Design Automation for Embedded Systems, 1, 69–120 (1996)
©1996 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

A Co-Synthesis Approach to Embedded System Design Automation

RAJESH K. GUPTA rgupta@cs.uiuc.edu

Department of Computer Science, University of Illinois, Urbana-Champaign
1304 W. Springfield Avenue, Urbana, IL 61801.

GIOVANNI DE MICHELI nanni@galileo.stanford.edu

Department of Electrical Engineering, Stanford University
Stanford, CA 94305.

Editor:

Abstract. Embedded systems are targeted for specific applications under constraints on relative timing of their actions. For such systems, the use of *pre-designed reprogrammable* components such as microprocessors provides an effective way to reduce system cost by implementing part of the functionality as a program running on the processor. However, dedicated hardware is often necessary to achieve the requisite timing performance. Analysis of timing constraints is, therefore, key to determination of an efficient hardware-software implementation. In this paper, we present a methodology for embedded system design as a *co-synthesis* of interacting hardware and software components. We present a decomposition of the co-synthesis problem into sub-problems, that is useful in building a framework for embedded system CAD. In particular, we present operation-level timing constraints and develop the notion of satisfiability of constraints by a given implementation both in the deterministic and probabilistic sense. Constraint satisfiability analysis is then used to define hardware and software portions of functionality. We describe algorithms and techniques used in developing a practical co-synthesis framework, VULCAN. Examples are presented to show the utility of our approach.

Keywords: Embedded system co-synthesis, Constraint analysis, Multithreading for embedded systems

1. Introduction

Application-specific systems are designed for dedicated applications. Examples of such systems can be found in medical instrumentation, process control, automated vehicle control, and networking and communication systems. As these systems are contained within a larger (sometimes) non-electronic environment, these are commonly referred to as *embedded systems*. Embedded computer systems have been applied to tasks erstwhile handled by electronic or electro-mechanical *non-computing* systems. As a result, the volume of the embedded electronics market has grown. For the year 1991, the industrial and medical electronics market alone accounted for $31 billion compared to the general purpose computing systems market of $46.5 billion [1]. The continuing growth in embedded systems has been fueled by the advent of microprocessor-based microcontrollers, the primary compute element in a system. For the year 1991, the market for microcontrollers amounted to $4.6 billion and has been rising at an 18% annual growth rate which is about twice the rate of growth for general-purpose systems [1].

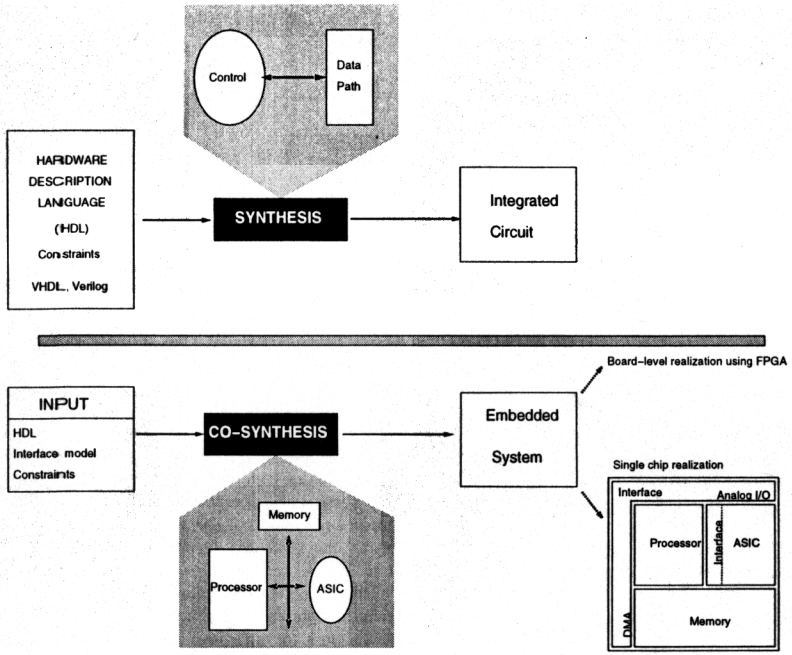


Figure 1. CAD for integrated circuits and embedded systems.

While there has been notable growth in the use and application of embedded systems, improvements in the design process for such systems have not kept pace, leading to a gap in the evolution of processor technology and its application in embedded computing systems. Currently, approximately 80% of the microcontrollers used in embedded systems are 4- and 8-bit processors of old generations [2]. Of the total $4.6 billion microcontroller market for 1991, 32-bit processors account for less than 4% or $184 million, even though such processors have been commonplace since 1985 and have been targets of almost all advances in processor technology.

There are several reasons for this discrepancy in the advancement of embedded versus general-purpose systems. Due to a specific and limited application, the efficiency of implementation and manufacturing costs are important considerations in the final choice of system components. Current practice in embedded systems is primarily a manual design process, characterized by a long design time and high cost of design. In recent years, a growing need has been felt for the use of computer-aided design (CAD) tools to help rapidly explore the design alternatives and achieve a cost-optimal implementation that meets constraints on performance [3], [4], [5], [6], [7], [8].

This work develops design automation of an embedded system by extending techniques for synthesis of digital hardware and software compilation to formulate a *co-synthesis*

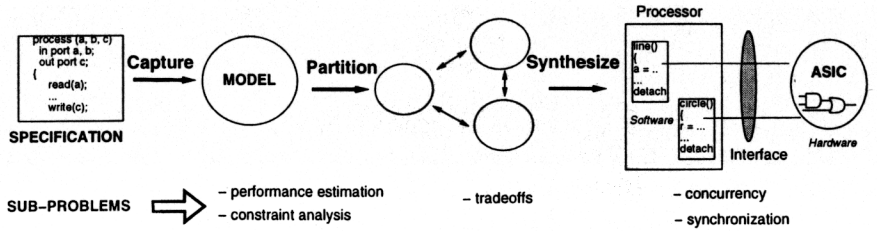


Figure 2. Synthesis approach to embedded systems.

approach for system hardware and software. Figure 1 shows the inputs and outputs of a co-synthesis system and its parallel in existing chip-level synthesis solutions shown by the top portion of the figure. A co-synthesis approach is specific to the input description of system functionality and the target architecture for system implementation. The search for a suitable language for system specification is an active area of research [9], [10], [11]. Our choice of a programming language is detailed in next section. The choice of a target architecture is influenced by the intended application. We choose a target architecture that consists of a processor assisted by application-specific hardware components. 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.

To keep the focus on important synthesis tasks, we make the following assumptions on the target architecture. Many of these assumptions can be dropped in a larger system co-design framework without affecting the underlying co-synthesis approach presented here. We restrict ourselves to use of a single re-programmable component. The multiprocessing issues such as shared access, though important, are orthogonal to the system co-synthesis problem addressed in this work. We also consider only the case where all memory accesses are to a single-level memory. The hardware modules are connected to the system address and data busses. Thus, all the communication between the processor and different hardware modules takes place over a shared medium. Finally, we assume that the re-programmable component is always the bus master. Almost all re-programmable components come with facilities for bus control. The inclusion of such functionality on the application-specific component would greatly increase the total hardware cost and, therefore, it is avoided. A single-chip realization of this target architecture would use a processor core and ASIC hardware circuits using standard cells or gate array technologies.

With the above-mentioned definition of inputs and target architecture, the co-synthesis problem is decomposed into sub-problems as shown in Figure 2. The behavioral input specification is captured into a system model that is partitioned for implementation into hardware and software. The partitioned model is then synthesized into interacting hardware and software components.

This paper is organized as follows. In Section 2 we present the input description and its compilation into an abstract model. We discuss constraint modeling and analysis techniques that are useful for embedded systems in Sections 3 and 4. We present our choice of the software and runtime system in Section 5. Section 6 describes the partitioning procedure for hardware and software based on the notion of non-determinism in the input system model. Section 7 describes the implementation experience for the Vulcan co-synthesis system. We summarize main contributions and directions for future research in Section 8.

2. Embedded System Modeling and Representation

We seek a model of system functionality in which the following properties of target applications must be modeled and represented:

- The system consists of parts that operate at different speeds of execution,
- The interaction between parts of a system requires synchronization operations,
- There are constraints on the relative timing of operations.

For system analysis and hardware synthesis purposes we need a language model that allows for explicit dependency analysis, deterministic data storage and timing attributes. Most current hardware description languages (HDLs) satisfy these requirements and are, therefore, our choice for input to system co-synthesis. In particular, we use *HardwareC* [12] for system modeling. The particular choice of *HardwareC* is immaterial and other HDLs, such as VHDL, may be used as well to generate the flow graph model upon which this work is based.

The basic entity for specifying system behavior is a *process*. A process executes concurrently with other processes mentioned in the system specification. A process restarts itself on completion of the last operation in the process body. Thus, the use of multiple processes to describe a system functionality abstracts the parts of a system implementation that operate at different speeds of execution.

Example 2.1. Example of a simple HDL process

```
process example (a, b, c)
  in port a[8] ;
  in channel b[8] ;
  out port c[8] ;
{
  boolean x[8], y[8], z[8] ;

  x = read(a);
  y = receive(b);
  if (x > y)
        z = x - y ;
  else
        z = x * y ;
  while (z >= 0) {
        write c = y ;
        z = z - 1 ; }}
```


This process performs two data input operations followed by a conditional operation and a loop operation, then it restarts. □

A communication between two operations is done either by a direct connection or over a shared medium such as memory. A choice of a particular communication mechanism depends on the individual operations and their implementation into hardware or software. Communication between operations belonging hardware and software occurs over *ports*. Ports represent communication to a shared-memory (SM), or inter-process communication using message-passing (MP) operations. In message-passing, the data transfer between operations is preceded by a *handshake protocol* that results in a simultaneous execution of the sending and receiving operations. The process of bringing operation executions together is referred to as a *synchronization*[1].

All communication between operations within a process body is based on shared memory. This shared storage is declared as a part of the process body (for example variables `x`, `y` and `z` in Example 2.1 above). Inter-process communication is specified by message-passing operations that use a *blocking* protocol for synchronization purposes. Example 2.1 above shows a process description containing a message-passing receive operation. The `read` refers to a synchronous port read operation that is executed unconditionally as a value assignment operation from the wire or register associated with the port `a`. `receive` is a message-passing based read operation where the channel `b` carries additional control signals to implement a blocking read operation based on the availability of data on channel `b`.

2.1. System Representation

Abstractions of operation-level concurrency and synchronization are important since these affect the amount of resources required for hardware implementations. Software representation requires an abstraction of its interaction with a non-trivial runtime environment. We use a polar graph model that represents operation-level concurrency explicitly, and the operations due to the runtime system by a suitable choice of the source and sink operations. System interaction with an environment is treated as a generalization of the interaction between its components. This generalization is supported by a port abstraction which, in implementation, can be a memory location, another system, or a device.

DEFINITION 1 *A* **flow graph model** *is a polar acyclic graph* $G = (V, E, \chi)$ *where* $V = \{v_0, v_1, \ldots, v_N\}$ *represent operations with* v_0 *and* v_N *being the source and sink operations respectively. The edge set,* $E = \{(v_i, v_j)\}$ *represents dependencies between operation vertices. Function* χ *associates a Boolean (enabling) expression with every edge. In the case of edges incident from a condition vertex or incident to a join vertex, the enabling expression refers to the condition under which the successor node for the edge is enabled.*

Operation vertices used in a flow graph model consist of arithmetic, logical, input/output, conditional, wait and link operations. A *wait* operation is used to represent

the synchronization events at model ports. A (call or loop) *link* operation is used to represent the hierarchy of models. The called flow graph may be invoked one or many times depending upon the type of the link vertex. Function and procedure calls are represented by a call link vertex where the body of function/procedure is captured in a separate graph model. A loop link operation consists of a loop condition operation that performs testing of the loop exit condition and a *loop body*. The loop body is represented as a separate graph model. All loop operations are assumed to be of the form "repeat-until", that is, the loop body is executed at least once. HDL specification of "while"-loops is transformed into equivalent "repeat"-loop operations using conditional operations.

Notationally, an edge (v_i, v_j) from v_i to v_j is indicated by the relation $v_i < v_j$ on operation vertices. A successor to a conditional operation is considered *enabled for execution* if the result of condition evaluation selects the branch to which the operation belongs. This is expressed by the enabling condition associated with the edge from the condition vertex. In general, a multiple in-degree operation vertex is enabled by evaluating an *input expression* consisting of logical AND and OR operations over enabling expressions of its input edges. Similarly, on completion of an operation, all or a set of its successor vertices can be enabled. For each vertex, its *output expression* is an expression over enabling conditions of its output edges. These expressions determine the flow of control through the graph model.

A flow graph is considered *well-formed* if the input and output expressions use either AND or OR operations but not both in the same expression. For a well-formed graph, a set of input or output edges to a vertex is considered *conjoined* if the corresponding expression is a conjunction over inputs or outputs. Similarly, a set of edges is *disjoined* if the corresponding expression is a disjunction. Structurally this makes the flow graph a *bilogic* graph [13]. For this reason, the flow graphs can be called *bilogic sequencing graphs* as opposed to (unilogic) *sequencing graphs* introduced in [12].

Example 2.2. Figure 3 shows example of a well-formed bilogic graph model for the process described in Example 2.1. Node 4 shows a one-bit condition variable, `c = (x > y)`. In general, it could be a multi-bit variable, thus leading to more than two branches. Note that for bilogic graphs, the `join` node is not essential since an appropriate input expression can be assigned to the successor node. However, a join node makes it easier in defining well-formed graphs. Conjoined and disjoined fanin and fanout of a vertex are also indicated by symbols '*' and '+' respectively. □

Finally, a system model consists of one or more flow graphs, that may be hierarchically linked to other flow graphs. That is, a system model is expressed as, $\Phi = \{G_1^*, G_2^*, \ldots, G_n^*\}$, where G_i^*, represents the process graph model G_i and all the flow graphs that are hierarchically linked to G_i.

2.2. *Implementation attributes*

An *implementation*, $\mathcal{I}(G)$, of a graph model, G, refers to assignment of delays and size properties to operations in G, and a choice of *runtime scheduler*, Υ, that enables execution of source operations in G. The assignment of values is related to the hardware

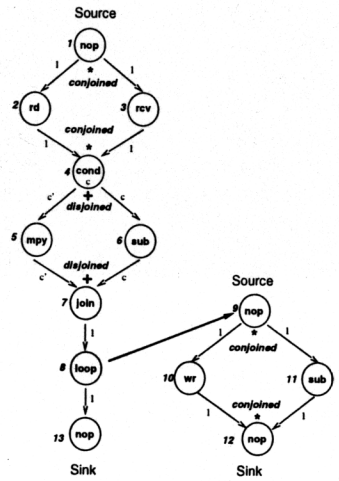


Figure 3. Flow graph of process `example`.

or software implementation of operations in G. For software, the runtime scheduler refers to the choice of a runtime system that provides the operating environment for execution of operations in G. This issue is discussed in Section 5.

Size attributes refer to the physical *size* of implementation of operations and graphs. A hardware implementation consists of hardware resources (also called data-path resources), control logic, registers, and communication structures such as busses and multiplexor circuits. The size of a hardware implementation is expressed in units of gates or cells (using a specific library of gates) required to implement the hardware. Each hardware implementation has an associated *area* that is determined by the outcome of the physical design. We estimate hardware size assuming a proportional relationship between size and area. The size attribute for software consists of program and data storage required.

A graph model captures the functionality of a system with respect to its behavior on its ports. The operational semantics of the graph model requires use of an *internal storage* to implement multiple assignments to variables and ports in HDL descriptions. Whereas additional variables can be created that avoid multiple assignments to the same variable, assignments to ports must still be preserved in a flow graph model. Further, a port is often implemented as a specific memory location (that is, as a shared variable) in software. The memory side-effects created by graph models are captured by a set $M(G)$ of variables that are referenced by operations in a graph model, G. $M(G)$ is independent of the cycle-time of the clock used to implement the corresponding synchronous circuitry and does not include storage specific to structural implementations of G (for example, control latches).

