

Hardware-software Co-synthesis for Digital Systems

Rajesh K. Gupta

Giovanni De Micheli

Computer Systems Laboratory

Stanford University, Stanford, CA 94305.

Abstract

As the complexity of system design increases, use of pre-designed components, such as general-purpose microprocessors, provides an effective way to reduce the complexity of synthesized hardware. While the design problem of systems that contain processors and ASIC chips is not new, computer-aided synthesis of such *heterogeneous* or mixed systems poses challenging problems because of the differences in model and rate of computation by application-specific hardware and processor software. In this article, we demonstrate the feasibility of achieving synthesis of heterogeneous systems which uses timing constraints to delegate tasks between hardware and software such that the final implementation meets required performance constraints.

1 Introduction

Most digital systems used for dedicated applications consist of general-purpose processors, memory and application-specific hardware circuits. Examples of such *embedded* systems can be found in medical instrumentation, process control, automated vehicle and networking and communication systems. In addition to being application-specific, such systems are also designed to respect constraints related to relative timing of their actions, hence these are referred to as *real-time embedded systems*.

Design and analysis of real-time embedded systems poses challenges in performance estimation, selection of appropriate parts for system implementation and verification of such systems for functional and temporal properties. In practice, such systems are implemented from their specification as a set of loosely-defined functionalities by taking a *design-oriented* approach. For instance, consider design of a network processor in Figure 1 that is connected to a serial line and memory. The purpose of the processor is to receive and send data over the serial line using a specific communication protocol (such as CS/CD protocol for ethernet links). The decision to map functionalities into dedicated hardware or implement them as programs on a processor is usually based on estimations of achievable performance and implementation cost of the respective parts. This division is largely based on designer's experience and takes place early on in the design process while its implications are felt in every stage of the design, often leading to portions of design that are either under- or over-designed with respect to their

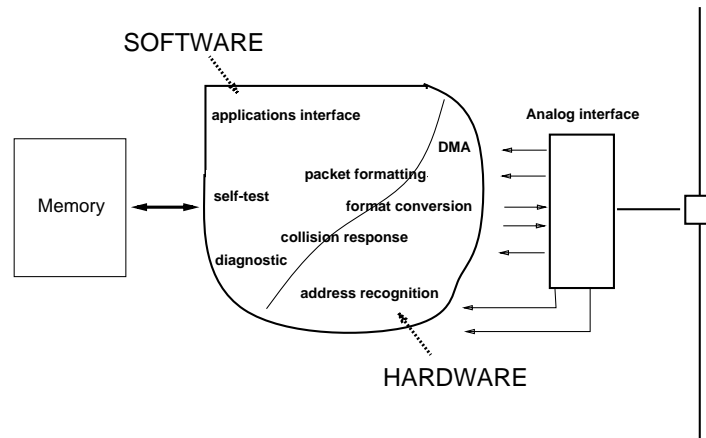


Figure 1: A design-oriented approach to system implementation.

required performance. More importantly, due to the ad-hoc nature of the overall design process, there is no guarantee that a given implementation meets required system performance (except possibly by overdesigning).

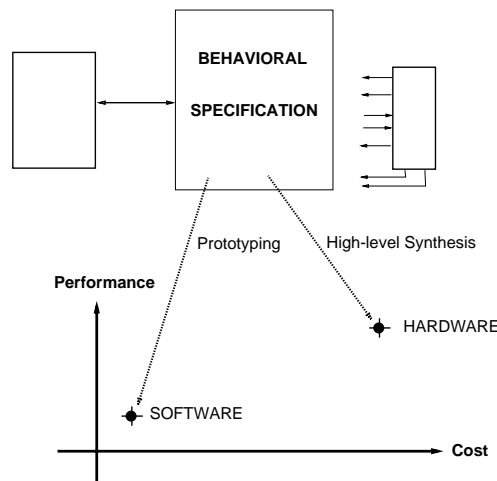


Figure 2: A synthesis-oriented approach to system implementation.

In contrast, a methodical approach to system implementation can be formulated as *synthesis-oriented* solution which has been enormously successful in design of individual integrated circuit chips (*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 choice of finding a suitable specification language for digital systems is a subject of on-going research, use of procedural *hardware description languages* (HDLs) to describe integrated circuits has been gaining wide acceptance in recent years.

A synthesis-oriented approach to digital circuit design takes a behavioral description of circuit func-

tionality and attempts to generate a gate-level implementation that can be characterized as a purely hardware implementation (Figure 2). Recent strides in high-level synthesis have made it possible to synthesize digital circuits from high-level specifications and several such systems are available from industry and academia [1, 2, 3, 4, 5, 6, 7]. The outcome of synthesis is 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 use of semi-custom or custom design technologies with associated increases in cost and design turn-around time. Therefore, for large system designs, synthesized hardware solutions tend to be fairly expensive depending upon the choice of technology required for chip implementation.

On the other end of the system development cost and performance spectrum, one can also create a simulatable software prototype of a system using a general-purpose programming language, for example, Rapide prototyping system [8]. Such software prototypes are rather quick to build and are often used for verifying system functionality. However, performance of software prototypes very often falls short of what is desired for time-constrained system designs (Figure 2).

