrency inherent in the input specification, thus making it easier to reason about properties of the input description. As we shall soon see, it also allows us to analyze timing properties of the input description.

Model properties. The sequencing graph is a polar one with source and sink vertices that represent no-operations. Associated with each graph model is a set of variables that defines the shared memory between operations in the graph model. Source and sink vertices synchronize executions of operations in a graph model across multiple iterations. Thus, polarity of the graph model ensures that there is exactly one execution of an operation with respect to each execution of any other operation. This makes execution of operations within a graph single rate (Figure 7). The set of variables associated with a graph model defines the storage common to the operations; it serves to facilitate communication between operations.

Given the single-rate execution model, it is relatively straightforward to ensure

ordering of operations in a graph model that preserves integrity of memory shared between operations. However, operations across graph models follow multirate execution semantics. That is, there may be variable numbers of executions of an operation for an operation in another graph model. Because of this multirate nature of execution, the operations use message-passing primitives like send and receive to implement communications across graph models. Use of these primitives simplifies specification of inter-model communications. A multirate specification is an important feature for modeling heterogeneous systems, because the processor and application-specific hardware may run on different clocks and speeds.

HDL descriptions contain operations to represent synchronization to external events, such as the receive operation, as well as data-dependent loop operations. These operations, called nondeterministic delay (ND) operations, present unknown execution delays. The ability to model ND operations is vital for reactive embedded system descriptions. Figure 6 indicates ND operations with double circles.

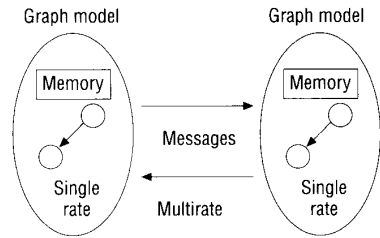


Figure 7. Properties of the graph model.

A system model may have many possible implementations. Timing constraints are important in defining specific performance requirements of the desired implementation. As shown in Figure 8, timing constraints are of two types:

- **Min/max delay constraints:** These provide bounds on the time interval between initiation of execution of two operations.
- **Execution rate constraints:** These provide bounds on successive initiations of the same operation. Rate constraints on input/output operations are equivalent to constraints on throughput of respective inputs/outputs.

These two types of constraints are sufficient to capture constraints needed by most real-time systems.¹⁵ Our synthesis system captures minimum delay constraints in the graphical representation by providing weights on the edges to indicate delay of the corresponding source operation. Capturing maximum delay constraints requires additional backward edges (Figure 9).

Model analysis. Having captured system functionality and constraints in a graphical model, we can now estimate system performance and verify the consistency of specified constraints. Performance measures require estimation of operation delays. We compute these delays separately for hardware and software implementations based on the

type of hardware to be used and the processor used to run the software. A processor cost model captures processor characteristics. It consists of an execution delay function for a basic set of processor operations, a memory address calculation function, a memory access time, and processor interruption response time.

Timing constraint analysis attempts to answer the following question: Can imposed constraints be satisfied for a given implementation? We indicate an implementation of a model by assigning appropriate delays to the operations with known delays (not ND) in the graph model. Constraint satisfiability relates to the structure as well as the actual delay and constraint values on the graph. Some structural properties of the graphs (relating to ND operations and their dependencies) may make a constraint unsatisfiable regardless of the actual delay values of the operations. Further, some constraints may be mutually inconsistent: for example, a maximum delay constraint between two operations that also have a larger minimum delay constraint. No assignment of nonnegative operation delay values can satisfy such constraints.

In the presence of ND operations in a graph model, we consider a timing constraint satisfiable if it is satisfied for all possible (and maybe infinite) delay values of the ND operations. We consider a timing constraint marginally satisfiable if it can be satisfied for all possible values within specified bounds on the delay of the ND operations. Marginal satisfiability analysis is useful because it allows the use of timing constraints that can be satisfied under some implementation assumptions (acceptable bounds on ND operation delays). Without these assumptions the general timing constraint satisfiability analysis would otherwise consider these constraints ill-posed.¹⁶

We perform timing constraint analysis by graph analysis on the weighted sequencing graphs. Consider first the case

where the graph model does not contain any ND operations. Here, we can label every edge in the graph with a finite and known weight. In such a graph, we cannot satisfy a min/max delay constraint if a positive cycle in the graph model exists.¹⁶ Next, in the presence of ND operations, timing constraints are satisfiable if no cycles containing ND operations exist. For a cycle containing an ND operation, it is impossible to determine satisfiability of timing constraints, and only marginal satisfiability can be guaranteed. As we will see, it is possible to break the cycle by graph transformations that preserve the HDL program semantics.

For nonpipelined implementations, we can treat rate constraints as min/max delay constraints between corresponding source and sink operations of the graph model. Thus we can apply the above min/max constraint satisfiability criterion to the analysis of rate constraints.