The timing properties of a flow graph model are derived using a *bottom-up* computation from individual operation delays. Let us first consider non-hierarchical flow graphs, that

is, graphs without link vertices. The *delay*, δ, of an operation refers to the execution delay of the operation. We assume that for a graph model, the delay of all operations are expressed as number of cycles for a given cycle time associated with the graph model. In a non-hierarchical flow graph, the delays of all operations (except `wait`) are fixed and independent of the input data. The `wait` operation offers variable delay which may or may not be data-dependent depending upon its implementation (see section 4.1). The *latency*, $\lambda(G)$, of a graph model, G, refers to the execution delay of G. The latency of a flow graph may be variable due to the presence of conditional paths.

Next, the hierarchical flow graphs also contain link vertices such as `call` and `loop` which point to the flow graphs in the hierarchy. Therefore, an execution delay can be associated with link vertices as the latency of the corresponding graph model times the loop index, or the number of times the called graph is invoked. Since the latency can be variable, therefore, the delay of a link vertex can be variable. It may also be *unbounded* in case of loop vertices, since these can, in principle, be invoked unbounded number of times.

We define a lower bound on latency as the *length*, $\ell(G) \in Z^+$, of the longest path between the source and sink vertices assuming the loop index to be one for the loop operations[2]. In presence of conditional paths, the length is a vector, $\underline{\ell} = (\ell[i])$ where each element $\ell[i]$ indicates the execution delay of a path in G. The elements of $\underline{\ell}$ are the lengths of the longest paths that are mutually-exclusive.

The length computation for a flow graph proceeds by a bottom-up computation of lengths from delays of individual operations. Given two operations, u and v with delays, δ_u, δ_v, these can be related in one of the following three ways in the flow graph: (a) Sequential composition, that is, $u < v$ or $v < u$. The combined delay of u and v is represented by $\delta_u \odot \delta_v$ and is defined as $\delta_u \odot \delta_v \stackrel{\Delta}{=} \delta_u + \delta_v$; (b) Conjoined composition, when the operations u and v belong to two branches of a conjoined fork. A conjoined composition is denoted by $\otimes$ and the delay is defined as $\delta_u \otimes \delta_v \stackrel{\Delta}{=} \max(\delta_u, \delta_v)$; (c) Disjoined composition, when the operations u and v belong to two branches of a disjoined fork. This composition is denoted by symbol $\oplus$ and the combined delay is defined as $\delta_u \oplus \delta_v \stackrel{\Delta}{=} (\delta_u, \delta_v)$. Clearly, a disjoined composition of two delays leads to a 2-tuple delay since the two operations belong to mutually exclusive paths. Binary operators, $\oplus$, $\otimes$ and $\odot$ are extended to k-ary operators over path lengths in a straightforward manner.

Rate of execution: We define the *rate of an execution* $\rho(k)$, at invocation k of an operation, as the inverse of the time interval between its k and $(k-1)^{th}$ execution. By this definition the rate of execution of an operation is computed only at times when the operation is executed. For a graph model, G, its *rate of reaction*, is the rate of execution of its source operation, that is, $\varrho_G(k) \stackrel{\Delta}{=} \rho_0(k)$. The reaction rate is a property of the graph model and it is used to capture the effect on the runtime system and the type of implementation chosen for the graph model. To be specific, the choice of a non-pipelined implementation of G leads to

$$\varrho_G(k)^{-1} = \lambda_k(G) + \gamma_k(G) \tag{1}$$

where $\gamma_k(G)$ refers to the *overhead delay*, that represents the delay in reinvocation of G. $\gamma_k(G)$ may be a fixed delay representing the overhead due to a runtime scheduler or it may be a variable quantity representing delay in case of conditional invocation of G.

2.3. *Non-determinism, Execution Rate and Communication*

From our discussion of delay estimation above it is clear that an operation presents fixed delay or variable delay during execution. This variance in delay is caused by the dependence of operation delay on either the *value* of input data or on the *timing* of input data. Example of operations with value-dependent delays are loops with data-dependent iteration counts. Since the execution delay (or latency) of a bilogic flow graph can, in general, be data-dependent due to the presence of conditional paths, the delay of a call vertex is also variable and data-dependent. Thus, in bilogic flow graphs the link vertices present value-dependent delays.

The second category concerns operations with delays that depend upon a response from the environment. An operation presents a timing-dependent delay only if it has *blocking* semantics. The only operation in the flow graph model with blocking semantics is the *wait* operation. The *read* and *write* operations are treated as non-blocking. Their blocking versions are created by adding additional control signals and the wait operation. (For this reason, the wait operation is also referred to as a *synchronization* operation.)

Data-dependent loop and synchronization operations introduce uncertainty over the precise delay and order of operations in the system model. Due to concurrently executing flow graph models, these loop and synchronization operations affect the order in which these and other operations are invoked. Due to this uncertainty, a system model containing loop and synchronization operations is called a *non-deterministic* [14] model and operations with variable delays are termed *non-deterministic delay* or $\mathcal{ND}$ *operations*.

For a given input/output operation, the system throughput at the corresponding port equals the rate of execution of the operation. For a flow graph containing no conditional and no $\mathcal{ND}$ operations, the rate of execution of all operations is the same and is independent of input data. Thus the execution of G proceeds at a *single rate*. For a single-rate graph model, the system throughput at all ports is identical. For two single-rate graph models, G_1 and G_2, there exists a fixed number of invocations of G_1 with respect to an invocation of G_2 given by the ratio $\frac{\varrho_1}{\varrho_2}$.

Example 2.3. Single rate graph models.

Figure 4 shows part of a vehicle cruise controller that consists of two single-rate graphs G_1 and G_2 implemented in hardware and software respectively. The latencies of the respective implementations are, $\lambda(G_1) = 25 \cdot \tau_h$ sec with $\gamma(G_1) = 0$ and $\lambda(G_2) = 665 \cdot \tau_s \cdot$ sec with $\gamma(G_2) = 85$ cycles. Assume that the clock cycle times for the hardware and software are 500 ns and 125 ns respectively. The reaction rates are $\varrho_{G_1} = 80,000$/sec and $\varrho_{G_2} = 10,666.6$/sec. Therefore, for each execution of the software model, there are a fixed number of $\frac{80000}{10666.6} = 7.5$ executions of hardware. □

The reaction rate of a graph model containing conditional and $\mathcal{ND}$ operations is variable. A graph model with variable reaction rate is termed a *multi-rate* execution model.

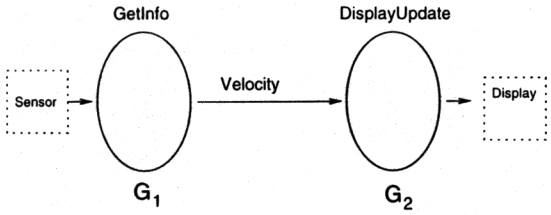


Figure 4. Single-rate models.

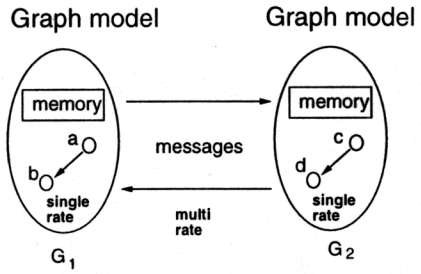


Figure 5. Graph model properties.

A multi-rate model has a bounded reaction rate if the model does not contain $\mathcal{ND}$ operations, else it is unbounded.

All communication in a single-rate graph model can be accomplished by means of shared storage since any execution of a graph model observes the partial order induced by its edges, *regardless of individual operation delays*. However, the relative ordering of operations *across* the graph model are dictated by the execution delay of individual operations. For software implementations this may lead to possible interleaving of operations in the graph models whereas a hardware implementation also includes the possibility of concurrent execution of operations across processes.

For any communication between operations across the graph models, a *safe* execution requires that the dependencies induced due to communication are always respected. For example, in Figure 5, a communication from operation c in G_2 to operation a in G_1 implies that only those executions are safe in which execution of c precedes execution of a. There are two ways to ensure that this ordering from c to a is always observed. One is to construct a single flow graph model by merging G_1 and G_2 in which an edge is added from c to a. This may not always be possible, particularly, if G_1 and G_2 have different reaction rates or use different clocks. An alternative is then to make the operation a block until c is available (and vice-versa). This is accomplished by using a message-passing protocol between G_1 and G_2.

The advantage of message-passing is realized when communicating across multi-rate model(s). In the case of single-rate models, use of message-passing provides a notational simplicity, when the use of shared-memory may be more efficient due to lower overheads. In principle, the communication between two single-rate models can be done using shared memory. This is, however, not convenient for different implementations, such as one in hardware and the other in software, of single-rate graph models, even if they have the same reaction rate. This is because, hardware and software may implement storage differently even though the access semantics in graph models are identical. On the other hand, communication across two multi-rate models using the same implementation can be accomplished by shared memory.

3. Timing Constraints

Operation-level timing constraints are of two types: (a) Operation *delay* constraints and (b) Execution *rate* constraints. Delay constraints are *unary* such as bounds on the delay of an operation, or *binary* as bounds on the delay between initiation time of two operations. By default, any sequencing dependency between two operations induces a minimum delay constraint. A *minimum timing constraint*, $l_{ij} \geq 0$ from operation vertex v_i to v_j requires that for all k:

$$t_k(v_j) \geq t_k(v_i) + l_{ij} \tag{2}$$

Similarly a *maximum timing constraint*, $u_{ij} \geq 0$ from v_i to v_j requires that for all k:

$$t_k(v_j) \leq t_k(v_i) + u_{ij} \tag{3}$$

We note that operation delay constraints are general and can be used to capture durational and deadline constraints in specifying real-time systems [15] and all possible binary timing relationships between operations [16] as shown in Figure 6. These relations can be directly specified in the HardwareC descriptions and are captured into a constraint graph model described later in this section.

Execution rate constraints refer to constraints on the interval of time between successive executions of an operation. In particular, execution rate constraints on input (output) operations refer to the rates at which the data is required to be consumed (produced). We assume that each execution of an input (output) operation consumes (produces) a *sample* of data. Execution rate constraints on input/output operations are called data rate constraints. A *minimum data rate constraint*, r_i (cycles^{-1}) defines the lower bound on the execution rate of operation v_i. Similarly, a *maximum data rate constraint*, R_i (cycles^{-1}), defines the upper bound on the execution rate of operation v_i.

$$\rho_{v_i}(k) \leq R_i \;\Rightarrow\; t_k(v_i) - t_{k-1}(v_i) \geq \tau \cdot R_i^{-1} \quad \text{for all} \quad k > 0 \tag{4}$$

$$\rho_{v_i}(k) \geq r_i \;\Rightarrow\; t_k(v_i) - t_{k-1}(v_i) \leq \tau \cdot r_i^{-1} \quad \text{for all} \quad k > 0 \tag{5}$$

In general, when considering the rate of execution of v_i we must consider the successive executions of v_i that may belong to separate invocations of G. A *relative execution rate*

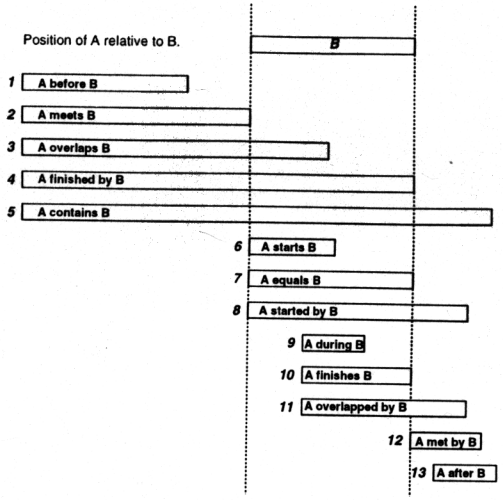


Figure 6. All possible binary timing relations.

constraint on an operation, v_i, with respect to a graph model, G, is a constraint on the rate of execution of v_i *when* G *is continuously enabled and executing*. In other words,

$$r_i^G \leq \rho_{v_i}(k) \leq R_i^G \tag{6}$$

for all $k > 0$ *and* there exists an execution, j, of G such that $t_j(v_0(G)) \leq t_{k-1}(v_i) \leq t_k(v_i) \leq t_j(v_N(G))$ where v_0 and v_N refer to source and sink vertices in G respectively. The relative rate of execution expresses rate constraints that are applicable to a specific *context* of execution as expressed by the control flow in G. Clearly, a relative rate constraint is meaningful when expressed relative to a flow graph in the hierarchy in which the operation resides.

We now explore the relationship of timing constraints to operation scheduling that is used in developing the notion of constraint satisfiability in the subsequent section.

3.1. *Relationship to Operation Scheduling*

For each invocation of a flow graph model, an operation is invoked zero, one or many times depending upon its position on the hierarchy of the flow graph model. The execution times $t_k(v)$ of an operation v are determined by two separate mechanisms: (a) The runtime scheduler, Υ, and (b) The operation scheduler, Ω. The runtime scheduler determines the invocation times of flow graphs, which may be as simple as fixed-ordered where the selection is made by a predefined order (most likely by the system control flow). This is typically the case in hardware implementations where the graph invocation

is purely a subject of system control flow. On the other hand, software implementations of the runtime scheduler are based on the choice of the runtime environment.

Given a graph model, $G = (V, E)$, the selection of a *schedule* refers to the choice of a function, Ω that determines the start time of the operations such that

$$t_k(v_i) \geq \max_{j \ni v_j < v_i} [t_k(v_j) + \delta(v_j)] \tag{7}$$

is satisfied for each invocation $k > 0$ of operations v_i and v_j. Here $\delta(\cdot)$ refers to the delay function and returns the execution delay of the operation.

Given a scheduling function, a timing constraint is considered *satisfied* if the operation initiation times determined by the scheduling function satisfy the corresponding Inequalities (2, 3, 4 and 5). Clearly, the satisfaction of timing constraints is related to the choice of the schedulers Ω and Υ. In general, choice of a particular operation scheduling mechanism depends upon the types of operations supported and the resulting control hardware or software required to implement the scheduler.

We consider first a model, G, where the delay of all operations in G is known and bounded. A schedule of G maps vertices to integer labels that define the start time of corresponding operations, that is, $\Omega_s : V \mapsto Z^+$ such that operation start times, $t_k(v_i) = \Omega_s(v_i)$ satisfy Inequality 7. A schedule is considered minimum if $|t_k(v_i) - t_k(v_o)|$ is minimum for all $v_i \in V$. Since this interval is fixed for all operations and for all k, therefore, such a schedule is referred to as a *static schedule*.

In general, due to the loop and wait operations, not all delays can be fixed or known statically, thus making a determination of an unique operation start time impossible for a static scheduler. We develop a bilogic relative schedule which is based on generalization of the relative schedule [17] for bilogic flow graphs. A bilogic relative scheduler uses runtime information to determine operation start times for each invocation of a graph model and, therefore, does not require $\delta(\cdot)$ to be a fixed quantity.

A bilogic relative schedule function maps vertices to a *set* of integers representing *offsets*. An offset $\underline{\theta}_{v_j}(v_i)$ of vertex v_i with respect to vertex v_j is the (finite) set of possible delays in starting execution of v_i after *completion* of operation v_j. A bilogic schedule, Ω_{br} computes the offset vectors such that

$$t(v_i) \geq \max_{a \in \mathcal{A}_b(v_i)} [t_a + \delta(a) + |\underline{\theta}_a(v_i)|_\infty] \tag{8}$$

where $|\cdot|_\infty$ refers to the largest element (or the infinity norm) of the vector. The bilogic anchor set of a vertex v_i is defined as the set of $\mathcal{ND}$ vertices that have a path to v_i, that is,

$$\mathcal{A}_b(v_i) = \{v_j \in V : v_j <^* v_i,\ v_j \text{ is } \mathcal{ND}\} \tag{9}$$

Note that the Inequality 8 can be derived from Inequality 7 for bilogic flow graphs by expressing the latter over the transitive closure, $G^{<^*}$ of G. Thus a solution to Inequality 8 will also satisfy Inequality 7 provided $|\underline{\theta}_a(v_i)|_\infty \geq \ell_M(a, v_i)$.