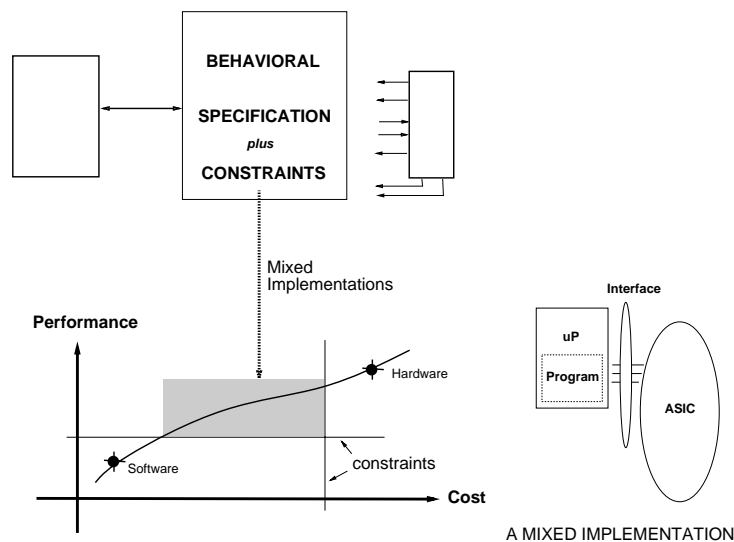


Figure 3: *Proposed approach to system implementation.*

However, we do know from our practical experience 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 that contain both hardware and software components. Such an approach would benefit from a systematic analysis of design trade-offs that is common in synthesis while at the same time creating systems that are cost-effective.

One way to accomplish this task would be to specify constraints on cost and performance of the resulting implementation (Figure 3). In this article, we present an approach to systematic exploration of system designs that is driven by the constraints. This work is built upon high-level synthesis techniques for digital hardware [6] by extending the concept of a resource needed for implementation. Figure 4

shows the essential aspects of this approach. A behavioral specification is captured into a system model that is partitioned for tentative implementation into hardware and software. The partitioned model is then synthesized into interacting hardware and software components for the target architecture shown in Figure 5. The target architecture uses one processor which is embedded with an application-specific hardware. The processor uses only one level of memory and address-space for its instructions and data. At this time, the application-specific hardware is not pipelined, for the sake of simplifying the synthesis and performance estimation task for the hardware component. Even with its relative simplicity, the target architecture is applicable to a wide-class of applications in embedded systems.

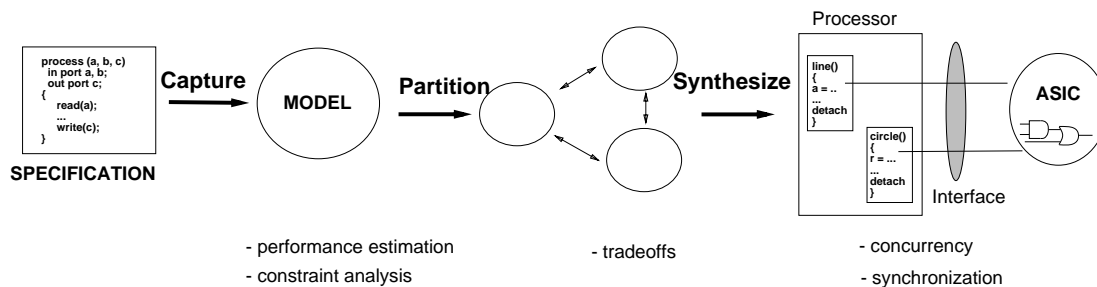


Figure 4: *Synthesis approach to embedded systems.*

Among the related work, [9] presents implementation of hardware or software from a co-specification; [10] describes synthesis of hardware or software for interface circuits; [11] describes a methodology for generation for hardware and software based on a unified FSM based model; given a system specification as a C-program [12] identifies portions of the program that can be implemented into hardware in order to achieve a speedup of overall execution times. [13, 14] present frameworks for generation of hardware and software components of a system. Several new architectures have been proposed that use field-programmable gate arrays to create special purpose co-processors to speed-up applications (PAM [15], MoM [16]) or to create prototypes (QuickTurn [17]).

This article is organized as follows. In Section 2 we present how we capture system functionality and constraints into an intermediate representation where partitioning trade-offs can be explored systematically. Section 3 we introduce a technique for partitioning system functionality. Section 4 presents synthesis techniques used for realizing mixed system designs. In Sections 5 and 6 we present an example design and conclusions from our experiments in system synthesis.

2 Capturing specification of system functionality and constraints

We describe system functionality using a hardware description language, *HardwareC* [18]. The co-synthesis approach formulated here does not depend upon the particular choice of the HDL and could use other HDLs such as VHDL or Verilog. However, use of *HardwareC* leverages the use of *Olympus*

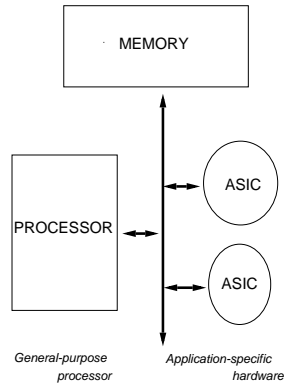


Figure 5: Target architecture.

tools developed for chip-level synthesis.

HardwareC follows much of the syntax and semantics of the programming language, C with modifications necessary for correct and unambiguous hardware modeling. A *HardwareC* description consists of a set of interacting *processes* which 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.

