A Co-Synthesis Approach to Embedded System Design Automation

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

Giovanni De Micheli
Department of Electrical Engineering
Stanford University
Stanford, CA 94305.

November 2, 1994

Abstract

Embedded systems are targeted for specific applications under constraints on relative timing of their actions. For such systems, use of *predesigned 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. Dedicated hardware is often necessary to achieve requisite timing performance. Analysis of timing constraints is key to determination of an efficient hardware-software implementation. In this paper, we present a methodology to achieve embedded system realizations as *co-synthesis* of interacting hardware and software components. This co-synthesis is based on synthesis techniques for digital hardware and software compilation under constraints. We present operation-level timing constraints and develop the notion of satisfiability of constraints by a given implementation. Constraint 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 demonstrate the utility of our approach.

1 Introduction

Application-specific systems are designed for dedicated applications. Examples of such systems can be found in medical instrumentation, process control, automated vehicles 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 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 a 18% annual growth rate which is about twice the rate of growth for general-purpose systems [1].

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 component 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, despite the fact that such processors have been commonplace since 1985 and almost all advances in processor technology since then have been concentrated in the design of 32-bit processors.

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 final choice of implementation. Current practice in embedded systems is primarily a manual design process, characterized by a long design time and high cost of design. In recent

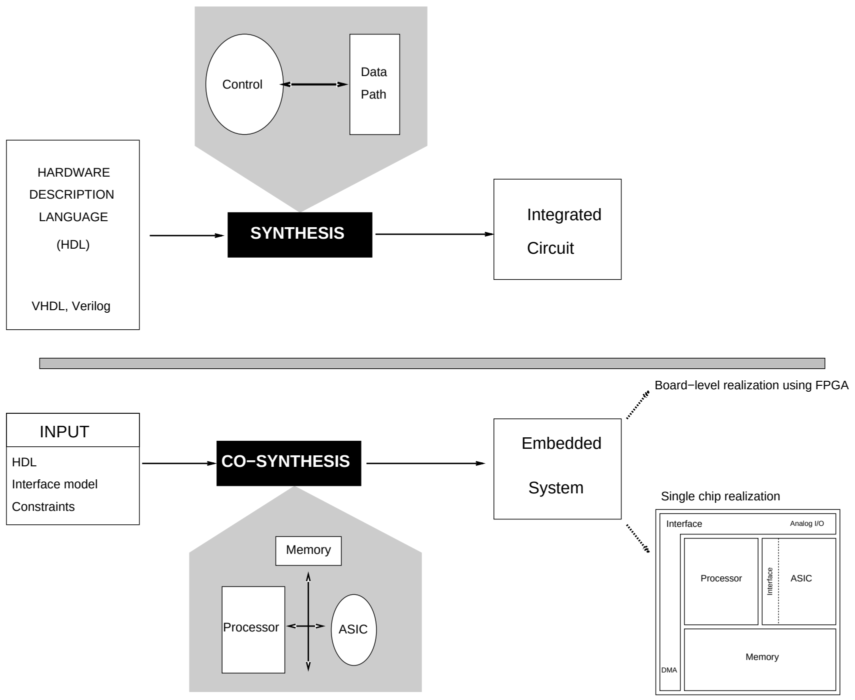


Figure 1: CAD for integrated circuits and embedded systems.

years, a growing need has been felt for the use of computer-aided design (CAD) tools to help rapidly explore the implementation alternatives and arrive at a cost-optimal implementation that meets constraints on performance [3] [4].

This work develops design automation of embedded system by extending techniques for synthesis of digital hardware and software compilation to formulate a *co-synthesis* 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 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 [5, 6, 7, 8]. 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.

We make the following assumptions relating to the target architecture to keep the important synthesis tasks subject to a systematic approach, while at the same time retaining the generality and effectiveness of the target architecture. Many of these assumptions can be dropped in a larger system co-design methodology without affecting the underlying co-synthesis approach presented here.