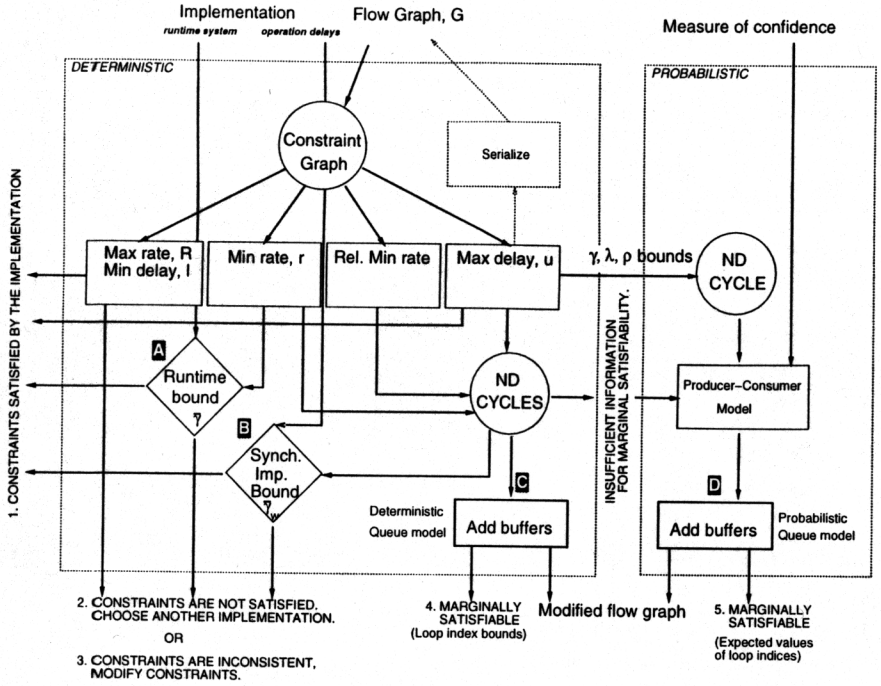


Figure 7. General flow of constraint satisfiability analysis.

4. Timing Constraint Satisfiability Analysis

For constraint analysis purposes, it is not necessary to determine a schedule of operations, but only to verify the *existence* of a schedule. Since there can be many possible schedules, constraint satisfiability analysis proceeds by identifying conditions under which no solutions are possible. A timing constraint is considered *inconsistent* if it can not be satisfied by *any* implementation of the flow graph model. Since the consistency of constraints is independent of the implementation, these are related to the structure of the graphs.

The timing constraints are abstracted in an edge-weighted *constraint graph model*, $G_T(V, E_f \cup E_b, \Delta)$, where the edge set contains forward edges E_f representing minimum delay constraints and backward edges E_b representing maximum delay constraints. An edge with weight $\delta_{ij} \in \Delta$ on edge $v_i < v_j$ defines constraint on the operation start times as $t_k(v_i) + \delta_{ij} \leq t_k(v_j)$ for all invocations k. The rate constraints are indicated as attributes on the operation vertices.

Figure 7 shows the flow of the constraint analysis. The given flow graph model with an implementation and a set of timing constraints is first input to deterministic constraint analysis. The constraint satisfiability analysis may proceed in one of the following ways:

1. The constraints are satisfiable and the particular choice of hardware or software implementation is acceptable.
2. A given set of constraints is violated by an implementation, for example, the operation delays may not be small enough for the choice of hardware or software implementation.
3. The constraints are inconsistent and can not be satisfied by any assignment of non-negative delays to operations.
4. Constraint analysis is inconclusive because of the non-determinism in the graph models. We distinguish between the cases where additional information about $\mathcal{ND}$ operations can be used for constraint satisfiability determination from truly undecidable cases. In the latter case, the user may relax the constraint satisfiability criterion by considering *marginal satisfiability* of constraints, that is, constraints are considered satisfiable if certain bounds on the delay of $\mathcal{ND}$ operations are observed. We develop these bounds to ensure constraint satisfaction. Alternative implementations of $\mathcal{ND}$ operations are then explored that improve these bounds.
5. Marginal satisfiability can also be defined probabilistically. In this case, we use additional information about the measure of confidence (for example, acceptable probability of error) to determine the amount of resources (such as buffers) needed. This provides an additional degree of freedom to the system designer for cost and performance tradeoffs.

We now summarize the results in constraint analysis for each of the above cases. Derivations and proofs can be found in [18]. We describe the effect of each type of constraint (operation delay or execution rate) on constraint satisfiability of an implementation. This allows the user to interactively explore the effects of the individual constraints on the system implementation, and, where necessary, modify constraints and implementations.

4.1. *Deterministic Constraint Analysis for Cases 1-3*

Deterministic constraint analysis attempts to determine the existence of a schedule of operations for all possible (and conceivably infinite) values of the delay of the $\mathcal{ND}$ operations. Let us first consider operation delay constraints. For a bilogic relative scheduler, a minimum delay constraint is always satisfiable since for any solution that satisfies Inequality 8, there always exists a solution such that $|\underline{\theta}_{v_j}(v_i)|_\infty \geq \max(\ell(v_j, v_i), l_{ji})$ for each constraint l_{ji}. Therefore, this solution satisfies both Inequalities 7 and 2. On the other hand, a maximum delay constraint may not always be satisfiable. A constraint graph is considered *feasible* if it contains no positive cycle when the delay of $\mathcal{ND}$ operations is assigned to zero. The following theorem [17] lays out necessary and sufficient condition to determine the satisfiability of constraints in presence of $\mathcal{ND}$ operations.

THEOREM 1 (RELATIVE SCHEDULING) *Operation delay constraints are satisfiable if and only if the constraint graph is feasible and there exist no cycles with $\mathcal{ND}$ operations.*

Next, the execution rate constraints are constraints on the time interval between invocations of the same operation. The conditions for checking satisfiability of minimum and maximum rate constraints are described next.

THEOREM 2 (MAXIMUM RATE CONSTRAINT) *A max-rate constraint, R_i, in G is satisfied if $\ell_m(G) \geq \tau \cdot R_i^{-1}$.*

Note that minimum delay constraints and maximum rate constraints are always satisfiable. When $\ell_m(G) < \tau \cdot R_i^{-1}$ the maximum rate constraint, R_i, can be satisfied by an appropriate choice of overhead delay that is applied to every execution of G.

Example 4.1. Maximum rate constraints.

For the process graph model `example` shown in Figure 3 the maximum rate of the `write` operation, determined by $\ell(G_1)$, is 1 cycle^{-1}, whereas the maximum rate of the `read` operation, determined by $\underline{\ell}(G_2) = ((1 \otimes 0) \odot (1 \oplus 3)) \odot (1 \otimes 1) = (3, 5)$ is $\frac{1}{3}$ cycle^{-1}. Any maximum rate constraint larger than or equal to $\frac{1}{3}$ is satisfied by the graph model. □

The lower bound ℓ_m used for checking the satisfaction of maximum rate constraints, also defines the fastest rate at which an operation in the graph model can be executed by a non-pipelined implementation. This points to the necessary condition for meeting a minimum rate constraint. We now consider sufficient conditions for minimum rate constraints. A minimum rate constraint places an upper bound on the interval between successive executions of an operation. A (static) determination of interval of successive executions of an operation that is conditionally invoked is undecidable. That is, there may not exist an upper bound on the invocation interval. For example, consider a statement

```
if (condition)
    value = read (a);
```

There is not enough information to determine the rate of execution of the "read" operation. Additional information, such as how frequently the condition is true, can be used to answer satisfiability in a probabilistic sense. For deterministic analysis purposes, we take a two-step approach to answer constraint satisfiability:

1. First, determine implementation satisfiability *assuming that the condition is always true*. In other words, the only uncertainty is conditional invocation of the graph which may correspond to the body of a process or a loop operation. This is consistent with the interpretation that a timing constraint specifies a bound on the interval between operation executions, but does not imply *per se* that the operation must be executed. Under this assumption, the loops are executed at least once (that is, loops are of the type "repeat-until") since a "while" loop is expressed as a conditional followed by a "repeat-until" loop.

2. Next we use the rate constraint on the "read" operation as the additional information about the invocation of the condition. That is, the rate constraint serves as a *property* of the environment in continuing the rate constraint analysis. This way, constraints are source of additional input which is far more convenient to specify than a priori determination of the probabilities of conditions taken. An alternative approach would be to use simulations to collect data on the likelihood of the condition being true and use it to derive constraint satisfiability.

Using this approach, a minimum rate constraint may lead to bounds on the runtime system or implementation of synchronization operations, indicated by A and B respectively in Figure 7. The actual execution delay or the latency, $\lambda(G)$, refers to the delay of the longest path in G. This path may contain $\mathcal{ND}$ operations in which case the latency can not be bounded. We define the time interval $[t_{k+1}(v_0(G)) - t_k(v_N(G))]$ as the *overhead* $\gamma_k(G)$ where v_0 and v_N refer to the source and sink operations respectively. If G is not a root-level flow graph, then there exists a parent flow graph G_+ that calls G using a link operation, say v. It can be shown that $\gamma_k(G)$ is upper bounded by $\overline{\gamma}$ defined as follows:

$$\gamma_k(G) \leq \overline{\gamma}(G) \triangleq [\ell_M(G_+) + \overline{\gamma}(G_+)] - \ell_m(G) \tag{10}$$

Note that by definition, $\ell_M(G_+) \geq \ell_M(G) \geq \ell_m(G)$, therefore, $\overline{\gamma}$ is always a positive quantity. Clearly, a bound on the overhead delay $\gamma_k(G)$ implies a bound on the invocation interval of G_+, and by induction, bound on the invocation interval of all graphs in the parent hierarchy. In particular, the bound on the invocation interval of the parent process graph G_0 corresponds to a bound on the delay due to the runtime scheduler. This places restrictions on the choice of the runtime scheduler. *Note that a bound on $\gamma_k(G)$ does not imply a bound on the latency λ of G which may, in fact, be unbounded.* We summarize the satisfiability of a minimum rate constraint in the following theorem.

THEOREM 3 (MINIMUM RATE CONSTRAINT WITH NO $\mathcal{ND}$) *A minimum rate constraint on operation, $v_i \in V(G)$, where G contains no $\mathcal{ND}$ operations is satisfiable if the overhead due to the runtime scheduler is bounded as follows:*

$$\overline{\gamma}_o \leq \underbrace{\frac{\tau}{r_i} - \ell_M(G) - \sum_{G_i=G_+}^{G_o} [\ell_M(G_i) - \ell_m(G_i)] - [\ell_m(G_o) - \ell_m(G)]}_{\gamma_M} \tag{11}$$

Thus a minimum execution rate constraint on a graph G that contains no $\mathcal{ND}$ operations is translated as an upper bound γ_M on the delay of the runtime system which is checked by comparing against $\overline{\gamma}_o$ indicated by case A in Figure 7.

Next, in presence of $\mathcal{ND}$ operations in G, the latency, $\lambda(G)$ can no longer be bounded by the longest path length, $\underline{\ell}$, in G. In addition, if G is not a root-level flow graph, its overhead $\gamma(G)$ may also not be bounded by the maximum path length of its parent graph. A relative rate constraint bounds the latency of the graph model and is represented as a

backward edge (that is, a maximum delay constraint) from the sink vertex to the source vertex in the constraint graph model of G. Since G is a connected graph, such a constraint invariably leads to an $\mathcal{ND}$-cycle in the constraint graph. According to Theorem 1, the maximum delay constraint can be satisfied only by bounding the delay of the $\mathcal{ND}$ operation, that is, by transforming the $\mathcal{ND}$ operation into a non-$\mathcal{ND}$ operation. Since $\mathcal{ND}$ operations represent synchronization or data-dependent loop delay, the bounds on the delay of these operations are considered individually:

- Let us first consider synchronization related $\mathcal{ND}$ operations. The implementation delay of a synchronization operation is the *synchronization overhead*, γ_w. Due to the availability of multiple concurrent execution streams in hardware, this overhead is zero. For software, γ_w, delay is determined by the implementation of the wait operation by the runtime scheduler. For example, a common implementation technique is to force a *context switch* in case an executing program enters a wait state. Here, γ_w would be twice the context-switch delay to account for the round-trip delay. The minimum rate constraint bounds the execution rate that must be supported by a given runtime implementation. With this interpretation, the $\mathcal{ND}$ operations are considered non-$\mathcal{ND}$ operations with a fixed delay, γ_w. Determination of this bound is indicated by Case B in Figure 7.

- The data-dependent loop operations use a data-dependent *loop index* that determines the number of times the loop body is invoked for each invocation of the loop link operation. The delay offered by the loop operation is its loop index times the latency of the loop body. As mentioned earlier, at the leaf-level of graph hierarchy, the latency of the loop body is given by its path length vector. The elements of a path length vector consists of lengths of all paths from source to sink and these are bounded. In the case the constrained graph contains a loop operation, v, on a path from source to sink, the minimum rate constraint can be seen as a bound on the number of times the loop body G_v corresponding to the loop operation, v, is invoked. This bound on loop index, x, is given by Equation 12 that is derived later. As in the case of constraint satisfiability for conditionals, this bound $\overline{x}$ is then treated as a *property* of the loop operation, consequently making it a non-$\mathcal{ND}$ operation with a bounded delay for carrying out further constraint analysis. Verification of these bounds requires additional input from the user. This is indicated by case C in Figure 7.

The satisfiability of a minimum execution rate constraint also includes a bound on the invocation delay γ of G. Clearly, a bound on $\gamma(G)$ implies a bound on the latency of G_+ which is equivalent to a minimum rate constraint on an operation in G_+. Thus, the minimum rate constraint is propagated upwards in the graph hierarchy. However, as discussed earlier this minimum rate constraint does *not* bound the loop index of link operation associated with G. The constraint satisfiability is then continued until G_+ corresponds to a process body, G_o. In practice, relative minimum rate constraints are more frequently used. For these constraints, the upward propagation of constraint stops at the graph model, G', relative to which the constraint is specified. This is reflected

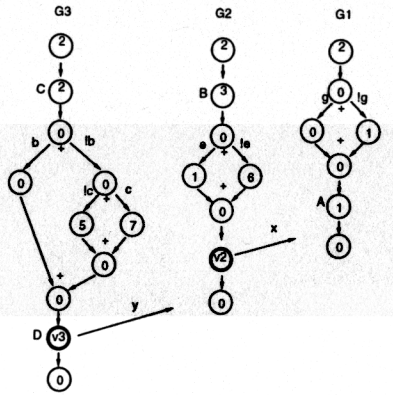


Figure 8. Example 4.2.

by assigning a zero value to the overhead term $\overline{\gamma}(G')$ in Equation 12. The following theorem summarizes the condition for checking constraint satisfiability.

THEOREM 4 (MINIMUM RATE CONSTRAINT WITH $\mathcal{ND}$) *Consider a flow graph G with an $\mathcal{ND}$ operation v representing a loop in the flow graph. A minimum rate constraint r_i on operation $v_i \in V(G)$ and $v_i \neq v$ is satisfiable if the loop index, x_v indicating the number of times G_v is invoked for each execution of v is less than the bound $\overline{x}_v$ where*

$$\overline{x}_v \triangleq \left\lfloor \frac{\tau r_i^{-1} - \overline{\gamma}(G) - \ell_M(G) + \mu(v)}{\ell_M(G_v)} \right\rfloor + 1 \tag{12}$$

where $\mu(v)$ refers to the mobility of operation v and is defined as the difference in length of the longest path that goes through v and ℓ_M.[3] The overhead bound, $\overline{\gamma}(G)$ is defined by Equation 10.

In presence of multiple $\mathcal{ND}$ operations that lie on the same path from source to sink, a minimum rate constraint on an operation v_i is satisfied if each loop index x_v is bounded as in Equation 12 above for all $\mathcal{ND}$ operations $v \in V(G)$ and $v \neq v_i$, and $\sum_{v \in \mathcal{ND}} \frac{x_v}{\overline{x}_v} \leq 1$. The last condition can be verified either deterministically, by substituting x_v by a (user-supplied) upper bound or by treating x_v as a random variable and using its expected value for x_v. Both cases require additional input to verify constraint satisfiability and are considered in the next section in the context of probabilistic satisfiability.