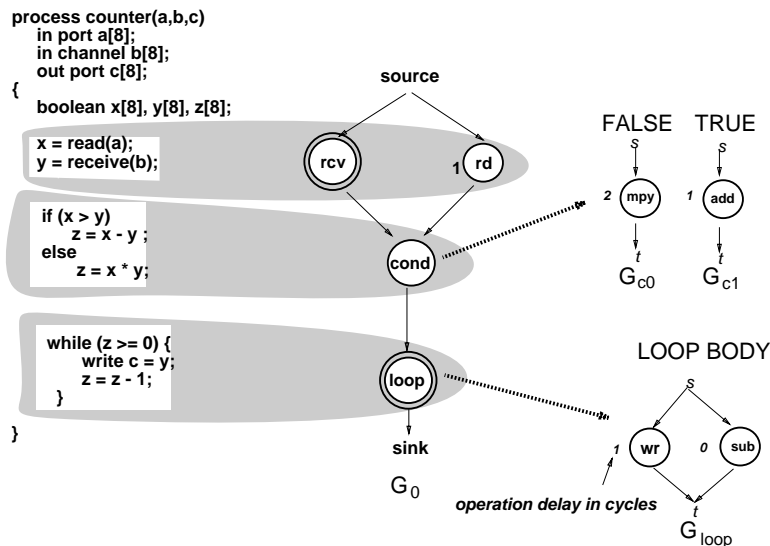


Figure 6: Example of input specification and capture.

Figure 6 shows an example of *HardwareC* specification. This example performs two data input operations, followed by a conditional in which a counter index is generated. The counter index, z , is used to seed a down-counter indicated by the `while` loop. This HDL specification is captured into a

graph-based representation as shown.

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

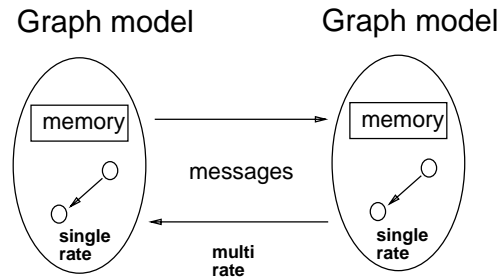


Figure 7: *Properties of the graph model.*

Model properties

The sequencing graph is a *polar* graph with source and sink vertices which represent *no-operations*. A set of variables is associated with each graph model which 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 and is used for facilitating communication between operations. Because of 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 a *multi-rate* execution semantics, that is, there may be a variable number of executions of an operation with respect to an operation in another graph model. Due to this multi-rate nature of execution, communications across graph models are implemented using message-passing primitives, like `send` and `receive`. Use of these primitives simplifies specification of inter-model communications. A multi-rate specification is an important feature for modeling heterogeneous systems, because the processor and application-specific hardware may run on different clocks and speeds.

HardwareC allows specification of operations to represent synchronization to external events, e.g., receive operation, as well data-dependent loop operations. These operations, referred to as non-deterministic delay () operations, present unknown execution delays. The ability to model operations is of key importance for reactive embedded system descriptions. In Figure 6, operations are indicated by

double circles.

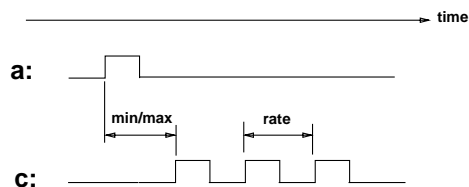


Figure 8: *Timing constraints.*

Timing constraints in specification

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

min/max delay constraints: These constraints provide bounds on the time interval between initiation of execution of two operations.

execution rate constraints: These constraints 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 [19]. Minimum delay constraints are captured in the graphical representation by providing weights on the edges indicating delay of the corresponding source operation. Additional *backward* edges are needed to capture maximum delay constraints (Figure 9).

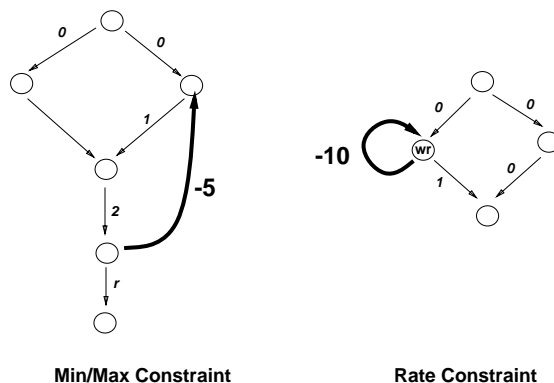


Figure 9: *Representation of timing constraints.*

Model analysis

Having captured system functionality and constraints in a graphical model, we are now able to make estimations on the system performance and verify the consistency of specified constraints. Performance measures require estimation of operation delays. These delays are computed separately for hardware and software implementation based on the type of hardware to be used and the processor used to run the software. Processor characteristics are captured using a *processor cost model* that consists of execution delay function for a *basic* set of processor operations, memory address calculation function and memory access time and processor interrupt response time.

Timing constraint analysis attempts to find answer to the following question: “Are imposed constraints satisfiable for a given implementation?” An implementation of a model is indicated by assigning appropriate delays to the operations with known delays (not) in the graph model. Constraint satisfiability is related to the structure as well as the actual delay and constraint values on the graph. Some structural properties of the graphs (relating to 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. Such constraints can not be satisfied by *any* assignment of non-negative operation delay values. In presence of operations in a graph model, a timing constraint is considered **satisfiable** if it is satisfied for all possible (and possibly infinite) delay values of the operations. A timing constraint is considered **marginally satisfiable** if it is satisfiable for all possible values within specified bounds on the delay of the operations. Marginal satisfiability analysis is useful in allowing use of timing constraints that are satisfiable under some implementation assumptions (i.e., acceptable bounds on operation delays) and without these assumptions these constraints would otherwise be considered *ill-posed* by the general timing constraint satisfiability analysis [20].