Note that in some cases system throughput (specified by rate constraints) can be optimized significantly with little or no impact on system latency by using a pipelined execution model and extra resources. Indeed, for deterministic and fixed-rate systems particularly used for digital signal processing applications, researchers have developed extensive transformations that determine and achieve bounds on system throughput.¹⁷ However, as noted earlier, systems modeled by the sequencing graphs generally operate at different rates. In addition, because of the presence of ND operations due to loops, the rate at which a particular operation executes may change over time. While this property is essential for modeling control-dominated embedded systems, it aggravates the problem of determining absolute bounds on achievable system throughput.

We illustrate the issue of rate constraints on graphs containing ND operations in Example A (next page).

In general, consider a process P that contains an ND operation due to an un-

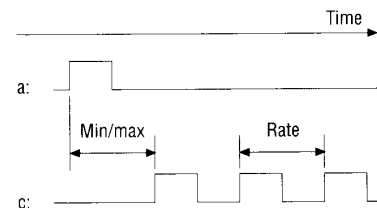


Figure 8. Timing constraints.

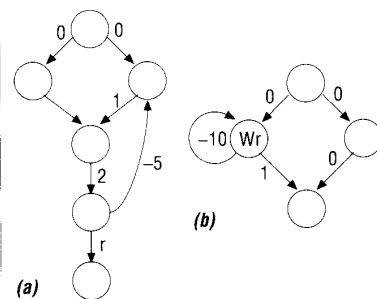


Figure 9. Representation of timing constraints: min/max constraint (a), rate constraint (b).

bounded loop. The ND operation induces a bipartition of the calling process, $P = F \cup B$, such that the set of operations in F (for example, the read operation in process test) must be performed before invoking the loop body. Further, the set of operations in B can only be performed after completing executions of the loop body. We can then use functional pipelining of F, B , and the loop to improve the reaction rate of P . Since we assume nonpipelined hardware, these transformations are used only in the context of the software component.

Constraint analysis and software.

The linear execution semantics imposed by the software running on a single-processor target architecture complicates constraint analysis for a software implementation of a graph model. That is, performing delay analysis for software operations requires a complete order of operations in the graph model. In creat-