Example 4.2. Bound on loop index due to minimum execution rate constraint.

Consider a minimum rate constraint of 0.02 /cycle on operation 'B' in graph model, G_2 shown in the figure above. Let the maximum delay due to the runtime scheduler be $\overline{\gamma}_M = 0$ (for example, hardware implementation). The bound on the loop index for operation v_2 is calculated as follows:

$$\overline{\gamma}(G_2) = \ell_M(G_3) + \overline{\gamma}_M - \ell_m(G_2) = 26 + 0 - 9 = 17$$

$$\overline{x}_2 = \left\lfloor \frac{\tau r_B^{-1} - \overline{\gamma}(G_2) - \ell_M(G_2) + \mu(v_2)}{\ell_M(G_1)} \right\rfloor + 1 = \left\lfloor \frac{50 - 17 - 15 + 0}{4} \right\rfloor + 1 = 5.$$

With this bound on loop index, the $\mathcal{ND}$ operation v_2 has a bound on its delay of 20 cycles. On the other hand, a *relative* rate constraint, $r_B^{G_2}$ of 0.02 /cycle leads to a bound on loop index of $\overline{x}_2 = \left\lfloor \frac{50-0-15+0}{4} \right\rfloor + 1 = 9$. with this bound the delay of v_2 is less than 36 cycles. □

In summary, satisfaction of the bounds on delay of $\mathcal{ND}$ operations requires additional information from their implementations (such as context switch delay, possible loop index values) against which the questions about satisfiability of minimum rate constraint can be answered. Because of these bounds, there is now a certain *measure* of constraint satisfiability that approaches certainty as the derived bound approaches infinity. More importantly, having bounds derived from timing constraints makes it possible to seek transformations to the system model which tradeoff these measures of constraint satisfiability against implementation costs. In the next section, we examine conditions under which these bounds can be extended by modifying the structure of the flow graphs with $\mathcal{ND}$ cycles.

4.2. *Bound Improvement for Cases 4 & 5*

In general, the loop body G_v consumes some data that is produced by the calling body G and produces some data that is consumed by G. We are interested in cases where the data transfer happens only in one direction, for example, from G to G_v. The data consumed by G_v is defined by the storage that is common to both G and G_v, i.e., $M(G) \cap M(G_v)$. As shown in Figure 9, the called graph can be modeled as a consumer and the calling graph as a producer.

There are various ways of modeling this dynamics of the producer-consumer system. Previous work using this approach to rate constraint analysis is by Amon and Borriello [19], where the producer-consumer system is modeled as a deterministic queue with bounds on maximum and minimum rates of data production and consumption. A bound on queue depth is developed based on the transient behavior of the queueing system over a finite time period. In [20] the authors use state reachability analysis to determine bound on the buffer depth for a system modeled as a network of interacting finite state machines. The primary difference with the producer-consumer formulation presented here and the previous work is that the queueing system created by $\mathcal{ND}$ cycles is not deterministic; instead the rate of data consumption depends upon the *value* of the data. It is more appropriately modeled as either as a queueing system with multiple arrivals and fixed rate of consumption or as a system with fixed arrivals with variable rate of consumption. We take the latter approach as explained below.

Let x be the index variable associated with the loop operation, indicating the number of times the loop body G is invoked for an invocation of the loop operation. The fastest rate of production of data by the producer model G is given by the inverse of its minimum latency. This rate of production is fixed by an imposed minimum rate constraint relative to G. The rate of consumption of data, however, is variable and depends upon the actual

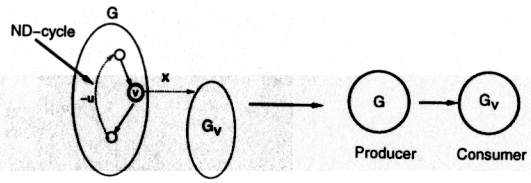


Figure 9. Modeling an $\mathcal{ND}$ loop as a producer-consumer system.

value of the loop index. That is, the larger the loop index, the longer it takes for the consumer to consume the data.

For a given rate constraint, the bound on the value of the loop index was computed in Section 4.1. To maintain correct behavior the producer model must *block* if at any time, the loop index exceeds this bound. This blocking leads to violation of the imposed rate constraint. For this producer-consumer system, since the data transfer occurs only from G to G_v, G need not block for completion of G_v if the loop index is bounded as above. Therefore, we can replace the unknown delay $\mathcal{ND}$ operation, v, by a fixed delay operation (i.e., non-$\mathcal{ND}$) which consists in transferring data to a waiting loop body *without* waiting for completion of the loop operation. Let $\underline{\ell}'(G)$ be the new length vector of G. For any invocation of loop link operation v, the executions of the loop body G_v must complete before the link operation is restarted. That is,

$$\ell'_m \geq x_v \cdot \ell_M(G_v) \quad \Rightarrow \quad x_v \leq \frac{\ell'_m}{\ell_M(G_v)} \tag{13}$$

This defines an upper bound on the value of the loop index, x_v. For a given producer-consumer system, we define the *blocking limit*, B_1, as the upper bound on value taken by the loop index beyond which the calling body must block before restarting:

$$B_1 = \left\lfloor \frac{\ell'_m}{\ell_M(G_v)} \right\rfloor \tag{14}$$

B_1 provides a conservative bound on the loop index value based on the fastest rate of production and the slowest rate of consumption of data.

Let us now consider an implementation of the producer-consumer system that is connected by a buffer of depth greater than one. Note that due to the semantics of the loop operation there is always a 1-deep buffer between producer and consumer. In this case, the blocking limit is extended to

$$B_k = \left\lfloor \frac{k \cdot \ell'_m}{\ell_M(G_v)} \right\rfloor \tag{15}$$

where k is the number of empty spots in the buffer ($\leq$ buffer depth, n). For any loop index value greater than B_1 the execution of consumer model (i.e., body of the loop) spans across successive executions of the link operation in the producer model and,

therefore, occupies a place in the buffer. We assume that each invocation of the loop link operation always produces a loop index value greater than or equal to one. That is, it is not the case that an invocation of the loop link operation does not enqueue data into the buffer. This is needed to keep the software synchronization simple with low overheads.[4]

A buffer can help in meeting rate constraints only in situations where there is irregularity in the values of the loop index and its *average* value still observes the blocking limit, B_1. In other words, given a finite depth buffer, the producer will always block eventually if the average rate of production is greater than the rate of consumption, that is the average value of loop index exceeds B_1. An algorithm for average rate matching is described in [21]. However, the time it takes to fill up the buffer depends upon the transient behavior of the producer-consumer queueing system. The producer-consumer system itself is conditionally invoked at a rate that is determined by the runtime scheduler or the parent graph model in which the producer-consumer system resides. A buffered implementation requires that for each conditional invocation of the producer-consumer there is a fixed number of unconditional invocations of the producer-consumer system and at the beginning of each conditional invocation, the producer-consumer system is started from the initial state, that is, all buffers are empty.

4.3. *Probabilistic Constraint Analysis for Case 5*

The notion of an unbounded delay does not automatically imply infinite delay, but the *possibility* that for any given value, d, the delay offered by the $\mathcal{ND}$ operation may exceed d. The situation is addressed by relating the possibility of constraint violation to the possibility of exceeding a specified bound on the delay of the $\mathcal{ND}$ operations by treating the operation delay δ as a random variable. For a given $\mathcal{ND}$-cycle, a constraint violation occurs if the delay, δ, offered by the $\mathcal{ND}$ operation exceeds a (deterministic) bound, f. We define violation error as $\max((\delta - f(u), 0)$.

There are various ways to carry out this probabilistic analysis. For a given distribution of δ one approach is to find supportable rate constraints that minimize some measure (absolute, mean-square, etc.) of the violation error. An alternative is to determine supportable rate constraints that limit the probability of constraint violation below some acceptable limit, ϵ. Finally for a given set of rate constraints, find an appropriate size of the buffer that bounds the probability of constraint violation below a given limit. This approach also fits with the general probability of failure for different parts of the system design. In principle, once a constraint violation is brought below a certain error probability that is comparable to probability of failure of other parts of system design, the corresponding constraint can be considered satisfiable.

For illustration purposes, let us consider a max-delay constraint of Equation 3, $t_k(v_j) - t_k(v_i) \leq u_{ij}$. Deterministic satisfiability requires that this equation holds true for all values of k. We consider a max-delay timing constraint *marginally satisfiable* if for a given bound ϵ, $(0 \leq \epsilon \leq 1)$ $\Pr\{t_k(v_j) - t_k(v_i) > u_{ij}\} \leq \epsilon$. That is, for each k, the check for constraint satisfiability is considered a trial. A marginally satisfiable constraint is then found to be satisfied by a given implementation if over a large number of such trials,

the (relative frequency) probability of constraint violation is within a certain specified bound.

Given the loop index of $\mathcal{ND}$ operation as *a random variable* we formulate the problem of constraint satisfiability as a determination of expected number of data items waiting and thus the size of the buffer required when the loop index has some nonzero probability of exceeding the blocking limit. A full buffer leads to blocking of the producer graph execution, and therefore a constraint violation occurs. The following presents statement of the problem.

> Given a constraint graph model, G with an $\mathcal{ND}$ cycle Γ caused by a *backward edge* with weight u. Assume that the loop index is a random variable with expected value, $\overline{x}$ and variance σ_x. Let $\underline{\ell}(G_v)$ be the length of the loop body, G_v.
>
> **Problem P1:** *Find a bound N on the buffer size k such that the probability,* $\Pr\{G \text{ blocks}\} \leq \epsilon$ *for all* $k \geq N$.

An alternative formulation of the above problem is to determine the value of the backward edge that would satisfy the blocking limit. This value can then be propagated to determine the achievable execution rate.

> **Problem P2:** *Given a buffer size of k find an upper bound $\overline{u}$ on the weight, u, of the backward edge that causes the $\mathcal{ND}$ cycle Γ, such that* $\Pr\{G \text{ blocks}\} \leq \epsilon$ *for all* $u \leq \overline{u}$. *Note that weight of a backward edge is a negative quantity, therefore, an upper bound $\overline{u}$ on u refers to a lower bound on the absolute value of the weight, u.*

Problems P1 and P2 are related. For a given acceptable error rate, probabilistic constraint satisfiability either seeks implementations that meet the required performance (P1) or seeks achievable performance that meets required implementation costs (P2). Note that in the limit $\epsilon \to 0$, the problems seek a deterministic solution.

Solution to problems P1 and P2 above depends upon the choice of a probability distribution function, $F_X(\cdot)$ for the random variable x. For analytic simplicity, we treat the random variable as continuous variable and use it to derive approximations to the value of the corresponding discrete parameters. We consider the case when the random variable is unbounded and exponentially distributed. The exponential distribution is chosen due to the fact that the value of the loop index is directly proportional to the interval of time it takes for the consumer to consume a data from the buffer, i.e., the delay offered by the $\mathcal{ND}$ operation. It has been shown that the exponential distribution is the most random distribution that can be used and thus the most conservative approach [22]. For an exponentially distributed loop index, the rates of data production and consumption follow a Poisson distribution. That is, the times at which data is produced or consumed (i.e., schedule of I/O operations) is *uniformly* distributed. In other words, the k start times of an I/O operation over an interval [0, T] are distributed as the order statistics of k uniform random variables on [0, T]. The following theorem provides a bound on the buffer depth that is normalized by the length ratio $\frac{\ell'_m}{\ell_M(G_v)}$.

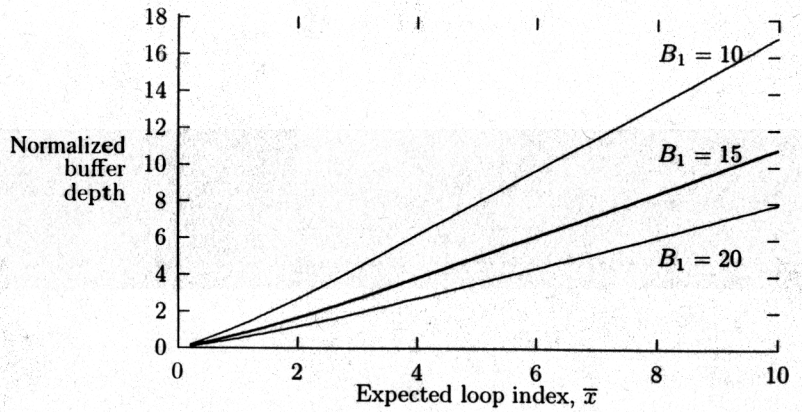


Figure 10. Buffer depth for exponential distributions ($\epsilon = 0.01\%$)

THEOREM 5 *For a given error probability, ϵ, the following express the bounds on buffer depth, k*

$$N_e = \left\lceil \frac{\ln \epsilon}{\ln(\frac{B_1}{\overline{x}}) - \ln(e^{B_1/\overline{x}} - 1)} \right\rceil \approx \left\lceil \frac{-\ln \epsilon}{\frac{B_1}{\overline{x}} - \ln(\frac{B_1}{\overline{x}})} \right\rceil \tag{16}$$

where $\overline{x} < B_1$ is the expected value of the loop index, x and B_1 is the blocking limit for 1-deep buffer.

Note that as $\epsilon \to 0$, $N_e \to \infty$. Figure 10 shows required buffer depth for an $\epsilon = 0.01\%$. A solution to problem P2 requires determination of an achievable blocking limit, B_1 for a given buffer depth, k. From the blocking limit we can determine the fastest rate of data production, or the maximum value of u and hence the supportable rate constraint. The analytic solution to Equation 16 is complex and not very useful as a general formula. As an alternative, the Equation 16 can be solved numerically using a solver such as Maple [23] for the blocking limit, B_1 for a given values of buffer depth $k = N_e$, expected loop index, $\overline{x}$ and the error probability, ϵ.

We now consider the use of software for embedded systems. We first describe the runtime system as it is used in a hardware-software implementation followed by the generation of software from the flow graph model.

5. A Model for Software and Runtime System

The concept of a *runtime system* applies to systems containing a set of operations or tasks and a set of resources that are used by the tasks. Operations may have dependencies

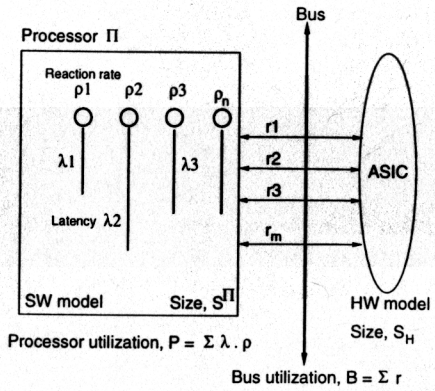


Figure 11. Software model to avoid creation of $\mathcal{ND}$ cycles.

that impose a (partial) ordering in which the tasks can be assigned to resources. In general a runtime system consists of a *scheduler* and a *resource manager*. The task of the runtime scheduler is to pick up a subset of tasks from the available set of tasks to run at a particular time step. The resource manager consists of two components: a resource *allocator* and a resource *binder*. The allocator assigns a subset of resource to a subset of tasks, whereas a binder makes specific assignments of resources to tasks. Depending upon the nature and availability of tasks and resources, some or all of these activities can be done either *statically* or *dynamically*. A static schedule, allocation or binding makes the runtime system simpler.

In this general framework, most synthesized hardware uses static resource allocation and binding schemes, and static or relative scheduling techniques. Due to this static nature, operations that share resources are serialized and the binding of resources is built into the structure of the synthesized hardware, and thus there are always enough resources to run the available set of tasks. Similarly, in software, the need for a runtime system depends upon whether the resources and tasks (and their dependencies) are determined at compile time or runtime.