Timing constraint analysis is performed by graph analysis on the weighted sequencing graphs. For the sake of explanation, let us consider first the case where the graph model does not contain any operations. Therefore, every edge in the graph can be labeled with a finite and known weight. In such a graph, a min/max delay constraint is unsatisfiable if there exists a positive cycle in the graph model [20]. Next, in presence of operations, timing constraints are satisfiable if there exist no cycles containing operations. For a cycle containing an operation, it is not possible to determine satisfiability of timing constraints and only marginal satisfiability can be guaranteed. It is possible to break the cycle by graph transformations that preserve the program semantics. We will illustrate this concept shortly by an example.

For non-pipelined implementations, rate constraints can be treated as min/max delay constraints between corresponding source and sink operations of the graph model. Thus the above min/max constraint satisfiability criterion can be applied to analysis of rate constraints. It should be noted that, in some cases system throughput (specified by rate constraints) can be optimized independently of the overall per execution delay, i.e., system latency by making use of a pipelined execution model and using extra

resources. Indeed for deterministic and fixed-rate systems particularly used for DSP applications, extensive transformations have been developed that determine and achieve bounds on system throughput [21]. However, as explained earlier, systems modeled by the sequencing graphs, in general, operate at different rates. In addition, due to the presence of operations due to loops, the rate at which a particular operation executes may change over time. While this property is essential to modeling control-dominated embedded systems, it makes the problem of determination of absolute bounds on achievable system throughput considerably harder. We illustrate the issue of rate constraints on graphs containing operations by the following example.

Example 2.1. Consider the following *HardwareC* process fragment.

```

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

```

v is a boolean array that represents an integer. In presence of a rate constraint, on the read operation the constraint graph has a cycle containing an 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 10).

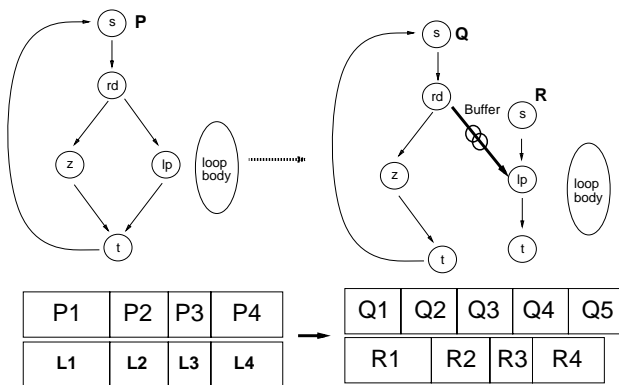


Figure 10: *Breaking cycle by graph transformation.*

The interval between successive executions of the read operation is determined by the overall execution time of the while loop. Due to this variable-delay loop operation, the input rate at port p is variable and can not always be guaranteed to meet the required rate constraint. In general, the determination of achievable throughput at port p is a hard problem. Marginal satisfiability of the rate constraint can be ensured by graph transformation and using finite size buffers as explained next.

Figure 10 shows the sequencing graph model, corresponding to process `test`. Identifier `rd` refers to read operation, `lp` refers to the while loop operation. Symbols `P1`, `P2` etc. in the execution trace

below indicate first, second, etc. invocations of the process `test`. L1, L2 etc. indicate multiple invocations of the `lp` operation. Depending of the side effects produced by the `loop-body` it is possible to transform the original graph into fragments and such that executions of and can be overlapped in order to improve the throughput of the `read` operation in . The data transfer from to is accomplished by means of a buffer. We elaborate on this example for the case when is implemented completely in software in Example 4.2.

In general, consider a process that contains an operation due to an unbounded loop. The operation induces a bipartition of the calling process, , such that the set of operations in (for example, `read` operation in process `test`) must be performed before invoking the loop body and the set of operations in can only be performed after completing execution of the loop body. Functional pipelining of , , and the loop can then be used to improve reaction rate of . Since we assume nonpipelined hardware, these transformations are used only in the context of the software component as described in Section 4.

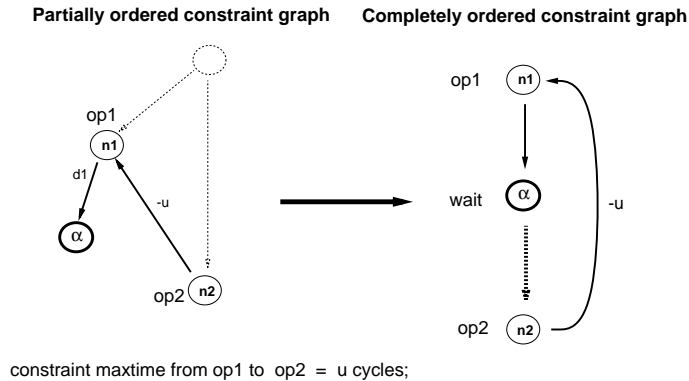


Figure 11: *Linearization in software leads to creation of unsatisfiable timing constraints.*

Constraint analysis and software

Constraint analysis for a software implementation of a graph model is complicated by the linear execution semantics imposed by the software running on a single-processor target architecture. That is, a complete order of operations in the graph model is needed in order to perform delay analysis for software operations. In creating a complete order of operations, it is likely that unbounded cycles may be created, which would make constraints unsatisfiable. As shown in Figure 11, any serialization that puts an operation between two operations `op1` and `op2` will make any maximum delay constraint between `op1` 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 operations such as `wait` for a synchronization or communication with some (unrelated) computation. But such an overlap requires the ability to schedule operations dynamically in software since the

simultaneously active operations may complete in different orders. 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 12). A thread is defined as a linearized set of operations that may or may not begin by an operation indicated by a circle in Figure 12. Other than the beginning operation, a thread does not contain any operations. The delay of the initial operation is considered 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 which may create unbounded cycles as explained by Figure 11 earlier.