Example A

Consider the following process fragment

```

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

Here, v is a Boolean array that represents an integer. In the presence of rate constraint r on the *read* operation, the constraint graph has a cycle containing an ND operation relating to the unbounded *while* loop operation. Note that the rate constraint corresponds to directed edge from sink t to source s in the graph of Figure A.

The overall execution time of the *while* loop determines the interval between

successive executions of the read operation. Due to this variable-delay loop operation, the input rate at port p is variable and cannot always be guaranteed to meet the required rate constraint. In general, determining achievable throughput at port p is difficult. As we explain next, marginal satisfiability of the rate constraint can be ensured by graph transformations and by using a finite-size buffer.

Figure A shows the sequencing graph model P corresponding to process test. Identifier rd refers to the read operation, lp refers to the while loop operation. Symbols $P1, P2$, and so forth in the execution trace below indicate successive invocations of the process test. $L1, L2, L3$, and $L4$ indicate multiple invocations of the lp operation. Depending on the side effects produced by the loop-body, the original graph P can be transformed into fragments Q and R such that executions of Q and R can overlap to improve the throughput of the read operation in Q . Data transfers from Q to R by means of a buffer. See Example B on page 37 for a consideration of a software implementation of P .

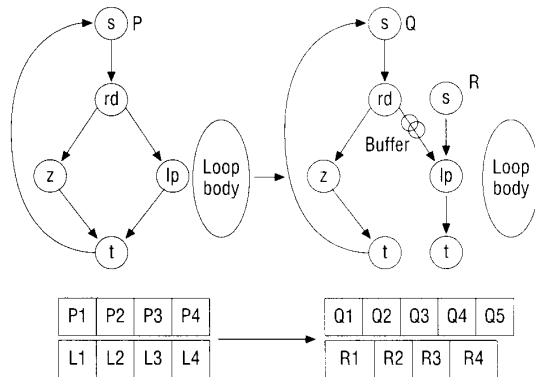


Figure A. Breaking ND cycle by graph transformation.

ing a complete order of operations, it is likely that unbounded cycles may be created, which would make constraints unsatisfiable.

As shown in Figure 10, any serialization that puts an ND operation between two operations $op1$ and $op2$ will make any maximum delay constraint between $op1$

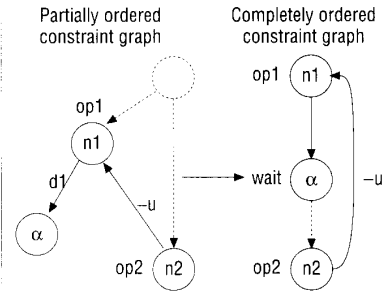


Figure 10. Linearization in software leads to creation of unsatisfiable timing constraints. Constraint maxtime from $op1$ to $op2 = u$ cycles.

and $op2$ unsatisfiable. However, note that while all computations must be performed serially in software, communication operations can proceed concurrently. In other words, it is possible to overlap execution of ND operations (wait for synchronization or communication) with some (unrelated) computation. But such an overlap requires the ability to schedule operations dynamically in software since the simultaneously active ND operations may complete in orders that cannot be determined statically.

Typically, dynamic scheduling of operations involves delay overheads due to selection and scheduling of operations. Therefore, a good model of software is to think of software as a set of fixed-latency concurrent threads (Figure 11). We define a thread as a linearized set of operations that may or may not begin by an ND operation indicated by a circle in Figure 11. Other than the beginning ND operation, a thread does not contain any ND operations. We consider the delay of the initial ND operation part of the scheduling delay and, therefore, not included in the latency of the program thread. Use of multiple concurrent program threads instead of a single program to implement the software also avoids the need for complete serialization of all operations that may create un-

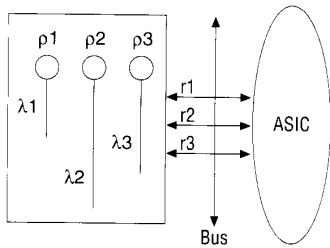


Figure 11. Software model to avoid creation of ND cycles.

bounded cycles.

In this software model, we can check marginal satisfiability of constraints on operations belonging to different threads, assuming a fixed and known delay of scheduling operations associated with ND operations (context switch delay, for example).

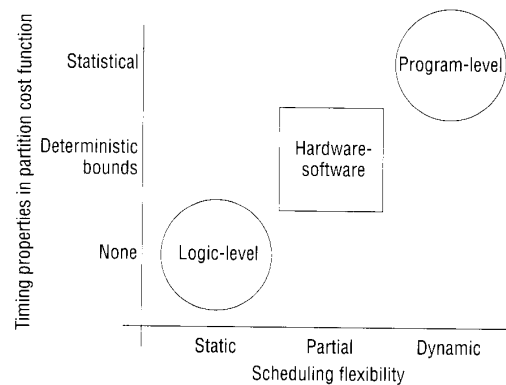
System partitioning

The system-level partitioning problem refers to the assignment of operations to hardware or software. The assignment of an operation to hardware or software determines the delay of the operation. In addition, assignment of operations to a processor and to one or more application-specific hardware circuits involves additional delays due to communication overheads.

Any good partitioning scheme must attempt to minimize this communication. Further, as operations in software are implemented on a single processor, increasing operations in software increases processor utilization. Consequently, overall system performance depends on the effect of hardware-software partition on utilization of the processor and the bandwidth of the bus between the processor and application-specific hardware.

A partitioning scheme thus must attempt to capture and make use of a partition's effect on system performance in making trade-offs between hardware and software implementations of an operation. An efficient way to do this would be to devise a partition cost func-

Figure 12. Use of timing properties in partition cost function.



tion that captures these properties. We would then use this function to direct the partitioning algorithm toward a desired solution, where an optimum solution is defined by the minimum value of the partition cost function.

Note that we need to capture not only the effects of sizes of hardware and software parts but also the effect on timing behavior of these portions in our partition cost function. In contrast, most partitioning schemes for hardware have focused on optimizing area and pinout of resulting circuits. Capturing the effect of a partition on timing performance during the partitioning stage is difficult. Part of the problem arises because the timing properties are usually global in nature, thus making it difficult to make incremental computations of the partition cost function as is essential for developing effective partition algorithms. Approximation techniques have been suggested to take into account the effect of a partition on overall latency.¹⁸