Since our target architecture contains only a single resource, that is, the processor, the tasks of allocation and binding are trivial, i.e., the processor is allocated and bound to all routines. However, a static binding would require determination of a static order of routines, effectively leading to construction of a single routine for the software. This would be a perfectly natural way to build the software given the fact that both resources and tasks and their dependencies are all statically known. However, due to the presence of $\mathcal{ND}$ operations in software, a complete serialization of operations may lead to creation of $\mathcal{ND}$ cycles in the constraint graph, which would make constraint satisfiability determination difficult. A solution to this problem is to construct software as a set of *concurrent program threads* as sketched in Figure 11. A thread is a linearized set of operations that may or may not begin with an $\mathcal{ND}$ operation. Other than the beginning

$\mathcal{ND}$ operation, a thread does not contain any $\mathcal{ND}$ operations. The latency λ_i of a thread i is the sum of the delay of its operations without including the initial $\mathcal{ND}$ operation whose delay is included into the delay due to the runtime scheduler. The reaction rate ϱ_i of thread i is the rate of invocation of the program thread per second. In presence of concurrent multiple threads of operation, a hardware-software system is characterized by following two parameters: (a) *Processor utilization*, $\mathcal{P}$ defined as $\mathcal{P} \triangleq \sum_{i=1}^{n} \lambda_i \cdot \varrho_i$ and (b) *Bus utilization*, $\mathcal{B}$ as a measure of the total amount of communication taking place between the hardware and software. For a set of m variables to be transferred between hardware and software, $\mathcal{B} \triangleq \sum_{j=1}^{m} r_j$ where r_j is the inverse of the minimum time interval between two consecutive samples for variable j. In general, $\mathcal{B}$ is bounded by a given bus bandwidth as a function of the bus cycle time and memory access time.

Bounds on $\mathcal{P}$ are developed based on the type of runtime system selected. We have so far implemented only non-preemptive runtime scheduling techniques, where a program thread executes either to its completion, or to the point where it detaches itself voluntarily. In general, a scheduler may be preemptive, that is, it may preempt a running thread by another program thread. Preemption leads to improved response time to program threads at the increased cost of implementation. The implementation of multiple program threads in a preemptive runtime environment requires handling of additional states (in addition to being *detached* or *running*) for the program threads which adds to the overhead delay caused by the runtime scheduler. Timing constraint analysis for preemptive runtime schedulers is beyond the scope of this work.

The basis for analysis of the runtime scheduler is provided by the intuition that it is sufficient to show the feasibility of the scheduler by considering the case when all threads are enabled at the same time [24]. A necessary condition to ensure that the reaction rates of all program threads can be satisfied by the processor is given by the constraint that processor utilization is kept below unity, i.e., $\mathcal{P} \leq 1$. However, this condition is not sufficient. Consider, for example, the case when the software contains a program thread with a long latency but very low reaction rate. Such a program thread will bound the achievable reaction rate for all program threads below the inverse of its latency even though from a processor utilization point of view higher reactions rates may be possible. For a program thread in a non-preemptive, non-prioritized FIFO runtime scheduler, a sufficient condition to ensure satisfaction of its reaction rate, ϱ, is given by the following condition:

$$\frac{1}{\varrho} \geq \sum_{\text{thread } k} \lambda_k \tag{17}$$

In case all the program threads are enabled simultaneously, a program thread is enabled again only after completing execution of all other program threads. Note that this condition is only sufficient. It is also necessary and sufficient for *independent* threads. In case of dependent program threads, only a subset of the total program threads are enabled for execution, that is, those threads that do not depend upon execution of the current thread. Therefore, the necessary condition will be weaker and can be estimated by summation over enabled program threads in Inequality 17. A sufficient condition for software reaction rate satisfiability is to ensure that

$$\frac{1}{\varrho_{\max}} \geq \sum_{\forall \text{ threads } k} \lambda_k \tag{18}$$

where $\varrho_{\max} = \max_i \varrho_i$ defines the maximum reaction rate over all program threads.

5.1. *Estimation of Software Properties*

Software size: A software implementation of a flow graph model G is characterized by a software size function S^{Π} that refers to the size of program S_p^{Π} and static data S_d^{Π} necessary to implement the corresponding program on a given processor Π. A processor is characterized by its instruction set architecture (ISA) which consists of its instructions and the memory model. We assume that the processor is a general-purpose *register* machines with only explicit operands in the instruction (no accumulator or stack). The memory addressing is assumed to be byte-level. The processor cost model Π consists of delays for a given basic set of assembly language operations, memory access time, interrupt response time. We assume the generated software to be *non-recursive* and *non-reentrant*, therefore, the size of the software can be statically computed. For a system model Φ,

$$S^{\Pi}(\Phi) = \sum_{G_i \in \Phi} S^{\Pi}(G_i) = \sum_{G_i \in \Phi} [S_p^{\Pi}(G_i) + S_d^{\Pi}(G_i)] \tag{19}$$

We postpone the discussion on estimation of program size to later in this section. The set $\mathcal{S}_d^{\Pi}$ consists of storage required to hold variable values across operations in the flow graph and across the machine operations and $|\mathcal{S}^{\Pi}| = S^{\Pi}$. This storage can be in the form of specific memory locations or the on-chip registers, since no aliasing of data items is allowed in input HDL descriptions. In general, $\mathcal{S}_d^{\Pi}(G)$ would correspond to a subset of the variables used to express a software implementation of G, that is,

$$S_d^{\Pi}(G) \leq |M(G)| + |P(G)| \tag{20}$$

where $M(G)$ refers to the set of variables used by the graph G and $P(G)$ is the set of input and output ports of G. This inequality is because not all variables need be *live* at the execution time of all machine instructions. At the execution of a machine instruction, a variable is considered live if it is input to a future machine instruction. In case $\mathcal{S}_d^{\Pi}(G)$ is a proper subset of the variables used in software implementation of G, additional operations (other than the operation vertices in G) are needed to perform data transfer between variables and their mappings into the set $\mathcal{S}_d^{\Pi}(G)$. In case $\mathcal{S}_d^{\Pi}(G)$ is mapped onto hardware registers, this set of operations is commonly referred to as *register assignment operations*. The number of operations needed to perform the requisite data transfer depends upon the scheduling scheme chosen. Typically in software compilers a schedule of operations is chosen according to a solution to the register allocation problem. The exact solution to the register assignment problem, for general programs, is NP-complete [25]. Hence heuristics solutions are commonly used. Most popular heuristics for code generation use a specific order of execution of successor nodes (e.g., left neighbour first) to reduce the size of $\mathcal{S}_d^{\Pi}$ [26].

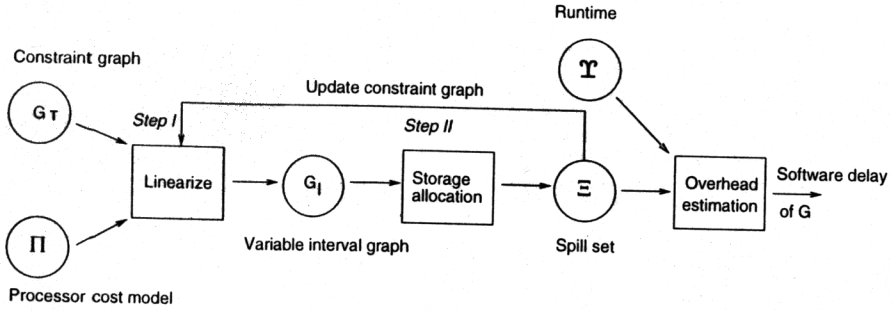


Figure 12. Software delay estimation flow.

In contrast to the register assignment in conventional software compilers which perform simultaneous register assignment and operation linearization, we use a two-step approach to estimate of data storage and software delays resulting from additional memory operations: (a) **linearize operations**, that is, find a schedule of operations; (b) estimate register/memory **data transfer operations**. This two-step approach preserves the flexibility in operation scheduling which must take into account timing constraints not used in traditional software compilers. Figure 12 illustrates the steps in estimation of software performance.

Software performance: Previous work in software estimation has been to annotate programs with relevant timing properties [27], [28]. Syntax-directed delay estimation techniques have been tried [29], [30] which provide quick estimates based on the language constructs used. In [31] the authors describe use of dynamic execution information to computing execution times. Timing estimation at the basic block-level is presented in [32]. We perform a static delay estimation on flow graph models for both hardware and software using the flow graphs. A software delay consists of two components: delay due to operations in the flow graph model, and delay due to the run-time environment. As described in Section 3 the effect of runtime is modeled as a constant overhead delay to each execution of the flow graph corresponding to the process graph. The software delay depends upon the delay of operations in the flow graph model *and* operations related to storage management.

We assume that the system bus is always available for instruction/data reads and writes and that all memory accesses are aligned. Each operation v in the flow graph is characterized by a number of read accesses, $m_r(v)$, a number of write accesses, $m_w(v)$ and a number of assembly-level operations, $n_o(v)$. The software operation delay function, η, is computed as follows:

$$\eta(v) = \sum_{i=1}^{n_o(v)} t_{op_i} + (m_r(v) + m_w(v)) \times m_i \tag{21}$$

where the operand access time, m_i, is the sum of effective address computation time and memory access time for memory operands. For some instruction-sets, not all possible combinations of ALU operations and memory accesses are allowed and often operand access operations are optimized and overlapped with ALU operation executions thus reducing the total execution delay. Due to this non-orthogonality in ALU operation execution and operand access operations, the execution time of some operations is often overestimated from real execution delays. In the one-level memory model, the number of read and write accesses depends upon the fanin and fanout of the operation.

Use of operation fanin and fanout to determine memory operations provides an approximation for processors with very limited number of available general purpose registers. Most processors with load-store (LS) instruction set architectures feature a large number of on-chip registers. Therefore, this procedure must be refined to include the effect of on-chip registers. This is considered next.

Estimation of register, memory operations: The number of read and write accesses is related to the amount and allocation of static storage, $\mathcal{S}_d^{\Pi}(G)$. Since it is difficult to determine actual register allocation and usage, estimation rules are devised. Let $G^D = (V, E^D)$ be the data-flow graph corresponding to a flow graph model, where every edge, $(v_i, v_j) \in E^D$ represents a data dependency. Vertices with no predecessors are called source vertices and vertices with no successors are defined as sink vertices. Let n_i and n_o be the number of source and sink vertices respectively. Let $i(v), o(v)$ be the indegree and outdegree of vertex v and finally, let r_r and r_w be the number of register read and write operations respectively.

Each data edge corresponds to a pair of read, write operations. These read and write operations can be either from memory (Load) or from already register-stored values. Register values, in turn, are either a product of load from memory or a computed result. Clearly, all values that are not computed need to be loaded from memory at least once (contributing to m_r). Further, all computed values that are not used must be stored into the memory at least once (and thus contribute to m_w). Let R be the total number of unrestricted (i.e., general purpose) registers available (not including any registers needed for operand storage). In case the number of registers R is limited, it may cause additional memory operations due to *register spilling*. A register spill causes a register value to be temporarily stored to and loaded from the memory. This spilling is fairly common in complex instruction-set processors and coupled with non-orthogonal instruction sets, result in a significant number of data transfers either to memory or to register operations (the latter being the most common). The actual number of spills can be determined exactly given a schedule of machine-level operations. Since this schedule is not under direct control, therefore, we concentrate on bounds on the size of the spill set.

Case I: $R = 0$ In this limiting case, for every instruction, the operands must be fetched from memory and its result must be stored back into the memory. Therefore, $m_r = |E|$ and $m_w = |V|$. Note that each register read results in a memory read operation and each register write results in a memory write operation, $(r_r = m_r)$ and $(r_w = m_w)$.

Case II: $R \geq R_l$ where R_l is the maximum number of live variables at any time. In this case no spill occurs as there is always a register available to store the result of every operation. Therefore, $m_r = n_i \leq |V| \leq |E|$ and $m_w = n_o \leq |V|$.

Case III: $R < R_l$ At some operation v_i there will not be a register available to write the output of v_i. This implies that some register holding the output of operation v_j will need to be stored into the memory. Depending upon the operation v_j chosen, there will be a register spill if the output of v_j is still live, that is, it is needed after execution of operation v_i. Of course, in the absence of a spill, there will be no effect of register reallocation on memory read/write operations. Let $\Xi \subset V$ be the set of operations that are chosen for spill. Therefore,

$$m_r = n_i + \sum_{\Xi} o(v_i) \leq \sum_{V} o(v_i) = |E| \quad \text{and} \quad m_w = n_o + |\Xi| \leq |V|$$

Clearly, the choice of the spill set determines the actual number of memory read and write operations needed. In software compilation, the optimization problem is then to choose a spill set, Ξ such that $\sum_{\Xi} o(v)$ is minimized. This is another way of stating the familiar register allocation problem. Our spilling heuristic is described in [18].

Finally, the program size, S_p^{Π} is a measure of the instructions generated by the compiler. For each operation in the graph model, we associate *rvalue* and a *lvalue* [26] that are needed in the corresponding generated C code. The rvalue is the result of evaluation of an expression (or simply the right-hand side of an assignment). For most assignment statements, the left side generates a lvalue and the right side generates a rvalue. The program size is then approximated by the sum over rvalues associated with operation vertices. This approximation is based on the observation that the number of instructions generated by a compiler is related to the number of rvalues. This is only an upper bound since global optimizations, in general, may reduce the total number of instructions generated.

5.2. *Operation Linearization*

We now consider the issue of generation of a program thread from the flow graph model. A program thread consists of a linearized set of operations in the flow graph. A flow graph specifies a partial order on the execution of its operations. Program threads are implemented to ensure that this partial order is always maintained. Depending upon the $\mathcal{ND}$ operations, multiple program threads may be created from a flow graph model. Two programs are either hierarchically related or concurrent. In case of hierarchically related program threads since the dependencies between threads are known, these are built into the threads as additional *enabling* operations. During execution, a program thread is in one of the following three states: *detached*, *enabled* or *running*. Before running, a thread must be enabled first. In case of concurrent program threads in a single-processor software implementation, concurrency between program threads is achieved by using an interleaved execution. In order for an interleaved execution of program threads to work

correctly, the input flow graph must be *serializable*, that is, there exists a complete order of operations that performs exactly as specified in the original graph model.

Serializability is a concern when a graph model contains operations with shared storage that may be enabled concurrently. (We consider two operations concurrent if their executions overlap in time). An operation in the flow graph can take one of the following actions on a given variable, u: (a) *Defines* u. This is the case when the operation corresponds to an assignment statement where the variable u appears on the left-hand side (lhs) of the assignment, for instance `u = <rhs>`; (b) *Uses* u. This is the case when the operation corresponds when u appears in an expression or the right-hand side (rhs) of an assignment; (c) *Tests* u. This is the case when u is a part of a conditional or loop exit condition. We note that the distinction between variable use and test operations is new and specific to the problem of hardware-software co-synthesis. A variable that is only tested (and not used) is considered a control variable. This distinction between use and test is made to model different strategies for control and data transfer across a hardware-software partition. With a data-transfer across hardware-software partition, use of a data-variable leads to its deletion since that can be utmost one use of a data value, whereas a control transfer is reflected by a (condition) variable that can be consumed (i.e., tested) by several operations until it is updated.

We consider a variable *private* to an operation if it is used or tested by that operation alone. A variable that is used or tested in multiple (concurrent) operations is considered a *shared* variable. For an atomic (hardware or software) implementation of an operation, a flow graph is *serializable* if the storage $M(G)$ can be partitioned into shared and private variables such that only the private variables can be both used and defined by the same operations. Variables that are both used and tested can be shared between concurrent operations. Thus the serializability of a graph can be easily checked by examination of definition and use operations for its variables. In case, an operation defines and uses a shared variable, this operation can be broken into two operations where one defines, and the other uses the variable. A data-dependency between the resulting operations ensures correct order of serialization.

Transformation of unserializable flow graphs: In general, a given flow graph may not be serializable. This happens when two operations are required to be executed in parallel *regardless* of actual operation dependencies, i.e., using "forced-parallel" semantics of HardwareC. Forced-parallel semantics is often used to specify behavior of hardware blocks declaratively, and for correct interaction with the environment assuming a particular hardware implementation. For instance, assuming a master-slave flip-flop for a storage (static) variable, concurrent read and write operations to the variable may refer to operations on two separate values of the variable corresponding to the slave and master portions of the flip-flop. Forced-parallel operations are serialized using the following procedure:

1. *Decompose* concurrent operations into multiple *simple* operation using intermediate variables; a simple operation either uses, defines or tests a shared variable;
2. Add dependencies between simple operations that observe polarization of the acyclic flow graph from its source to sink;

The new graph can now be linearized into one or more program threads.