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

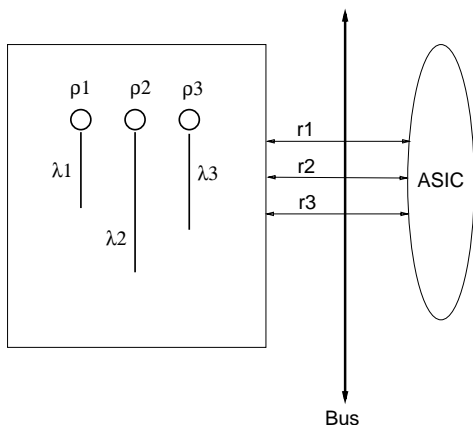


Figure 12: *Software model to avoid creation of cycles.*

3 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 circuit 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 is determined by the effect of hardware-software partition on utilization of the processor and the bandwidth of the bus between the processor and application-specific hardware. Thus a partitioning scheme must attempt to capture and make use of its 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 function* that captures these properties and use it to direct the partitioning algorithm towards a desired solution where an optimum solution is defined by the minimum value of the partition cost function.

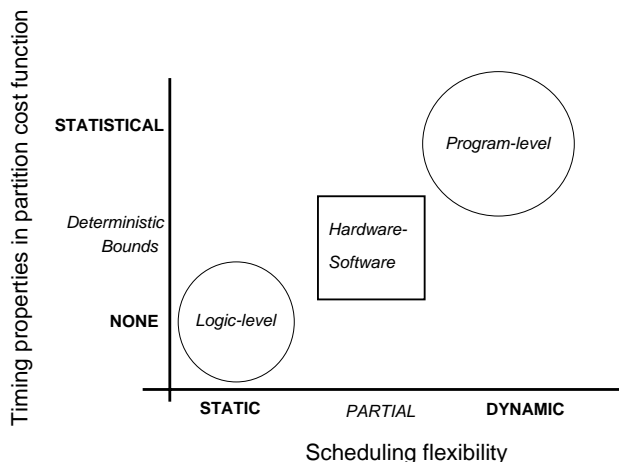


Figure 13: *Use of timing properties in partition cost function*

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 been focused on optimizing area and pinout of resulting circuits. It is hard to capture the effect of a partition on timing performance during partitioning stage. Part of the problem lies in the fact that timing properties are usually *global* in nature, thus making it difficult to make incremental computations of the partition cost function which is essential in order to develop effective partition algorithms. Approximation techniques have been suggested to take into account effect of a partition on overall latency [22].

Note, however, that partitioning in software world does make extensive use of statistics timing properties in order to drive the partitioning algorithm [23]. The distinction between these two extremes of hardware and software partitioning is drawn by the flexibility to schedule operations. Hardware partitioning attempts to divide circuits which implement scheduled operations. On the other hand, the program-level partitioning addresses operations that are scheduled at run-time.

In our approach to partitioning for hardware and software we take an intermediate approach (Figure 13) where we use deterministic bounds to compute timing properties that are incrementally computable in the partition cost function, that is, the new partition cost function can be computed in constant time. This is accomplished by using a software model in terms of a set of program threads as shown in Figure 12 and a partition cost function, C , that is a linear combination of its variables. The software component is characterized by following properties:

1. **Thread latency**, t_{lat} (seconds) indicates the execution delay of a program thread.
2. **Thread reaction rate**, r_{thr} (per second) is the invocation rate of the program thread.
3. **Processor utilization**, U_{proc} indicates utilization of the processor. It is calculated by

$$U_{proc} = \frac{1}{T} \sum_{i=1}^n t_{lat_i}$$

4. **Bus utilization**, U_{bus} (per second) is the total amount of communication taking place between the hardware and software. For a set of n variables to be transferred between hardware and software,

$$U_{bus} = \frac{1}{T} \sum_{i=1}^n t_{bus_i}$$

where t_{bus_i} is the inverse of the minimum time interval (in seconds) between two consecutive samples for variable x_i which is marked for destination to one of the program threads.

Characterization of software using t_{lat} and r_{thr} parameters makes it possible to calculate static bounds on software performance. Use of these bounds is helpful in selecting 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 since typically there is a distribution of thread invocations and communications based on actual data values being transferred.

The hardware size S_{hw} is computed bottom-up from the size estimates of the resources implementing the individual operations. In addition, the interface between hardware and software is characterized 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 above.