Note, however, that partitioning in the software world does make extensive use of statistical timing properties to drive the partitioning algorithms.¹⁹ We draw the distinction between these two extremes of hardware and software partitioning by the flexibility to schedule operations. Hardware partitioning attempts to divide circuits that implement scheduled opera-

tions. Conversely, the program-level partitioning addresses operations that are scheduled at runtime.

Our approach to partitioning for hardware and software takes an intermediate approach. As shown in Figure 12, we use deterministic bounds on timing properties that are incrementally computable in the partition cost function. That is, we can compute the new partition cost function in constant time. We accomplish this by using a software model in terms of a set of program threads as shown in Figure 11 and a partition cost function, f , that is a linear combination of its variables. The following properties characterize this software component:

- **Thread latency** λ_i (seconds) indicates the execution delay of a program thread.
- **Thread reaction rate** ρ_i (per second) is the invocation rate of the program thread.
- **Processor utilization** P is calculated by

$$P = \sum_{i=1}^n \lambda_i \cdot \rho_i$$

- **Bus utilization** B (per second) is the total amount of communication taking place between the hardware and software. For a set of m vari-

ables to be transferred between hardware and software,

$$B = \sum_{j=1}^m r_j$$

r_j is the inverse of the minimum time interval (in seconds) between two consecutive samples for variable j , which is marked for destination to one of the program threads.

Characterization of software using λ , ρ , P , and B parameters makes it possible to calculate static bounds on software performance. Use of these bounds is helpful in selecting an appropriate partition of system functionality between hardware and software. However, it also has the disadvantage of overestimating performance parameters such as processor and bus bandwidth utilization. Typically, there is a distribution of thread invocations and communications based on actual data values being transferred, which is not accounted for in these parameters.

We compute hardware size S_H bottom-up from the size estimates of the resources implementing the individual operations. In addition, we characterize the interface between hardware and software by a set of communication ports (one for each variable) between hardware and software that communicate data over a common bus. The overhead due to communication between hardware and software is manifested by the utilization of bus bandwidth as described earlier.

Given the cost model for software, hardware, and interface, we can informally state the problem of partitioning a specification for implementation into hardware and software as follows:

From a given set of sequencing graph models and timing constraints between operations, create two sets of sequencing graph models such that one can be implemented in hardware and the other in software and the following is true:

- Timing constraints are satisfied for the two sets of graph models.
- Processor utilization, $P \leq 1$.
- Bus utilization, $B \leq \bar{B}$.
- A partition cost function, $f = f(S_H, B, P^{-1}, m)$ is minimized.

An exact solution to the constrained partitioning problem—a solution that minimizes the partition cost function—requires that we examine a large number of solutions. Typically, that number is exponential to the number of operations under partition. As a result, designers often use heuristics to find a “good” solution, with the objective of finding an optimal value of the cost function that is minimal for some local properties.

Most common heuristics to solving partitioning problems start with a constructive initial solution that some iterative procedure can then improve. Iterative improvement can follow, for example, from moving or exchanging operations and paths between partitions. A good heuristic is also relatively insensitive to the initial solution. Typically, exchange of a larger number of operations makes the heuristic more insensitive to the starting solution, at the cost of increasing the time complexity.

In the following, we describe the intuitive features of the partitioning algorithm. We have presented details elsewhere.²⁰ The procedure identifies operations that can be implemented in software such that the corresponding constraint graph implementation can be satisfied and the resulting software (as a set of program threads) meets required rate constraints on its inputs and outputs. As an initial partition we assume that ND operations related to data-dependent loop operations define the beginning of program threads in software, while all other operations are implemented in hardware. The rate constraints on software inputs/outputs translate into bounds on required reaction rate ρ_i of corresponding program thread T_i . Maximum achievable reac-

tion rate $\bar{\rho}_i$ of a program thread is computed as the inverse of its latency. The latency of a program thread is computed using a processor delay cost model and includes a fixed scheduling overhead delay.