- We restrict ourselves to use of a *single re-programmable component.* The presence of multiple re-programmable components requires additional software synchronization and memory protection considerations to facilitate safe multiprocessing. Multiprocessor implementations also increase the system cost due to requirements for additional system bus bandwidth to facilitate inter-processor communications. The multiprocessing issues, though important, are orthogonal to the system co-synthesis problem addressed in this work.

- The memory used for program and data-storage may be on-board the processor. However, the interface buffer memory needs to be accessible to all of the hardware modules directly. Because of the complexities associated with modeling hierarchical memory design, we consider only the case where

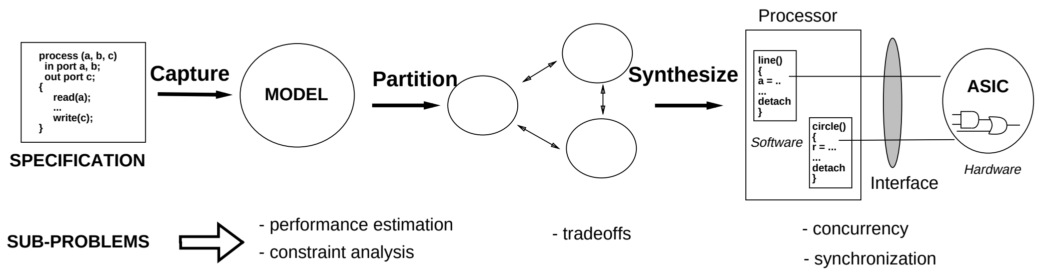


Figure 2: Synthesis approach to embedded systems.

all memory accesses are to a single level memory, i.e., outside the re-programmable component. 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.

- 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.
- All the communication between the re-programmable component and the application-specific circuits is done over named channels whose width (i.e. number of bits) is the same as the corresponding port widths used by read and write instructions in the software component. The physical communication takes place over a shared bus.
- The re-programmable component contains a 'sufficient' number of maskable interrupt input signals. For the purpose of simplicity, we assume that these interrupts are unvectored and there exists a predefined destination address associated with each interrupt signal.
- The application-specific components have a well-defined $RESET$ state that is achieved through a system initialization sequence.

A single-chip realization of this target architecture uses a processor core and ASIC hardware circuits using standard cells or gate array technologies. Figure 2 shows the essential aspects of the proposed co-synthesis approach. A 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 Section 3. We present our choice of the software and runtime system in Section 4. Section 5 partitioning procedure for hardware and software based on a notion of non-determinism in the input system model. Section 6 describes the implementation for the VULCAN co-synthesis system and example implementations. We summarize main contributions and directions for future research in Section 7.

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 irrespective of actual system implementation:

- 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.

The input to our co-synthesis system consists of a description of system functionality in a hardware description language (HDL), *HardwareC* [9]. Our choice of an HDL for input specification is due to several reasons. One, the use of an HDL makes it possible to use existing synthesis techniques for digital hardware in system implementation. Two, most HDLs allow for computation of explicit dependencies between operations and memory usage by use of static data types and unaliased data references. As we shall in later sections that both of these features are essential for analysis of constraints on timing and size of implementation. The particular choice of *HardwareC* is immaterial and other procedural 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. Example 2.1 describes a simple process specification.

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 ;
{
  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. □

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. We describe next how synchronization and constraints are specified.

Memory and communication. A communication between two operations is accomplished by means of a direct connection or over a shared medium such as memory. A choice of a particular communication mechanism depends upon the individual operations and their implementation as hardware or software. Communication between operations belonging hardware and software is generalized to occur over *ports*. Ports represent communication to a shared-memory (SM) or inter-process communication by means of message-passing (MP) operations. In the case of message passing communication between two operations, the actual data transfer is preceded by a *handshake protocol* that requires the sending and receiving operations to execute simultaneously (Figure 3). 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 the 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 that facilitate a blocking read operation based on the availability of data on channel `b`.