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

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

1. *Timing constraints are satisfied for the two sets of graph models*
2. *Processor utilization, $U_{proc} \leq 1$,*
3. *Bus utilization, $U_{bus} \leq 1$ and*
4. *A partition cost function, $C_{part} = C_{hw} + C_{sw} + C_{int}$ is minimized.*

An exact solution to the constrained partitioning problem, that is, a solution that minimizes the partition cost function requires examination of a large number of solutions which is typically exponential in the number of operations under partition. As a result, heuristics to find a ‘good’ solution are often used with the objective of finding an optimal value of the cost function that is minimal with respect to some local properties. Most common heuristics to solving partitioning problem start with a constructive initial solution which is then improved by some iterative procedure. Iterative improvement can be achieved, for example, by 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. Details have been presented elsewhere [24]. The procedure proceeds by identifying operations that can be implemented in software such that the corresponding constraint graph implementation is satisfiable 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 operations related to data-dependent loop operations define 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, r_{req} , of the corresponding program thread, θ . The maximum achievable reaction rate, r_{max} 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 partition affects its execution delay. It also affects the latency reaction rate of the thread to which the operation is moved. Its effect on processor and bus bandwidth utilization is similarly computed. At any step, operations are selected for migration so that the move lowers the communication cost and timing constraints satisfiability is maintained. In addition, we check for communication feasibility by verifying that $r_{req} \leq r_{max}$ for each thread, and processor and bus utilization constraints are satisfied.

4 System synthesis

From partitioned graph models, the next problem is to synthesize individual hardware and software components. Generation of hardware circuits from sequencing graph models has been addressed in detail elsewhere ([18] and other approaches in [1, 2, 3, 5, 7]). So 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 that the issues related to memory management are not relevant to this problem. The partitioning in previous section resulted in identification of graph models that are to be implemented in hardware and operations (organized as program threads) that are to be implemented in software.

Example 4.1. The process `test` shown in Example 2.1 can be implemented 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 the other thread T2 consists of operations in the body of the loop. For each execution of thread T1 there are v execution of thread T2.

The program generation from 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 more appropriate. A dependency between two operations can be either a data dependency or a control dependency. Depending upon predecessor relationships and timing of the operations, some of these dependencies can be made redundant by insertion of some other dependencies such that resulting program threads are *convex*, that is, all external dependencies are limited to the first and last operations. For a given subgraph corresponding to a program thread, an incoming data dependency can be moved up to its first operation and an outgoing data dependency can be moved down to its last operation. This procedure results in a potential loss of concurrency, however, it makes the task of routine implementation easier since all the routines can be implemented as independent programs with statically embedded control dependencies.

Rate constraints and software

As mentioned earlier, in presence of dependencies on operations, it is not always possible to guarantee that a given software implementation will meet the data rate constraints on its I/O ports. In case of synchronization related operations, it is possible to check for marginal satisfiability of timing constraints by assigning context-switch delay to the respective wait operations. However, in case of unbounded loop related operations, the delay due to these operations consists of active computation time, therefore, marginal timing satisfiability analysis requires estimation of loop index values. We illustrate this by an example below.

Example 4.2. Consider the threads T1 and T2 generated from process `test` mentioned in Example 2.1. The interval between successive executions of the `read` operation is determined by the overall execution time of the `while` loop. Due to this variable-delay loop operation, the input rate at port `p` is variable and the reaction rate of T1 can not always be guaranteed to meet it. Since the set of operations in `loop-body` may alter the contents of memory in process `test`, this process can be thought of consisting of two parallel processes as shown in Figure 14. The first operation of thread T2, `wait1`, is needed to observe the data-dependency of operations in the thread T2. The second wait operation, `wait2`, is needed to guarantee that any memory side-effects of T2 for variables in T1 are correctly reflected. In order 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

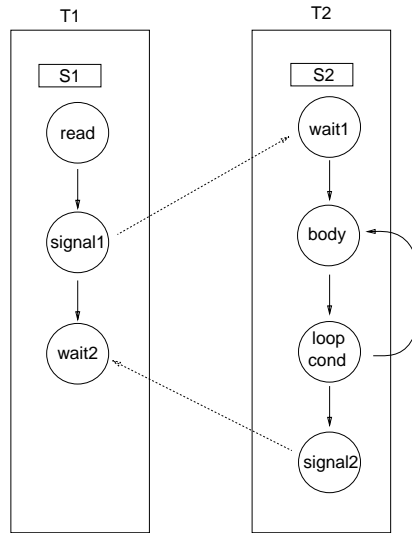


Figure 14: *Dependence of a program thread on a program thread corresponding to a loop*

program threads may also lead to violation of processor utilization constraint as described in previous sections.

However, it is possible to overlap execution of the loop thread T2 with execution of thread T1, and ensure marginal timing constraint satisfiability. Observe that operation wait2 can be removed if the looping thread does not produce any side effect on storage, S1, of the calling thread. That is, the variables common to 1 and 2 are only read and not modified by the loop body. In such cases the reaction rate of a program thread can be maintained by use of data-buffers between program threads. For implementation details, the reader is referred to [25].

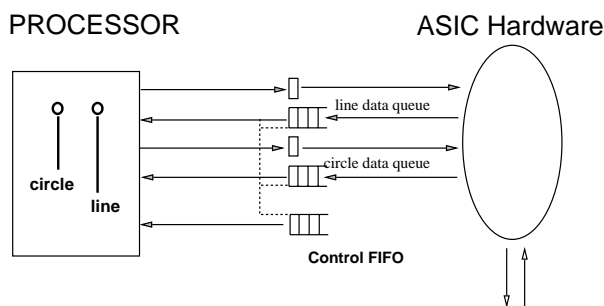
Hardware-software interface

Due to the serial execution of the software component a data transfer from hardware to software must be explicitly synchronized. Using a *polling* strategy, the software component can be designed to perform *pre-meditated transfers* from the hardware components based on its data requirements. This requires static scheduling of the hardware component. In cases where the software functionality is communication limited, that is, the processor is busy-waiting for an input-output operation most of the time, such a scheme would be sufficient. Further, in absence of any unbounded-delay operations, the software component in this scheme can be simplified to a single program thread and a single data channel since all data transfers are serialized. However, this would not support any branching, no reordering of data arrivals since dynamic scheduling of operations in hardware would not be supported.