From an initial solution we perform iterative improvement by migrating operations between the partitions. Migration of an operation across a partition affects its execution delay. It also affects the latency and reaction rate of the thread to which this operation is moved. We similarly compute its effect on processor and bus bandwidth utilization. At any step, we select operations for migration so that the move lowers the communication cost, while maintaining timing constraint satisfiability. In addition, we check for communication feasibility by verifying that $\bar{\rho}_i \geq \rho_i$ for each thread, and that processor and bus utilization constraints are satisfied.

System synthesis

From partitioned graph models, our next problem is to synthesize individual hardware and software components. Ku¹⁴ and others^{1,2} address in detail the generation of hardware circuits for sequencing graph models. Therefore, we concentrate on generation of software and interface circuitry from partitioned models. The problem of software synthesis is to generate a program from partitioned graph models that correctly implements the original system functionality. We assume that the resulting program is mapped to real memory, so the issues related to memory management are not relevant to this problem. The partitioning discussed previously identified graph models that are to be implemented in hardware and operations (organized as program threads) that are to be implemented in software. See Example B.

The program generation from a thread can either use a coroutine or subroutine scheme. Since, in general, there can be dependencies into and from the program threads, a coroutine model is

Example B

We can implement the process test shown in Example A as following two program threads in software.

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

In its software implementation of process test, thread T1 performs the reading operations, and thread T2 consists of operations in the body of the loop. For each execution of thread T1 there are v executions of thread T2.

more appropriate. A dependency between two operations can be either a data or a control dependency. Depending upon predecessor relationships and timing of the operations, we can make some of these redundant by inserting other dependencies such that resulting program threads are convex—all external dependencies are limited to the first and last operations.

For a given subgraph corresponding to a program thread, we can move an incoming data dependency up to its first operation and move an outgoing data dependency down to its last operation. This procedure produces a potential loss of concurrency. However, it makes the task of routine implementation easier since we can implement all the routines as independent programs with statically embedded control dependencies.

Rate constraints and software. In the presence of dependencies on ND operations, we cannot always guarantee that a given software implementation will meet the data rate constraints on its

Example C

Consider the threads T1 and T2 generated from process test mentioned in Example A. The overall execution time of the while loop determines the interval between successive executions of the read operation. Due to this variable-delay loop operation, the input rate at port p is variable so we cannot always guarantee the reaction rate of T1. Since the set of operations in loop-body may alter the contents of memory in process test, thread T1 must be blocked until the completion of T2. Thus the process test can be thought of as consisting of two parallel processes, as shown in Figure B. We need the first operation of thread T2, wait1, to observe the data dependency of operations in thread T2. We need the second wait operation, wait2, to guarantee that any memory side effects of T2 for variables in T1 are correctly reflected. To obtain a deterministic bound on the reaction rate of the calling thread, it is possible to unroll the looping thread by creating a variable number of program threads. However, in this case each iteration of the looping thread would carry scheduling overhead. Dynamic creation of program threads may also lead to violation of processor utilization constraint as described in previous sections.

However, it is possible to overlap execution of loop thread T2 with execution of thread T1, and to ensure marginal timing constraint satisfiability. Note that we can remove operation

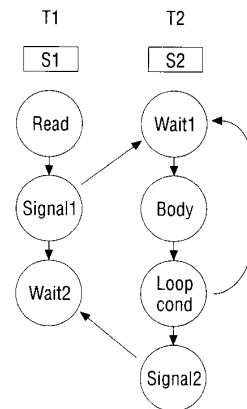


Figure B. Dependence of a program thread on a program thread corresponding to a loop.

wait2 if the looping thread does not produce any side effect on storage S1 of the calling thread. That is, the loop body only reads and does not modify the variables common to S1 and S2. In such cases we can use data buffers between program threads to maintain the reaction rate of a program thread. For implementation details, see Gupta, Coelho, and De Micheli.¹

Reference

1. R.K. Gupta, C. Coelho, and G. De Micheli, "Program Implementation Schemes for Hardware-Software Systems," *Notes of Int'l Workshop Hardware-Software Codesign*, Oct. 1992, and CSL Tech. Report TR-92-548, Stanford University, Stanford, Calif., 1992.

I/O ports. In case of synchronization-related ND operations, we can check for marginal satisfiability of timing constraints by assigning a context-switch delay to the respective wait operations. However, in the case of unbounded loop-related ND operations, the delay due to these operations consists of ac-

tive computation time. Marginal timing satisfiability analysis therefore requires that we estimate loop index values. We illustrate this in Example C.

Hardware-software interface. Because of the serial execution of the software component, a data transfer from

Example D

Consider the mixed implementation of a graphics controller that contains two threads for generation of line and circle coordinates in software as shown in Figure C. The interface protocol using control FIFO is specified as follows:

```
queue [2] controlFIFO [1];
queue [16] line_queue [1], circle_queue [1];

when ((line_queue.dequeue_rq+ & !line_queue.empty) & !controlFIFO.full) do
controlFIFO enqueue #2;
when ((circle_queue.dequeue_rq+ & !circle_queue.empty) & !controlFIFO.full)
do controlFIFO enqueue #1;
when (controlFIFO.dequeue_rq+ & !controlFIFO.empty) do controlFIFO de-
queue dlx.0xff000[1:0];
```

In this example, two data queues with 16 bits of width and 1 bit of depth, *line_queue* and *circle_queue*, and one queue with 2 bits of width and 1 bit of depth, *controlFIFO*, are declared. The guarded commands specify the conditions on which the number 1 or the number 2 is enqueued—here, a '+' after a signal name means a positive edge and a '-' after the signal means a negative edge. The first when condition states that when a dequeue request for the queue *line_queue* arrives and this queue is not empty and the queue *controlFIFO* is not full, then enqueue the value 2 (representing identifier for a corresponding program thread that consumes data from the line queue) into the *controlFIFO*.

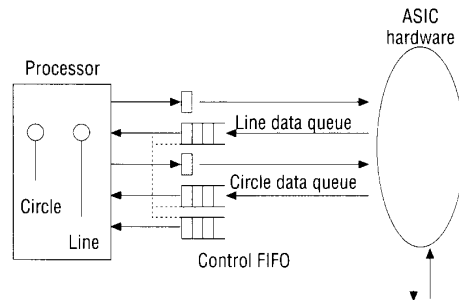


Figure C. Mixed implementation.

hardware to software must be explicitly synchronized. By using a polling strategy, we can design the software component to perform premeditated transfers from the hardware components based on its data requirements. This requires static scheduling of the hardware component. Where software functionality is limited

by communications—that is, where the processor is busy waiting for an input-output operation most of the time—such a scheme would suffice. Further, in the absence of any unbounded-delay operations, we can simplify the software component in this scheme to a single program thread and a single data chan-

nel since all data transfers are serialized. However, this approach would not support any branching nor any reordering of data arrivals, since the design would not support dynamic scheduling of operations in hardware.

To accommodate differing rates of execution among the hardware and software components, and due to unbounded delay operations, we look for a dynamic scheduling of different threads of execution. Availability of data forms the basis for such a scheduling. One mechanism to perform such scheduling is a control FIFO (first in, first out) buffer, which attempts to enforce the policy that data items are consumed in the order in which they are produced. As shown in Example D, the hardware-software interface consists of data queues on each channel and a control FIFO that holds the identifiers for the enabled program threads in the order in which their input data arrives. The control FIFO depth equals the number of threads of execution, since a thread execution stalls pending availability of the requested data.

Note that thread scheduling by means of a control FIFO does not explicitly prioritize the program threads. This is because, for safety reasons, the control FIFO serves program threads strictly in the order in which their identifiers are enqueued. In some systems we may want to invoke a program thread as soon as its needed data becomes available. Such systems would be better served by a preemptive scheduling algorithm based on relative priorities of the threads. However, preemption comes at significant operating system overhead. In contrast, nonpreemptive prioritized scheduling of program threads is possible with relatively minor modifications to control FIFO. Example E describes the actual interconnection schematic between hardware and software for a single data queue.

We can implement the control FIFO and associated control logic either in hardware as a part of the ASIC compo-

nent or in software. If we implement the control FIFO in software, the system no longer needs the FIFO control logic since the control flow is already in software. In this case, the q_rq lines from data queues connect to processor unvectored interruption lines, where the system uses respective interruption service routines to enqueue the thread identifier tags into the control FIFO. During the enqueue operations the system disables the interruptions to preserve integrity of the software control flow.

Example

As an experiment in achieving mixed system designs, we attempted synthesis of an Ethernet-based network coprocessor. The coprocessor is modeled as a set of 13 concurrently executing processes that interact with each other by means of 24 send and 40 receive operations. The total description consists of 1,036 lines of HDL code. A hardware-software implementation of the coprocessor takes 8,572 bytes of program and data storage for a DLX processor²¹ and 8,394 equivalent gates using an LSI Logic 10K library of gates.

We can thus build the mixed implementation using only one ASIC chip plus an off-the-shelf processor. A complete hardware implementation would require use of a custom chip or two ASIC chips. More importantly, we can guarantee that the mixed solution using a DLX processor running at 10 MHz will meet the imposed performance requirements of a maximum propagation delay of 46.4 μs, a maximum jam time of 4.8 μs, a minimum interframe spacing of 67.2 μs, and an input bit-rate of 10 Mbytes/s.

SYNTHESIS OF EMBEDDED REAL-TIME systems from behavioral specifications constitutes a challenging problem in hardware-software cosynthesis. Due to the relative simplicity of the target architecture compared to general-purpose

Example E

Figure D shows schematic connection of the FIFO control signals for a single data queue. In this example, the data queue is memory mapped at address 0xee000 while the data queue request signal is identified by bit 0 of address 0xee004 and enable from the microprocessor (up_en) is generated from bit 0 of address 0xee008. The following describes the FIFO and microprocessor connections. *cntc* refers to a data queue associated with the circle drawing program threads. *mp* refers to a model of the microprocessor. A signal name is prefixed with a period to indicate the associated hardware or software model.