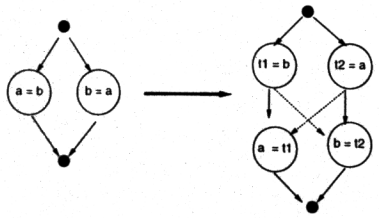


Figure 13. Creation of serializable graphs.

Example 5.1. Creation of serializable flow graphs. Consider the following fragment representing a *swap* operation:

```
static int a, b;
<
a = b;
b = a;
>
```

This is translated into two sets of simple operations as shown in Figure 13.

For the new graph model, any of the four valid serializations lead to correct implementation of the original concurrent operations. □

Linearization Techniques: For a serializable graph, G, linearization refers to finding a complete order of operations in $V(G)$ that is a consistent enumeration of the partial order in G. A linearization also specifies the order in which data is transferred between operations. A software implementation of variable storage uses a two-level hardware storage, memory and registers to implement inter-operation data transfer. We consider the following two problems:

Problem P3: [Feasible Linearization]: *For a given timing constraint graph model $G_T = (V, E, \Delta)$ and an upper bound R on the maximum number of registers available for arbitrary variable storage, find a linearization of operations in V such that for a given choice of the spill set, $\Xi_R \subset V$, all timing constraints are satisfied.*

Problem P4: [Optimum Linearization]: *For a given timing constraint graph model $G_T = (V, E, \Delta)$ and an upper bound R on the maximum number of registers available for arbitrary variable storage, find a linearization of operations in V and a spill set, $\Xi_R \subset V$, such that all timing constraints are satisfied, and the size of spill set $|\Xi|$ is minimum.*

Problems P3 and P4 are related. P4 is a storage optimization problem over possible solutions to P3. We solve Problem P4 by incorporating spill set estimation in pruning the linearization results obtained as a solutions to P3. In the presence of timing constraints,

the problem of linearization can be reduced to the problem of 'task sequencing of variable length tasks with release times and deadlines' which is shown to be NP-complete in the strong sense [25]. It is also possible that there exists no linearization of operations that satisfies all timing constraints. An exact ordering schemes under timing constraints is described in [12] that considers all possible valid orders to find one that meets the imposed timing constraints. In [33] the authors present an operation ordering scheme for a static non-preemptive software model using *modes*. We use a heuristic ordering based on a *vertex elimination scheme* that repetitively selects a zero in-degree vertex (i.e., a root vertex) and outputs it. The algorithm consists of following three steps:

1. Select a root operation to add to the linearization,
2. Perform timing constraint analysis to determine if the addition of the selected root operation to the linearization constructed thus far leads to a feasible complete order, else select another root vertex,
3. Eliminate selected vertex and its dependencies, update the set of root operations.

The main part of the heuristic is in selection of a vertex to be output from among a number of zero in-degree vertices. This selection is based on the criterion that the induced serialization does not create a positive-weight cycle in the constraint graph. Among the available zero in-degree vertices, we select a vertex based on the least value of its *urgency*. This measure is derived from the intuition that a necessary condition for existence of a feasible linearization (i.e., scheduling with a single resource) is that the set of operations have a schedule under timing constraints *assuming unlimited resources*. A feasible schedule under no resource constraints corresponds to an assignment of operation start times according the lengths of the longest path to the operations from the source vertex. Since a program thread contains no $\mathcal{ND}$ operations, the length of this path can be computed. However, this path may contain cycles due to the backward edges created by the timing constraints. A feasible schedule under timing constraints is obtained by using the operation slacks to determine the longest path delays to operations. The length of the longest path is computed by applying an iterative algorithm that repetitively increases the path length until all timing constraints are met. This has the effect of moving the invocation of all closely connected sets of operations to a later time in the interest of satisfying timing constraints on operations that have been already linearized. This scheduling operation either fails when it detects a positive cycle in the constraint graph or returns a feasible schedule. In case the algorithm fails to find a valid assignment of start times, the corresponding linearization also fails since the existence of a valid schedule under no resource constraints is a necessary condition for finding a schedule using a single resource. In case a feasible schedule exists, the operation start times under no resource constraints define the urgency of an operation.

Since the condition for a feasible linerization used in the urgency measure is (necessary but) not sufficient, therefore, the heuristic may fail to find any feasible linearization while there may exist a valid ordering. Under such conditions a (computation-intensive) exact ordering search that considers all possible topological orders can be applied. In practice, since the heuristic procedure uses the structure of the constraint graph to quickly determine an operation linearization it performs operation linearization in polynomial time for most cases [34].

We now discuss the problem of system partitioning and how it is affected by our choice of the software and system architecture.

6. Partitioning

In practice, often a partitioning into hardware and software portions is carried out at levels of abstraction that are higher than what can be modeled using conventional modeling schemes. In absence of requisite modeling capability, the system partitioning simply can not be carried out without human interaction. Thus, there exists a strong relationship between the models used for capturing system functionality and the abstraction level at which the partitioning is carried out. The partitioning procedure presented in this section attempts to perform a division of functionality *at the level of operations specified in the hardware description language* and is not a substitute for partitioning usually carried out at higher levels of abstractions such as "conceptual" or architectural-level [35] or configuration-level [36] where the authors consider architectural partitioning for multiple processors. The problem of partitioning here refers specifically to a partition of the system functionality represented by a flow graph model. Since the flow models are developed to allow estimation of performance and constraint analysis for hardware and software, this analysis can be used in conjunction with partitioning transformations to ensure that partitioning objectives are met. This work considers an iterative improvement scheme for partitioning. Related work using clustering techniques can be found in [37], [38] and other approaches in [39], [40], [41]. A survey is provided in [5].

The partitioning problem for a flow graph refers to the assignment of operations in the graph to hardware or software. This assignment to hardware or software determines the delay of the operation. Further, the 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 the number of operations in software increases the degree of utilization of the processor. Consequently, the overall system performance is determined by the effect of hardware-software partition on performance parameters that go beyond the area and delay attributes described earlier. Therefore, the key to achieving an effective partition is to develop an appropriate *cost model* that captures relevant performance parameters from the attributes of the flow graphs. This cost model is described next.

6.1. Partition Cost Model

The partition cost model is built upon the processor cost model Π and the software model using multiple program threads shown in Figure 11. The software performance and size parameters were discussed in the previous section. The hardware component is characterized by its size. The hardware size, S_H is expressed in terms of number of cells needed to implement hardware using a specific library of gates. It provides a measure of the actual area of hardware implementation.

A partition of the system model Φ is refers to a partition of the flow graphs in Φ into two groups such that one is implemented into hardware and the other into software. Since a flow graph may be partitioned into hardware-software implementations, this separation is modeled by transformations on the flow graph that generate two separate interacting flow graphs. Since these transformations may, in general, require replication of vertices and edges in order to correctly model functionality, a partition of the *flow graph* is not exactly a partition of graphs in the mathematical sense, that is, there may be an overlap of vertices across partitions. This situation is not peculiar to hardware-software partitioning but a necessary side-effect of partitioning at a behavioral level of abstraction [42].

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

Problem P5: *Given a system model, Φ, as set of flow graphs, and timing constraints between operations, create a partition $\varpi(\Phi) = \Phi_S \cup \Phi_H$ in to two sets of flow graph model, Φ_H and Φ_S such that a hardware implementation of Φ_H and a software implementation of Φ_S implements Φ and the following is true:*

1. *Timing constraints are satisfied for all flow graphs in Φ_H and Φ_S,*
2. *Processor utilization, $\mathcal{P} \leq 1$,*
3. *Bus utilization, $\mathcal{B} \leq \overline{\mathcal{B}}$. The bound on bus utilization is a function of bus bandwidth and memory latency.*
4. *A partition cost function,*

$$f(\varpi) = a_1 \cdot S_H(\Phi_H) - a_2 \cdot S^{\Pi}(\Phi_S) + b \cdot \mathcal{B} - c \cdot \mathcal{P} + d \cdot |m| \qquad (22)$$

is minimized. Here $|m|$ defines the cumulative size of variables m that are transferred across the partition, and a_1, a_2, b, c and d are positive constants.

Parameters a_1, a_2, b, c and d represent desired tradeoff among size of hardware, software implementation, processor and bus utilization and the communication overheads. Let us first consider the size metrics: S_H and S^{Π}. The size of hardware is computed from the size attribute of a operation.

$$S_H(\Phi_H) = \sum_{G_i \in \Phi_H} S_H(G_i) = \sum_{G_i \in \Phi_H} \sum_{v \in V(G_i)} S(v) \qquad (23)$$

Since $S(v)$ is a local property of the vertex, it does not include hardware costs for control and scheduling logic circuits. As discussed in the previous section, we use the following (worst case) approximation for the program size as the number of *rvalues*, denoted by ω, associated with a vertex. That is,

$$S_p^{\Pi}(v) = O(\omega(v))$$

where $O()$ refers to the order-of approximation.

When partitioning a flow graph model into two models, to the first order we can assume that the data set $M(G)$ is replicated in both partitions to ensure correct functionality of the resulting partitioned graph models. Moving an operation across partition boundary changes set of input and output ports of G. Even though this change increases the software size, it does not help the goal of maximization of operations in the software. Instead, this change is accounted for as an adverse effect on the quality of the partition by increasing its communication overhead, m. Therefore, for the purposes of partition, we assume

$$S_d^{\Pi}(v) = 0 \Rightarrow S^{\Pi}(\Phi_S) = \sum_{G_i \in \Phi_S} \sum_{v \in V(G_i)} O(w(v)) \tag{24}$$

The next three parameters in the cost function, $\mathcal{B}$, $\mathcal{P}$ and m are related to the communication cost of the partition and the number of operations in the program threads. All of these parameters can, therefore, be calculated from operation dependencies and their attributes in the flow graph. Let us assign a weight, $c(u, v)$ to the edge (u, v) as the size of data transfer from vertex u to vertex v. In our implementation, a control transfer in the flow graph is simulated by means a data transfer on a port. The communication cost $|m|$ refers to the sum of edge weights over the edges that lie across the partition, ϖ.

The task of hardware-software partitioning requires evaluation and selection of operation vertices in the individual flow graphs. An exact solution to the constrained partitioning problem, that is, a solution that minimizes the partition cost function requires evaluation of a large number of operation groupings which is typically exponentially related to the number of operations in the system model. As a result, heuristics to find a 'good' solution are often used with the objective of finding an optimal value of the cost function. This optimality of the cost function is defined over grouping of operations achieved by some neighbourhood selection operations. Our heuristics to solving the partitioning problem starts with a constructive initial solution which is then improved by an iterative procedure by moving operations across the partition.

Before describing the heuristic, let us first take a look at the nature of the cost function that is used to direct the heuristic search. The cost function consists of a set of *properties*, for example, $S_H, S^{\Pi}, \mathcal{P}, \mathcal{B}, m$ in our case. Since a large number of group of operations are possible candidates for iterative improvement, the computation of the cost function, $f(\varpi)$ significantly affects the overall time complexity of the procedure. A cost function consisting of properties that require evaluation of the entire graph model at each step of the iteration would add to the complexity of the search procedure. On the other hand, an ideal case would be a cost function that can be *incrementally* updated from previous value, that is, the change in the cost function can be computed quickly. Such a quick estimation also allows for variable-depth neighbourhood search methods also such as Kernighan-Lin [43], where many hypothetical alternatives can be explored to select the "best" group of operations to move. Let us first examine the type of properties that constitute a cost function.

The Partitioning Cost Function: From a CAD point of view, the computation of the cost function is a challenging problem since it captures not only the effects of *sizes* of hardware and software parts but also the effect of *timing* behavior of these portions.

During the partitioning iteration, a movement of an operation across the partition causes a change in its delay and, therefore, the latency of the flow graph.

In general, it is hard to capture the effect of a partition on timing performance during partitioning stage. This is because the timing properties are *global* in nature, thus making it difficult to make incremental computation of the partition cost function. For each local move, the timing properties must be calculated for the entire graph model, which adds to the computational complexity of the heuristic search process.

Traditionally, the use of timing properties in driving the partitioning search process varies significantly, from almost none in hardware-based partitioning schemes, to use of statistical timing properties in program partitioning [44]. 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, therefore, there is not much need to consider the effect of partition on overall latency which is to a great extent dependent upon the choice of schedule for the operations.[5] In contrast, the program-level partitioning problem addresses operations that are scheduled at run-time. Because of this ability to schedule operations at run time, the timing properties are more complex and dependent upon the operating environment and external input to the software. Thus, a statistical measure is often used to capture the timing properties of a partition.

We take an intermediate approach to partitioning for hardware/software systems, 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 as a set of program threads as shown in Figure 11 and a partition cost function, f, that is a linear combination of its variables. Thus, the characterization of software using λ, ϱ, $\mathcal{P}$ and $\mathcal{B}$ parameters makes it possible to calculate static bounds on software performance. Use of these bounds is helpful in selecting appropriate partition of the system functionality into hardware and software. However, it also has the disadvantage of overestimating performance parameters. For example, the actual processor and bus utilization depends upon the distribution of data values and communication based on actual data values being transferred across the partition. Instead we determine the *feasibility* of a partition based on the worst case scenario and ensure that this worst case scenario is handled by the partition alternatives being evaluated by the partition cost function.

6.2. *Partitioning Feasibility*

A system partition into an application-specific and a re-programmable component is considered *feasible* when it implements the original specifications and it satisfies the performance constraints. We assume that the hardware and software compilation, done using standard tools, preserves the functionality. We, therefore, concentrate on timing constraints to drive the partitioning procedure.

As mentioned earlier, when partitioning system model into hardware and software components the data rates may not be uniform across models. The discrepancy in data rates is caused by the fact that the application-specific hardware and re-programmable

components may be operated with different clocks and the system execution model supports multi-rate executions that makes it possible to produce data at a rate faster than it can be consumed by the software component when using a using a finite sized buffer. In the presence of multi-rate data transfers, feasibility of hardware-software partition requires that for all data transfers across a partition, the production and consumption data rates are compatible with a finite and size-constrained interface buffer. That is, for any data transfer across a partition, data consumption rate is at least as high as the data production rate. The production and consumption rates for a data transfer are defined by the reaction rate of the corresponding flow graphs. The notion of feasibility of a partition between hardware and software builds upon the feasibility of each of the two components and additional constraints on processor and bus utilization. To be specific, a flow graph, $G_i \in \Phi_H$ is considered feasible if it meets the timing constraints under the assignment of hardware operation delays and no runtime scheduler. A flow graph, $G_i \in \Phi_S$ is considered feasible if it meets the imposed timing constraints under the assignment of software operation delays and software storage operations and there exists a feasible linearization of operations in G_i under the timing constraints. Timing constraint satisfiability is verified using the conditions described in Section 3. A partition of Φ in Φ_S and Φ_H is considered feasible if it is hardware feasible for Φ_H, software feasible for Φ_S, the bus bandwidth is bounded by $\overline{B}$ and for all program threads T_i in software $\frac{1}{\max \varrho(T_i)} \geq \sum_k \lambda_k$. The last condition also ensures that processor utilization is below unity.

Our procedure for partitioning a flow graph model is based on the iterative improvement procedure presented in Section 6.1. There are three main components to this procedure:

I. Creation of initial partition. The initial partition is constructed by creating program threads for each of the $\mathcal{ND}$ loop operations in the flow graph model. It is assumed that for all the external synchronization operations, a corresponding rate constraint is provided, which is used as a property of the environment with which the system interacts. Due to this property, rates of data transfer for all inputs to the software are known. This rate of data transfer then defines the reaction rate of the corresponding destination program thread. On the other hand, the rate of data production from the software is determined by achievable reaction rate of the associated program thread.

II. Selection of a group of operations to move across partition. Selection of operations requires a check for partitioning feasibility. Among the available vertices we pick operation vertices with known and bounded delay. With this vertex moved across the partition to software, its attributes are updated and the corresponding graph model is checked for constraint satisfiability. If the move is a feasible one, the next vertex to be selected is one of the immediate successors of the vertex moved. This way, the group of vertices selected for a move constitutes path in the flow graph. This heuristic selection is made to reduce the communication cost.