In order to accommodate different rates of execution of the hardware and software components, and due to unbounded delay operations, we look for a *dynamic* scheduling of different threads of execution. Such a scheduling is done based on availability of data. One mechanism to perform such scheduling is by means of a **control FIFO** which attempts to enforce the policy that data items are consumed in the order

in which they are produced. 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 is equal to the number of threads of execution, since a thread execution is stalled pending availability of the requested data. The hardware-software interface protocol is described using guarded commands as shown in the Example 4.3 below.

Example 4.3. Consider mixed implementation of a graphics controller that contains two threads for generation of line and circle coordinates in software as shown in the figure below.



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 dequeue
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 are 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` comes 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`.

Note that thread scheduling by means of 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 it may be desirable 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 overheads.) The actual interconnect schematic between hardware and software is described for a single data queue in the Example 4.4 below.

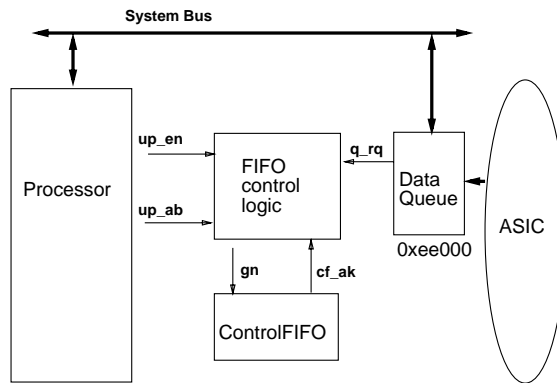


Figure 15: *Control FIFO schematic*

Example 4.4. The hardware-software interface schematic.

Figure 15 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 while the data queue 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 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 for generation of the enqueue is described by a simple state transition diagram shown in Figure 16. The control FIFO is ready to enqueue (indicated by $gn = 1$) a 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 indegree queues, the $enqueue_rq$ is generated by OR-ing the requests of all inputs to the queues. In case of multiple-outdegree queues, the signal $dequeue_rq$ is generated also by OR-ing all dequeue requests from the queue.

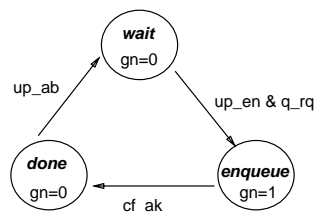


Figure 16: *FIFO control state transition diagram*

The control FIFO and associated control logic can be implemented either in hardware as a part of the ASIC component or in software. In case 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 `q_rq` lines from data queues are connected to processor unvectored interrupt lines, where the respective interrupt service routines are used to enqueue the thread identifier tags into the control FIFO. During the enqueue operations the interrupts are disabled in order to preserve integrity of the software control flow.

5 Example

As an experiment in achieving mixed system designs, we attempted synthesis of an ethernet-based network co-processor. The co-processor is modeled in *HardwareC* as a set of 13 concurrently executing processes which interact with each other by means of 24 send and 40 receive operations. The total description consists of 1036 lines of HDL code. A hardware-software implementation of the co-processor takes 8572 bytes of program and data storage for a DLX processor [26] and 8394 equivalent gates using LSI logic 10K library of gates. The mixed implementation is thus possible to be built using only one ASIC chip plus an off-the-shelf processor where as a complete hardware implementation would require use of a custom chip or two ASIC chips. More importantly, the mixed solution using a DLX processor running at 10 MHz is guaranteed to meet the imposed performance requirements of maximum propagation delay of 46.4 ns, maximum jam time of 4.8 ns, minimum interframe spacing of 67.2 ns and an input bit-rate of 10 Mb/sec.

6 Conclusions

Synthesis of embedded real-time systems from behavioral specifications constitutes a challenging problem in hardware-software co-synthesis. Due to relative simplicity of the target architecture compared to general-purpose computing systems, it also provides an opportunity in computer-aided design where such systems can be automatically synthesized from a unified specification. Further, the ability to perform constraint and performance analysis for such systems provides a major motivation for using synthesis approach over design-oriented implementation approaches. Even when manually designed, such systems can benefit greatly from prototypes created by a co-synthesis approach. A co-synthesis approach provides the ability to reduce the size of chip-synthesis task, while meeting the performance constraints, such that it makes it possible to use field or mask-programmable hardware to provide fast turn-around on complex system designs.

For hardware-software synthesis to be effective, specification languages that utilize capabilities of both hardware and software are needed. The approach presented in this article makes use a hardware-description language in order formulate the problem of co-synthesis as an extension of hardware synthesis. In the process many simplifications are made for the generated software, and room for considerable optimization of the software component exists.

7 Acknowledgment

The authors acknowledge discussions and contributions by Claudionor Coelho and David Ku. This research was sponsored by NSF-ARPA, under grant No. MIP 9115432 and by a fellowship provided by Philips at the Stanford Center for Integrated Systems.