```
cntc.rq_line [0:0] = @ mp.0xee004[0:0];      # request
cntc.en_line [0:0] = mp.0xee008[0:0];      # enable up_en
cntc.ab_line [0:0] = mp.0xee000_rd;        # absorb up_ack
```

The control logic needed to generate the enqueue is described by a simple state transition diagram shown in Figure E. The control FIFO is ready to enqueue (indicated by *gn = 1*) process id if the corresponding data request (*q_rq*) is high and the process has enabled the thread for execution (*up_en*). Signal *up_ab* indicates completion of a control FIFO read operation by the processor.

In case of multiple in-degree queues, the enqueue_rq is generated by OR-ing the requests of all inputs to the queues. In case of multiple-out-degree queues, the signal dequeue_rq is generated also by OR-ing all dequeue requests from the queue.

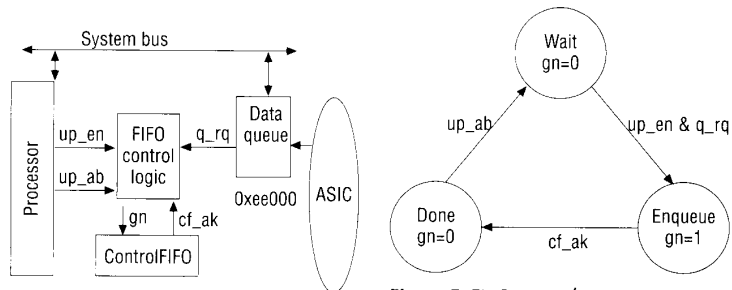


Figure D. Control FIFO schematic.

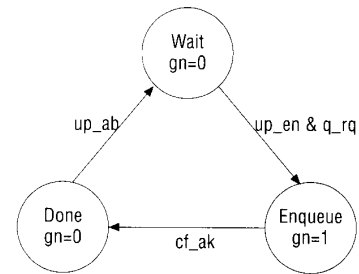



Figure E. FIFO control state transition diagram.

computing systems, it also affords an opportunity in computer-aided design, by which we can automatically synthesize such systems from a unified specification. Further, the ability to perform constraint and performance analysis for such systems provides a major motivation for using the synthesis approach instead of design-oriented implementation approaches.

Even when manually designed, such systems can benefit greatly from prototypes created by a cosynthesis approach. A cosynthesis approach lets us reduce the size of the chip-synthesis task, while meeting the performance constraints, such that we can use field- or mask-programmable hardware to provide fast turnaround on complex system designs.

For hardware-software synthesis to be

effective, we need specification languages that capture and use capabilities of both hardware and software. The approach presented in this article makes use of an HDL to formulate the problem of cosynthesis as an extension of hardware synthesis. In the process, the approach makes many simplifications for the generated software and leaves room for considerable optimization of the software component.

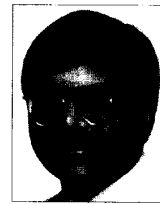
Currently, we are attempting to develop transformations to simplify control flow in the sequencing graph models, which we can use to minimize interface synchronization requirements. We also plan to investigate extensions to the target architecture to include hierarchical memory schemes and multiple processors. 

Acknowledgments

We acknowledge discussions and contributions by Claudionor Coelho and David Ku. This research was sponsored by NSF-ARPA, under grant MIP 9115432, and by a fellowship provided by Philips at the Stanford Center for Integrated Systems.

References

1. *Silicon Compilation*, D. Gajski, ed., Addison Wesley, Reading, Mass., 1988.
2. *High-Level VLSI Synthesis*, R. Camposano and W. Wolf, eds., Kluwer Academic Publishers, Norwell, Mass., 1991.
3. D.C. Luckham, "Partial Ordering of Event Sets and Their Application to Prototyping Concurrent Timed Systems," *J. Systems and Software*, July 1993.
4. G. De Micheli et al., "The Olympus Synthesis System for Digital Design," *IEEE Design & Test of Computers*, Vol. 7, No. 5, Oct. 1990, pp. 37-53.
5. N. Woo, W. Wolf., and A. Dunlop, "Compilation of a Single Specification Into Hardware and Software," *Notes of Int'l Workshop Hardware-Software Codesign*, Oct. 1992.
6. P. Chou, R. Ortega, and G. Borriello, "Synthesis of the Hardware/Software Interface in Microcontroller-Based Systems," *Proc. Int'l Conf. Computer-Aided Design*, IEEE Computer Society Press, Los Alamitos, Calif., 1992, pp. 488-495.
7. M. Chiodo et al., "Synthesis of Mixed Hardware-Software Implementations from CFSM Specifications," Memo UCB/ERL M93/49, June 1993, Univ. of California at Berkeley, and *Notes of Int'l Workshop on Hardware-Software Codesign*, Oct. 1992.
8. J. Henkel and R. Ernst, "Ein Softwareorientierter Ansatz zum Hardware-Software CoEntwurf" [A Software-oriented Approach to Hardware-Software Codesign], *Proc. ITG Conf., Recnergestuetzter Entwurf und Architektur mikroelektronischer Systeme*, Darmstadt, Germany, 1992, pp. 267-268.
9. M.B. Srivastava and R.W. Brodersen, "Rapid-Prototyping of Hardware and Software in a Unified Framework," *Proc. Int'l Conf. Computer-Aided Design*, IEEE CS Press, 1991, pp. 152-155.
10. J. Buck et al., "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," to be published in *Int'l J. Computer Simulations*.
11. P. Bertin, D. Roncin, and J. Vuillemin, "Introduction to Programmable Active Memories," in *Systolic Array Processors*, J. McCanny, J. McWhirter, and E. Swartzlander, Eds., Prentice Hall, New York, 1989, pp. 300-309.
12. R.W. Hartenstein, A.G. Hirschbiel, and M. Weber, "Mapping Systolic Arrays Onto the Map-Oriented Machine," in *Systolic Array Processors*, J. McCanny, J. McWhirter, and E. Swartzlander, eds., Prentice Hall, New York, 1989, pp. 300-309.
13. S. Walters, "Reprogrammable Hardware Emulation Automates System-Level ASIC Validation," *Wescon/90 Conf. Records*, Electron. Conventions Mgt., Nov. 1990, pp. 140-143.
14. D. Ku and G. De Micheli, *High-Level Synthesis of ASICs Under Timing and Synchronization Constraints*, Kluwer Academic Publishers, Norwell, Mass., 1992.
15. B. Dasarathy, "Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Method for Validating Them," *IEEE Trans. Software Engineering*, Vol. SE-11, No. 6, Jan. 1985, pp. 80-86.