III. Update of the cost function. After the initial computation of the cost function, changes to the cost function are computed incrementally. A vertex move from hardware to software, entails the following *changes* to the partition cost function:

$$\Delta f = a_1 \Delta S_H - a_2 \Delta S^{\Pi} + b\Delta \mathcal{B} - c\Delta \mathcal{P} + d\Delta|m| \tag{25}$$

1. Hardware size S_H is reduced by the size attribute of the vertex according to Equation 23. So the reduction in cost function due to move of vertex, v into software is given by $\Delta f|_{S_H} = a_1 \cdot S(v)$.
2. Software size S^{Π} is increased by the *rvalue* attribute, ω of the vertex according to Equation 24. So the reduction in cost function is given by $\Delta f|_{S^{\Pi}} = a_2 \cdot \omega(v)$. Note that the effect of an operation move into software is to increase the size of the software which decreases the cost function.
3. The change in communication cost, Δm, is computed by examining the neighbours of vertex, v. This is also used to compute the change in bus utilization, $\Delta \mathcal{B}$.
4. Processor utilization, $\mathcal{P}$ is computed by considering two cases. One where the reaction rate of the destination thread is unaffected. In this case the reduction in cost function due to vertex move, v to thread k is given by

$$\Delta f|_{\mathcal{P}} = c \cdot \varrho_k \delta(v) \tag{26}$$

where $\delta(v)$ refers to the software delay of operation v. In case, the operation move changes the reaction rate of a thread to ϱ'_k, the effect on cost function reduction is given by

$$\Delta f|_{\mathcal{P}} = c[\varrho'_k \delta(v) + (\varrho'_k - \varrho_k)\lambda_k] \tag{27}$$

The algorithm to perform graph-based partitioning is described by the following pseudo-code. Starting with a system model, it examines flow graph models that in Φ corresponding to each process model for possible partition. Procedure *graph_partition* returns a *tagged* graph indicating its partition into two graphs. Vertices in a tagged graph are labeled according to their partition membership. This graph is then subject to partition transformations such that the resulting graphs correctly implement the specified functionality.

Input: System model, $\Phi = \{G_i\}$, Processor model, Π, Bus bandwidth $\overline{B}$, Runtime overhead, $\overline{\gamma}$
Output: Partitioned system graph model, $\Phi = \Phi_S \cup \Phi_H$

```
partition(Φ) {
    Φ_S = Φ_H = ∅;
    for each process graph in G_i ∈ Φ {
        graph_partition(G_i);                          /* returns a tagged graph, G_i */
        (G_H, G_S) = partition_transformation(G_i);    /* create separate graphs */
        Φ_S = Φ_S ∪ {G_S};
        Φ_H = Φ_H ∪ {G_H};
    }
}
graph_partition(G) {
    V_H = V;
    V_S = ∅;
▷I  for v ∈ V(G) {                                     /* create initial partition */
```

```
        if v is an NP link operation
            V_S = V_S + {v};
    }
    create software threads (V_S);                  /* create initial |V_S| routines */
    compute reaction rates, ϱ for each thread;      /* based on rate constraints */
    if not check_feasibility(V_H, V_S)              /* no feasible solution exists */
        exit ;
    f_min = f(V_H, V_S);                            /* initialize cost function */
    repeat {
        for vertex v ∈ V_H and v is not NP {        /* pick a det. delay operation from hw */
▷II         f_min = move(v);                        /* select operations to move to sw */
} until no further reduction in f_min;
    return(V_H, V_S);
    }
}
move(v) {                                           /* consider v for move from V_H to V_S */
    if check_feasibility(V_H − {v}, V_S + {v})
▷III    if Δf > 0
            V_H = V_H − {v};                        /* move this operation to sw */
            V_S = V_S + {v};
            f_min = f_min − Δf;
            update software threads;
            update thread reaction rate of the destination thread;
            for u ∈ succ(v) and u ∈ V_H             /* identify successor for move */
                move(u);
    return f_min;
}
check_feasibility(Φ_H, Φ_S) {                       /* check partition feasibility */
    for all G_i ∈ Φ_H                               /* check timing constraints for hardware */
        if not check_satisfiability(G_i);
            return not feasible;
    for all G_i ∈ Φ_S                               /* check timing constraint for software */
        if not check_satisfiability(G_i);
            return not feasible;
    for all T_i ∈ Φ_S
        if (1 / max ϱ(T_i) < Σ_k λ_k)               /* runtime scheduler */
            return not feasible;
    if B > B̄                                        /* check bus utilization */
        return not feasible;
    return feasible;
}
```

The markers ▷ indicate the main steps of the algorithms as described earlier in the beginning of this section. The algorithm uses a greedy approach to selection of vertices for move into Φ_S. There is no backtracking since a vertex moved into Φ_S stays in that set throughout rest of the algorithm. For each vertex move, the change in the partition cost function, Δf is computed in constant time. The constraint satisfiability algorithm *check_satisfiability* computes the all pair longest paths in the constraint graph which using Floyd's algorithm runs in $O(|V|^3)$ time. Since this satisfiability check is done for each possible vertex move, therefore, the complexity of the algorithm is $O(|V|^4)$.

7. System Implementation

It should be evident by now that hardware-software co-synthesis is not a single task but consists of a series of tasks that must be carried out interactively by the system designer. These tasks are related to modeling of functionality and constraints, analysis of constraints, model transformations to ensure constraint satisfiability, partitioning of the model and partitioning-related transformations, synthesis of hardware and software components, and simulation of the final system design. These subtasks have been implemented in a general framework, called VULCAN, that allows user interaction at each step of the co-synthesis process and guides the system designer to the goal of realizing a hardware-software system design. This section discusses the implementation of the Vulcan system and its relationship to other tools to accomplish synthesis and simulation of hardware-software systems.

Vulcan is written in the C programming language and consists of approximately 60,000 lines of code. Through its integration with the Olympus Synthesis System [45] and DLX processor compilation and simulation tools [46], it provides a complete path for synthesis of hardware and software from *HardwareC* descriptions.

The input to Vulcan consists of two components: a description of system *functionality* and a set of *design constraints*. The design constraints consist of timing constraints and constraints on parameters used during the co-synthesis process. Timing constraints are specified along with the system functionality in *HardwareC* using an *attribute* mechanism. These attributes make use of statement *tags* that identify the operation subject to constraints.

Example 7.1. A HardwareC description is annotated by the following attribute commands to specify minimum and maximum execution rate constraints, identify loop index variables, and specify clock names and cycle times.

```
.attribute ''constraint minrate [<num>] of <tag> = <num> cps''   Rate constraints
.attribute ''constraint maxrate [<num>] of <tag> = <num> cps''   Rate constraints
.attribute ''loop-index <str> [<num>]''                          Index variable
.attribute ''clock <str> [<num>]''                               Clock signal
.constraint mintime from <tag> to <tag> = <num> cycles           Min delay
.constraint maxtime from <tag> to <tag> = <num> cycles           Max delay
.constraint finish|finishedby|before|during <tag> <tag>
```

□

The input description is compiled into the bilogic flow graph model. The organization of data in Vulcan is shown in Figure 14. Vulcan maintains a list of hierarchically connected graph models. The flow graph models may be implemented or unimplemented. In addition, Vulcan maintains a list of processor cost models and cost models for implementation of software and hardware. The cost models for hardware and software store the results of actual hardware and software synthesis and are updated by the mapping results from hardware synthesis, and by parsing the dissembler output respectively. The algorithms for analysis and transformations are applied on these graph models. At this time, all transformations of the graph model are user driven. The model manipulation

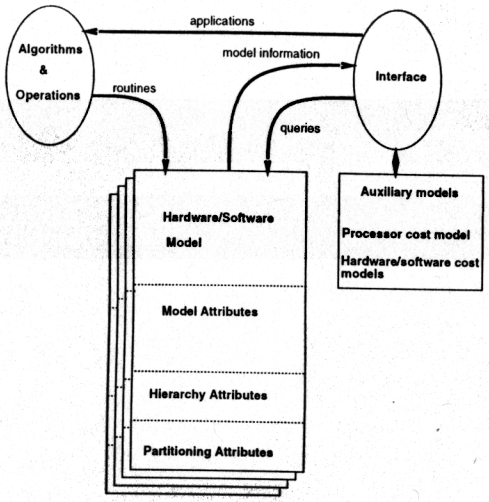


Figure 14. Data organization in Vulcan

routines automatically update the list of models after transformations and update the cost models and attributes with the result of analysis.

Figure 15 shows the organization of Vulcan subsystems and their relationship to hardware and software synthesis. Vulcan consists of following sub-systems.

1. **Model maintenance and manipulations** (command `Enter_model`) supports manipulations and constraint satisfiability analysis on the flow graph model. The maintenance functions include reading and writing of graph models, and cost model for the processors, identification of data and control dependencies and types of loop operations.

2. **Model partitioner** (command `Enter_partitioner`) Partitioning is accomplished by first 'tagging' a flow graph model, followed by creation of individual flow graph models through partitioning transformations. Assignment of graphs is made on the basis of partitioning feasibility analysis that checks for constraint satisfiability of hardware and software implementations of the individual flow graphs, and feasibility of the runtime scheduler to support the hardware and software portions based on processor and bus utilization.

3. **Hardware synthesis** (command `Enter_hardware`) is performed by passing the corresponding sequencing graph model to programs HEBE and CERES in the Olympus synthesis system.

4. **Software synthesis** (command `Enter_software`) consists of tasks of program thread identification, serializations, generation of routines. Figure 16 shows the command

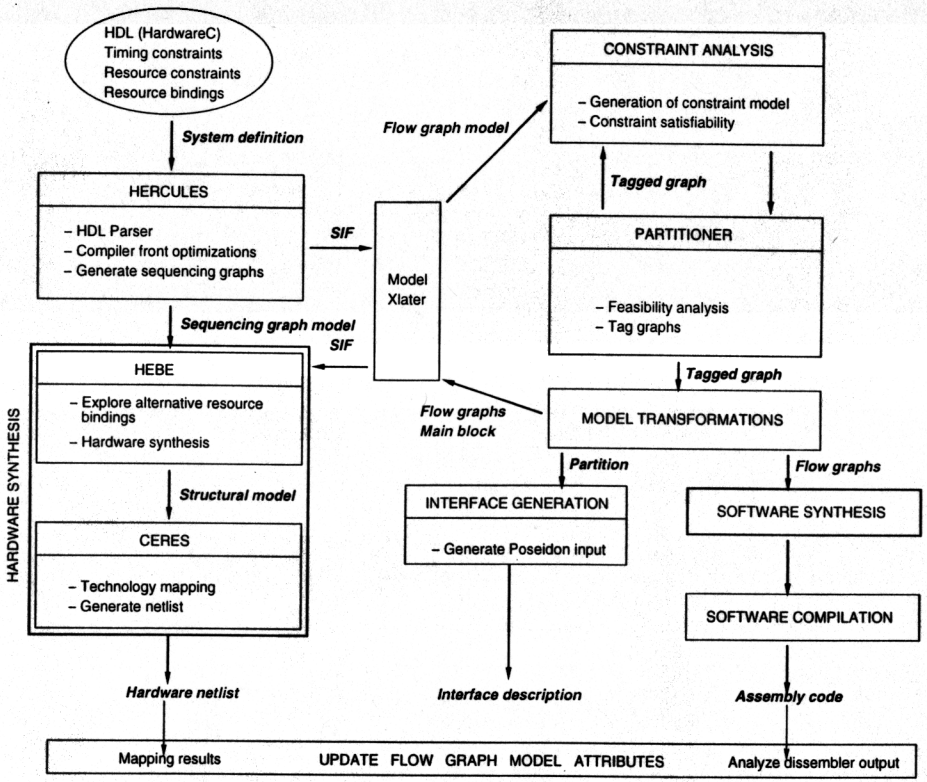


Figure 15. Vulcan subsystems and the Olympus Synthesis System

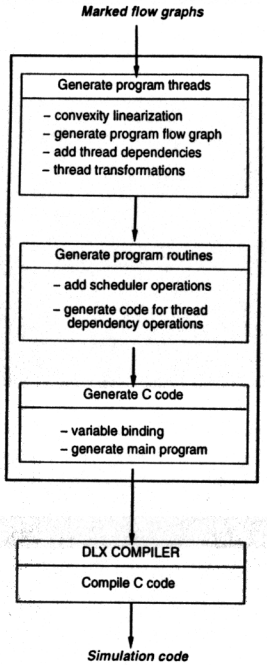


Figure 16. Flow of software synthesis in Vulcan

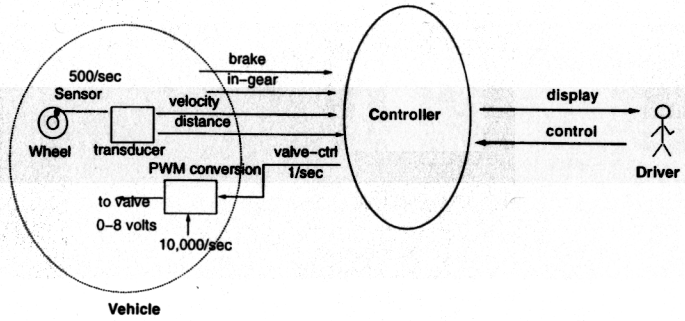


Figure 17. Controller specification diagram.

flow in generation of the software component. The software synthesis also performs generation of scheduling routines and hardware-software interface routines for the runtime system.

5. **Interface synthesis and model simulations** (command `Enter_interface`) Interface description is currently entered manually from specified models and interface protocols. Simulation of the mixed hardware-software system is performed by the program POSEIDON.

7.1. Examples

Vehicle cruise controller: We consider the design of a vehicle controller that monitors vehicle cruising speed, a record of average speed, monitors fuel consumption and provides vehicle status and maintenance feedback to the driver through a display controller. Figure 17 shows an overview of the controller.

The controller performs *velocity regulation* by making sure that the valve is completely closed (0 volts) when the vehicle speed is 5 mph over the desired speed. It is completely open (8 volts) when the vehicle speed is 5 mph under the desired speed. Between these two extremes the `valve-ctrl` value is proportional to the difference between the actual and desired speed. Acceleration control depends upon whether the driver is accelerating or not. If the current acceleration (computed by difference between consecutive speed values) is greater than 5 mph/sec then the valve is completely closed, if current acceleration is less than 1 mph/sec then the valve is completely open, else the valve is proportionally opened. The valve position should be changed no faster than 1 per second. Finally, the controller reaction time for driver requests must be less than 1 second, except for braking which must be as fast as possible (at most 1 ms).

This controller is described modularly as a collection of 7 blocks. The final is shown in Figure 18. A hardware-software implementation that considers implementation of the `STATE` block into software while the rest is implemented into hardware. The delay of

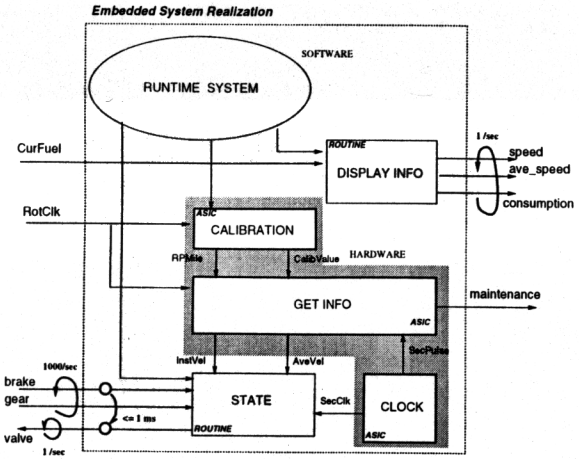


Figure 18. Controller implementation.

the program estimated from the flow graph of the STATE block is 1968 cycles (worst case, 362 cycles for best case) which corresponds to a software delay of 0.1 ms for a 20 MHz DLX processor. The actual delay of the compiled software is 750 clock cycles. The size of the software is 3.6 KB. Another implementation alternative includes the GETINFO block into software. The following table summarizes alternatives where the size of hardware is expressed in terms of cells using LSI logic 10K library of gates. All three implementations meet the timing constraints on the state update and valve control.