References

- [1] R. K. Brayton, R. Camposano, G. D. Micheli, R. Otten, and J. van Eijndhoven, "The Yorktown Silicon Compiler System," in *Silicon Compilation* (D. Gajski, ed.), pp. 204–310, Addison Wesley, 1988.
- [2] R. Camposano and W. Rosenstiel, "Synthesizing Circuits from Behavioral Descriptions," *IEEE Transactions on CAD/ICAS*, vol. 8, no. 2, pp. 171–180, Feb. 1989.
- [3] J. Rabaey, H. D. Man, and *et. al.*, "Cathedral II: A Synthesis System for Multiprocessor DSP Systems," in *Silicon Compilation* (D. Gajski, ed.), pp. 311–360, Addison Wesley, 1988.
- [4] C. M. Chu, M. Potkonjak, M. Thaler, and J. Rabaey, "HYPER: an interactive synthesis environment for high performance real time applications," in *Proceedings of the International Conference on Computer Design*, (Cambridge, MA), Oct. 1989.
- [5] D. Thomas, E. Lagnese, R. Walker, J. Nestor, J. Rajan, and R. Blackburn, *Algorithmic and Register-Transfer Level: The System Architect's Workbench*. Kluwer Academic Publishers, 1990.
- [6] G. D. Micheli, D. C. Ku, F. Mailhot, and T. Truong, "The Olympus Synthesis System for Digital Design," *IEEE Design and Test Magazine*, pp. 37–53, Oct. 1990.
- [7] W. Wolf, A. Takach, C.-Y. Huang, R. Manno, and E. Wu, "The Princeton University Behavioral Synthesis System," in *Proceedings of the 29 Design Automation Conference*, June 1992.
- [8] D. C. Luckham, J. Vera, D. Bryan, and L. Augustin, "Partial Ordering of Event Sets and Their Application to Prototyping Concurrent Timed Systems," *Journal of Systems and Software*, July 1993.
- [9] N. Woo, W. Wolf, and A. Dunlop, "Compilation of a single specification into hardware and software," in *International Workshop on Hardware-Software Co-design*, Oct. 1992.
- [10] P. Chou, R. Ortega, and G. Borriello, "Synthesis of the Hardware/Software Interface in Microcontroller-Based Systems," in *Proceedings of the International Conference on Computer-Aided Design*, (Santa Clara), pp. 488–495, Nov. 1992.
- [11] M. Chiodo and A. Sangiovanni-Vincentelli, "Design Methods for Reactive Real-Time Systems CoDesign," in *International Workshop on Hardware-Software Co-design*, Oct. 1992.

- [12] J. Henkel and R. Ernst, "Ein softwareorientierter Ansatz zum Hardware-Software Co-Entwurf.," in *Proceedings Rechnergestuetzter Entwurf und Architektur mikroelektronischer Systeme.*, (Darmstadt, Germany), pp. 267–268, 1992.
- [13] M. B. Srivastava and R. W. Broderon, "Rapid-Prototyping of Hardware and Software in a Unified Framework," in *Proceedings of the International Conference on Computer-Aided Design*, (Santa Clara), pp. 152–155, 1991.
- [14] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *International Journal of Computer Simulations*, to appear.
- [15] P. Bertin, D. Roncin, and J. Vuillemin, "Introduction to Programmable Active Memories," in *Systolic Array Processors* (J. McCanny, J. McWhirter, and E. S. Jr., eds.), pp. 300–309, Prentice Hall, 1989.
- [16] 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. S. Jr., eds.), pp. 320–336, Prentice Hall, 1989.
- [17] S. Walters, "Reprogrammable hardware emulation automates system-level ASIC validation," in *WESCON/90 Conference Record*, (Anaheim, California), Nov. 1990.
- [18] D. Ku and G. D. Micheli, *High-level Synthesis of ASICs under Timing and and Synchronization Constraints*. Kluwer Academic Publishers, 1992.
- [19] B. Dasarathy, "Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Method of Validating Them," *IEEE Trans. Software Engineering*, vol. SE-11, no. 1, Jan. 1985.
- [20] D. Ku and G. D. Micheli, "Relative Scheduling Under Timing Constraints: Algorithms for High-Level Synthesis of Digital Circuits," *IEEE Transactions on CAD/ICAS*, vol. 11, no. 6, pp. 696–718, June 1992.
- [21] K. K. Parhi, "Algorithm Transform Techniques for Concurrent Processors," in *Proceedings of the IEEE*, pp. 1879–1895, Dec. 1989.
- [22] R. Gupta and G. D. Micheli, "Partitioning of Functional Models of Synchronous Digital Systems," in *Proceedings of the International Conference on Computer-Aided Design*, (Santa Clara), pp. 216–219, Nov. 1990.
- [23] V. Sarkar, *Partitioning and scheduling parallel programs for multiprocessors*. MIT Press, Cambridge, Mass., 1989.
- [24] R. K. Gupta and G. D. Micheli, "System-level Synthesis Using Re-programmable Components," in *Proceedings of the European Design Automation Conference*, pp. 2–7, Mar. 1992.
- [25] R. K. Gupta, C. C. Jr., and G. D. Micheli, "Program Implementation Schemes for Hardware-Software Systems," in *International Workshop on Hardware-Software Co-design*, Oct. 1992.
- [26] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, ch. 3. Morgan-Kaufmann, 1990.