16. D. Ku and G. De Micheli, "Relative Scheduling Under Timing Constraints: Algorithms for High-level Synthesis of Digital Circuits," *IEEE Trans. CAD/ICAS*, Vol. 11., No. 6, June 1992, pp. 696-718.
17. K.K. Parhi, "Algorithm Transform for Concurrent Processors," *Proc. IEEE*, Dec. 1989, IEEE Press, Piscataway, N.J., pp. 1879-1985.
18. R.K. Gupta and G. De Micheli, "Partitioning of Functional Models of Synchronous Digital Systems," *Proc. Int'l Conf. Computer-Aided Design*, IEEE CS Press, 1990, pp. 216-219.
19. V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*, MIT Press, Cambridge, Mass, 1989.
20. R.K. Gupta and G. De Micheli, "System-Level Synthesis Using Re-programmable Components," *Proc. European Design Automation Conf.*, IEEE CS Press, 1992, pp. 2-7.
21. J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman Publishers, Palo Alto, Calif., 1990, pp. 88-137.



Rajesh K. Gupta is a doctoral student in the Department of Electrical Engineering at Stanford University. His primary research interests are the design and synthesis of VLSI circuits and systems. Gupta received an MS in electrical engineering and computer science from the University of California, Berkeley, and a BTech in electrical engineering from the In-

dian Institute of Technology in Kanpur. Earlier he worked on VLSI design at various levels of abstraction as a member of the design teams for the 80386-SX, 486, and Pentium microprocessor devices at Intel. He is coauthor of a patent on a PLL-based clock circuit, and is currently a Phillips fellow at the Center for Integrated Systems at Stanford.



Giovanni De Micheli is an associate professor of electrical engineering and computer science at Stanford University. His research interests include several aspects of the computer-aided design of integrated circuits with particular emphasis on automated synthesis, optimization, and verification of VLSI circuits. He is coeditor of *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation*, and coauthor of *High-Level Synthesis of ASICs Under Timing and Synchronization Constraints*. He graduated from the Politecnico di Milano with a degree in nuclear engineering and received a PhD in electrical engineering and computer science from the University of California, Berkeley. De Micheli is a senior member of the IEEE and is associate editor of the *IEEE Proceedings*, the *IEEE Transactions on VLSI Systems*, and *Integration: The VLSI Journal*.

Send correspondence about this article to the authors at the Center for Integrated Systems, CIS 18, Stanford University, Stanford, CA 94305; rgupta@momus.stanford.edu.

SEPTEMBER 1993

IEEE COMPUTER SOCIETY PRESS

VLSI ALGORITHMS AND ARCHITECTURES Fundamentals

edited by N. Ranganathan

This first book introduces basic approaches to the design of VLSI algorithms and architectures and provides a reliable reference source for advanced readers. It addresses introductory and fundamental topics related to VLSI algorithms and architectures and provides a concise tutorial on the subject. The chapters in this volume:

- ◆ Introduce Basic Concepts and Discuss Various Issues Related to the Design of VLSI Algorithms and Architectures
- ◆ Present Papers on Systolic and Wavefront Arrays
- ◆ Focus on VLSI Implementation of Data Structures and Sorting
- ◆ Deal with VLSI Structures for Matrix and Algebraic Computations
- ◆ Address Important Application Areas

Sections: An Overview of VLSI Algorithms and Architectures, Systolic and Wavefront Arrays, Data Structures and Sorting, Matrix and Algebraic Computations, Pattern Matching and Text Retrieval, VLSI Processor Designs.

320 pages. July 1993. Hardcover. ISBN 0-8186-4392-7.
Catalog # 4392-01 — \$40.00 Members \$32.00

VLSI ALGORITHMS AND ARCHITECTURES Advanced Concepts

edited by N. Ranganathan

This companion volume features an in-depth examination into the latest designs of VLSI algorithms and architectures for the engineering community. It contains many new studies and elaborates on various computationally intensive problems requiring VLSI solutions. It also addresses advanced techniques and VLSI architectures for a broad range of application areas.

The first chapter discusses important architectural design issues as well as the realization of these architectures as VLSI systems. It discusses design issues such as layout methodology, processor synchronization, area-time trade-offs, and performance. The next chapter focuses on advanced concepts for systolic arrays and algorithms for the automatic synthesis of systolic arrays. The subsequent chapters describe special-purpose architectures for a wide range of computationally intensive problems. They discuss special-purpose architectures; VLSI chips for problems in image and speech processing, AI, and vision applications; application issues for dictionary machines and data compression; and hardware architectures for iterative algorithms.

Sections: VLSI Architecture Design Issues; Advanced Topics in Systolic Arrays; Image, Speech, and Signal Processing; Artificial Intelligence and Computer Vision; Dictionary Machines and Data Compression; Iterative Algorithms.

320 pages. July 1993. Hardcover.
ISBN 0-8186-4402-8.
Catalog # 4402-01 — \$40.00 Members \$32.00

To order call toll-free
1-800-CS-BOOKS

in CA — 714/ 821-8380 ◆ FAX — 714/ 821-4641