Table 1. Cruise controller implementation

Implementation	*Hardware size*	*Software size*
All hardware	5247 cells	-
Mixed implementation 1	3032 cells	3.6 KB
Mixed implementation 2	582 cells	7.5 KB

Graphics controller: The graphics controller is used for generating pixel coordinates for specified geometries such as lines and circles. The input to the controller is a specification of the geometry and its parameters, such as end points of a line. The controller is intended for use in a system where the graphics controller accepts input geometries at the rate of 2×10^5 per second and outputs at about 2 million pixel coordinates per second to a drawing buffer that is connected to the display device controller.

The mixed implementation is shown in Figure 19. Corresponding to the two $\mathcal{ND}$ operations in the input description, the software consists of line and circle drawing routines while the ASIC hardware performs initial coordinate generation and data transfer to the drawing buffer. The software component consists of two threads of execution corresponding to the line and circle drawing routines. The two program threads generate

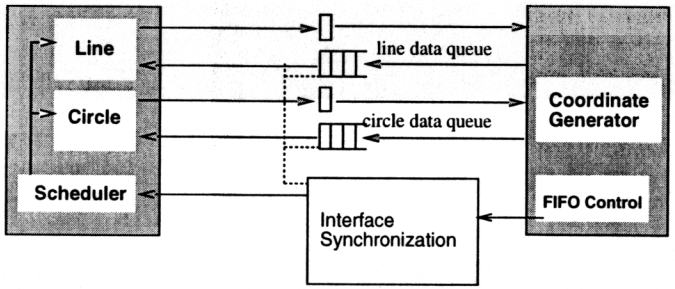


Figure 19. Graphics controller implementation.

coordinates that are used by the dedicated hardware. In addition, the runtime scheduler is implemented as a separate program thread that interacts with the interface synchronization block.

Hardware/Software interface for the controller: The hardware-software interface consists of data queues on each channel and a FIFO that holds the identifiers for the enabled program threads in the order in which their input data arrives. The control FIFO depth is sized with the number of threads of execution, since a program thread is stalled pending availability of the requested data. Thus the maximum number of places in the control FIFO buffer is three. The following describes the specification of the interface between hardware and software:

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

dlx.0xff000[2:2] = !controlFIFO.empty;
```

Here two data queues with 16 bits of width and unit depth, line_queue and circle_queue, and one queue with 2 bits of width and unit 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 condition states that when a request for a dequeue on the queue line_queue comes and the queue is not empty and the queue controlFIFO is not full, then enqueue the value 1 in the controlFIFO. The last command just specifies a direct connection between signal not controlFIFO.empty and bit 2 of signal dlx.0xff000.

Table 2. System performance under different software implementation schemes.

Scheme	*Software size* (bytes)	*Runtime overhead*	*Input data rate*$^{-1}$ (cycles/coordinate)	*Output data rate*$^{-1}$ (cycles/coordinate)			
				line		circle	
				ave.	peak	ave.	peak
Hardware interface	5972	19	81	535.2	502	76.4	30
Software interface	6360	19+84=103	95	651	330	94	31

The control FIFO and associated control logic can be implemented either in hardware as a part of the ASIC component or in software. In the case that 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 the data queues are connected to the 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.

Table 2 compares the performance of different program implementations using hardware and software interface implementations. The hardware interface implementation uses a FIFO queue with associated control logic that is synthesized into 228 gates using LSI 10K library of gates. A software implementation adds 388 bytes to the overall program size of the software component and increases the runtime overhead by 84 cycles. Note that the data input and output rates have been expressed in terms of number of cycles it takes to input or output a coordinate. Due to the data-dependent behavior of program threads, the actual data input and output rates would vary with respect to value of the input data. In this example simulation, the input rate has been expressed for a simultaneous drawing of a line and 5 pixel radius with width of 1 pixel each and the results are accurate to one pixel. An input rate of 81 cycles/coordinate corresponds to approximately 0.25 million samples/sec for a processor running at 20 MHz. Similarly, a peak circle output rate of 30 cycles/coordinate corresponds to a rate of 0.67 million samples/sec. Figure 20 shows implementation of the *main* program that provides runtime scheduling of threads for the two cases of interface implementations.

Table 3. Graphics controller implementations.

Implementation	*Size*	*Performance*
Hardware implementation	10,642 gates	14.70
Mixed implementation	228 gates, 5972 bytes	0.25
Software implementation	6360 bytes	0.21

Table 3 lists alternative implementations of the controller. The hardware size is reported as number of cells using LSI 10K standard cell library. The software results are reported for a DLX processor running at 20 MHz. The performance column lists the input data rate expressed in million samples per second. Performance of a pure software implementation of the controller depends strongly upon the choice of the runtime system. A conventional subroutine based scheduler would add substantial overheads due to storage management operations. On the other hand, a software control FIFO implementation can be treated


```
#include "transfer_to.h"

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

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

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

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

int nextCoroutine;

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

POLLED PROGRAM THREADS

```
#include "transfer_to.h"

int *int1_ak = (int *) 0xb00000;
int *int2_ak = (int *) 0xc00000;

int controlFIFO[16];
int queuein=0, queueout=0, empty=1, full=0;

enqueue(id)
      int id;
{
      queuein = (queuein + 1) & 0xf;
      controlFIFO[queuein] = id;

      empty = 0;
      full = (queuein == queueout);
}

dequeue()
{
      queout = (queout + 1) & 0xf;

      full = 0;
      empty = (queuein == queueout);
      return controlFIFO[queueout];
}
```

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

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

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

int nextCoroutine;

void scheduler() {
        resume (LINE);
        resume (CIRCLE);
        while (1) {
            while (empty);
              transfer_to (dequeue());
        }
}
```

INTERRUPT–DRIVEN PROGRAM THREADS

Figure 20. Graphics controller software component.

as a kind of pure software implementation (using interrupts) which gives an input rate of 0.21 million samples/sec.

8. Summary and Future Work

We have presented a co-synthesis approach to achieve computer-aided design of embedded systems. This methodology is driven by the analysis of constraints on the timing performance of the final system implementation. The overall system cost is minimized by reducing the use of application-specific hardware component. From the input description using a hardware-description language (HDL), we develop a graph based model that is applicable to synthesis of both hardware and software due to its explicit treatment of operation-level concurrency and synchronization. Based on this graph-based model, the problem of co-synthesis is divided into sub-problems of performance modeling and estimation for hardware and software, the identification of hardware and software and finally the synthesis and integration of hardware and software components.

We have developed the notion of constraint satisfiability both in the deterministic and probabilistic sense as an existence condition for a feasible schedule of operations. We describe the operation-level timing constraints commonly used in embedded systems and methods for determination of constraint satisfiability using graph analysis techniques. We model software as a set of multiple concurrent program threads instead of a single program. This implementation of software avoids the need for complete serialization of all operations, which helps in determination of constraint satisfiability in the presence of timing uncertainty in executions. Synthesis of hardware is carried out by use of high-level synthesis techniques. Synthesis of software poses challenging issues due to

the need for serialization of all operations and development of a low overhead runtime system. We use a FIFO-based runtime scheduler to implement the software as a set of multiple concurrent coroutines. Finally, the hardware-software system is put together by design of a low overhead hardware-software interface.

It is clear that hardware-software co-synthesis is primarily a system building activity that spans several disciplines from CAD-theoretic aspects of algorithms for constraint analysis and partitioning to system implementation issues of concurrency and run-time systems to support multi-programming. This work formulates its decomposition into the various sub-problems and proposes solutions for a practical co-synthesis approach. Where possible, uniqueness of the sub-problems as applied to co-synthesis are pointed out. In the process, several simplifications are made in an attempt to keep the focus on essentials of the co-synthesis problem while delegating peripheral (though sometimes no less important) problems to a workable engineering solution. In order to make co-synthesis general and useful for different application domains, extensions to the target architecture and alternative target architectures must be explored. Software generation in current work is limited to generation of high-level language code from the graph models, whereas the actual task of compilation is delegated to existing compilers. We are currently examining techniques for machine-level code generation from flow graph models under timing constraints.

Acknowledgments

We acknowledge discussions and contributions by Claudionor Coelho and David Ku. This research was sponsored by a grant from the AT&T Foundation and from NSF-ARPA Nos. MIP 91-15432 and 95-01615.

Notes

1. Synchronization is a general concept. Sometimes synchronization is needed to manage availability of shared resources. In our HDL specifications, synchronization is explicitly indicated only in the context of communication operations. A static resource allocation and binding paradigm is assumed, thus obviating the need for resource synchronization, i.e., avoiding conflicts when the same resource implements more than one operation. Therefore, synchronization in this work is mentioned in the context of communication operations.
2. Recall that loop vertices represent 'repeat-until' type operations. The length computation treats the loop operation as a call operation.
3. The mobility is the difference in the starting times of ALAP and ASAP schedules of a deterministic delay flow graph constructed by considering all link vertices to be call link vertices with delay as the maximum path length of the called graphs.
4. Note that in hardware-software implementations the buffer between producer and consumer can also be implemented as a serial-parallel *rewording* operation, where the data to be transferred from producer to consumer is reworded as a multiple of original data width. The producer then assembles new words which consists of multiple invocations of the producer.
5. Partitioning for unscheduled flow graphs was considered in [47] that considers a cost function using latency and size properties. The update of the latency in the inner loop of search for the best group of operations was achieved by an approximation technique that allowed for incremental update of the latency, even though it is a global property.

References

1. K. P. Juliussen and E. Juliussen, *The 6th Annual Computer Industry Almanac 1993*. The Reference Press, Austin, TX, 1993.
2. M. L. Bader, "Market survey," *Bader Associates, Mountain View, California*, 1993.
3. J. Aylor, R. Camposano, M. Schuette, W. Wolf, and N. Woo, "The future of embedded system design," in *Proc. International Conference on Computer Design*, pp. 144–146, 1992.
4. W. Wolf, "Hardware-Software Co-design of Embedded Systems," *IEEE Proceedings*, vol. 82, no. 7, pp. 965–989, July 1994.
5. D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems*. Prentice Hall, 1994.
6. M. Chiodo, P. Giusto, , A. Jurecska, A. Jurecska, A. S. Vincentelli, and L. Lavagno, "Hardware-software codesign of embedded systems," *IEEE Micro*, vol. 14, no. 4, pp. 26–36, Aug. 1994.
7. D. E. Thomas, J. K. Adams, and H. Schmit, "A Model and Methodology for Hardware-Software Codesign," *IEEE Design & Test of Computers*, pp. 6–15, Sept. 1993.
8. A. Kalavade and E. A. Lee, "A Hardware-Software Codesign Methodology for DSP Applications," *IEEE Design and Test Magazine*, pp. 16–28, Sept. 1993.
9. D. Harel, "Biting the silver bullet," *IEEE Computer*, pp. 8–20, Jan. 1992.
10. N. Halbwachs, *Synchronous programming of reactive systems*. Kluwer Academic, 1993.
11. A. Burns and A. Wellings, *Real-Time Systems and Their Programming Languages*. Addison-Wesley, 1990.
12. D. Ku and G. D. Micheli, *High-level Synthesis of ASICs under Timing and and Synchronization Constraints*. Kluwer Academic Publishers, 1992.
13. V. Cerf, *Multiprocessors, Semaphores and a Graph Model of Computation*. PhD thesis, UCLA, Apr. 1972.
14. D. Bustard, J. Elder, and J. Welsh, *Concurrent Program Structures*, Prentice Hall, 1988.
15. 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, pp. 80–86, Jan. 1985.
16. J. F. Allen, "Maintaining knowledge about temporal intervals," *Communications ACM*, vol. 26, no. 11, pp. 832–843, Nov. 1983.
17. 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.
18. R. K. Gupta, *Co-Synthesis of Hardware and Software for Digital Embedded Systems*. Kluwer Academic Publishers, Boston, 1995.
19. T. Amon and G. Borriello, "Sizing Synchronization Queues: A Case Study in Higher Level Synthesis," in *Proc. 28th Design Automation Conference*, pp. 690–693, June 1991.
20. T. Kolks, B. Lin, and H. D. Man, "Sizing and Verification of Communication Buffers for Communicating Processes," in *Proc. IEEE International Conference on Computer-Aided Design*, pp. 660–664, Nov. 1993.
21. A. Mathur and R. K. Gupta, "Rate analysis for embedded systems," technical report, Computer Science, University of Illinois, 1995.
22. D. R. Cox and W. L. Smith, *Queues*. John Wiley and Sons, 1961.
23. A. Heck, *Introduction to Maple*. Springer-Verlag, 1993.
24. A. Mok, *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment*. PhD thesis, M.I.T., 1983.
25. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.
26. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.
27. A. Shaw, "Reasoning about Time in Higher Level Language Software," *IEEE Trans. Software Engg.*, vol. 15, no. 7, pp. 875–889, July 1989.
28. A. Mok and et. al., "Evaluating Tight Execution Time Bounds of Programs by Annotations," in *Proc. IEEE Workshop Real-Time Operating Systems and Software*, pp. 74–80, May 1989.
29. C. Y. Park and A. C. Shaw, "Experiments with a Program Timing Tool Based on Source-Level Timing Schema," in *Proc. 11th IEEE Real-Time Systems Symposium*, pp. 72–81, Dec. 1990.

30. W. Hardt and R. Camposano, "Trade-offfs in HW/SW Codesign," in *International Workshop on Hardware-Software Co-design*, Oct. 1993.
31. C. Y. Park, "Predicting program execution times by analyzing static and dynamic program paths," *Real-Time Systems*, vol. 5, no. 1, pp. 31–62, Mar. 1993.
32. W. Ye, R. Ernst, T. Benner, and J. Henkel, "Fast timing analysis for hardware-software co-synthesis," in *Proc. International Conference on Computer Design*, 1993.
33. P. Chou and G. Borriello, "Software scheduling in the Co-Synthesis of Reactive Real-Time Systems," in *Proc. Design Automation Conference*, June 1994.
34. R. K. Gupta and G. D. Micheli, "Constrained software generation for hardware-software systems," in *Proc. Third Intl. Workshop on Hardware-Software Co-design*, Sept. 1994.
35. S. Kumar, J. H. Aylor, B. W. Johnson, and W. A. Wulf, "A Framework for Hardware/Software Codesign," *Computer*, vol. 25, no. 12, pp. 39–45, Dec. 1993.
36. J. G. D'Ambrosio and X. Hu, "Configuration-level hardware/software partitioning for real-time embedded systems," in *Proc. Intl. Workshop on Hardware/Software Codesign*, 1994.
37. F. Vahid and D. D. Gajski, "Specification partitioning for system design," in *Proc. Design Automation Conference*, 1992.
38. E. Barros, W. Rosenstiel, and X. Xiong, "Hardware/Software Partitioning with UNITY," in *Workshop on Hardware/Software Co-design*, Oct. 1993.
39. G. Menez, M. Auguin, F. Boeri, and C. Carriere, "A Partitioning Algorithm for System-Level Synthesis," in *Proc. IEEE International Conference on Computer-Aided Design*, pp. 482–487, Nov. 1992.
40. R. Ernst, J. Henkel, and T. Benner, "Hardware-Software Cosynthesis for Microcontrollers," *IEEE Design & Test of Computers*, pp. 64–75, Dec. 1993.
41. T. B. Ismail, K. O'Brien, and A. Jerraya, "Interactive system-level partitioning with PARTIF," in *Proc. European Design Automation Conference*, 1994.
42. R. K. Gupta and G. D. Micheli, "Vulcan - A System for High-Level Partitioning of Synchronous Digital Systems," CSL Technical Report CSL-TR-471, Stanford University, Apr. 1991.
43. B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Technical Journal*, vol. 49, pp. 291–307, Feb. 1970.
44. V. Sarkar, *Partitioning and scheduling parallel programs for multiprocessors*. MIT Press, Cambridge, Mass., 1989.
45. 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.
46. J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan-Kaufman, 1990.
47. R. Gupta and G. D. Micheli, "Partitioning of Functional Models of Synchronous Digital Systems," in *Proc. IEEE International Conference on Computer-Aided Design*, pp. 216–219, Nov. 1990.