[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.

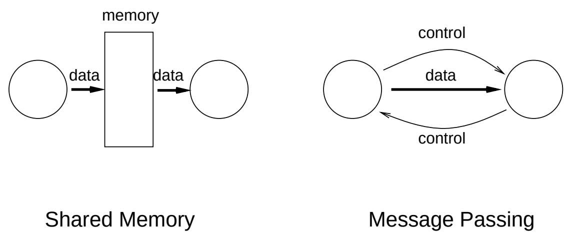


Figure 3: Shared memory versus message passing communication

2.1 System Representation

A system representation is needed to efficiently carry out the performance estimation tasks for hardware and software. 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.* In the following we present a graph-based model that represents operation-level concurrency explicitly while making a provision for encapsulating operations due to the runtime system by a suitable implementation of the source and sink operations in the *polar* graph-model. 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 2.1 *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,* $E = \{(v_i,\ v_j) \text{ or } v_i > v_j\}$ *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.*

Table 1 lists operation vertices used in a flow graph model. A *wait* operation is used to represent synchronization events at model ports. A *link* operation is used to represent hierarchy of models by means of call or loop operations. 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.

Note that the presence of multiple case values for the same branch leads to multiple edges between the condition and its successor vertex. Thus the flow graph is in general a multigraph. The flow graph model is similar to sequencing graph model in [9] with the following differences:

- A *wait* operation is added to abstract operations that represent synchronization events at model ports. This distinguishes a synchronization operation such as "`wait(signal)`" from a loop operation such as "`while(!signal)`". The reason for this distinction is that a software implementation of a wait operation is different from that of a loop operation. Whereas, due to the presence of multiple threads of execution in hardware, the wait operation is synthesized as a busy-waiting loop operation.

- Conditional `cond` and `join` operations. These operations have been added for the purposes of simplicity in data structures and constitute simple syntactic alteration to the sequencing graph model.

The advantage of the above changes to the sequencing graph models of [9] is that they permit the distinction in abstraction of intra-model and inter-model communications as based on shared memory or message passing respectively.

Operation	*Description*
no-op	No operation
cond	Conditional fork
join	Conditional join
op-logic	Logical operations
op-arithmetic	Arithmetic operations
op-relational	Relational operations
op-io	I/O operations
wait	Wait on a signal variable
link	Hierarchical operations

Table 1: Operation vertices in a flow graph

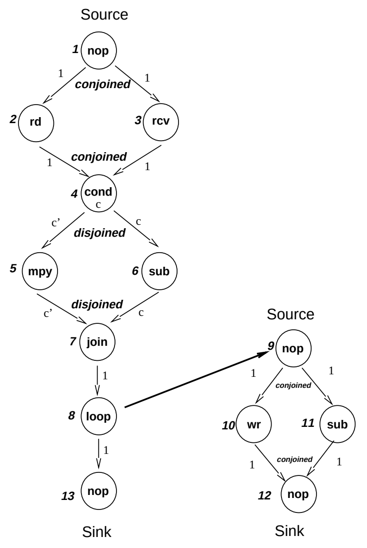


Figure 4: Flow graph of process `example`.

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 fanin 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 fanout 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. A conjoined output directs the flow of control to all its branches, whereas a disjoined output selects one of the successors based on condition index. Similarly, a conjoined input requires arrival of control on all its inputs before enabling the vertex. Structurally this makes the flow graph a *bilogic* graph [10]. For this reason, the flow graphs can be called *bilogic sequencing graphs* as opposed to (unilogic) *sequencing graphs* introduced in [9]. Bilogic graphs are a fairly common occurrence in control graphs.

Example 2.2. Figure 4 shows example of a well-formed bilogic graph model for the process described in Example 2.1. The example shows a one-bit condition variable, `c = (x > y)`. In general, it can 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. A flow graph model that is common to two hierarchies of a system model is called a *shared model.*

Execution semantics. The operational semantics associated with the execution of operations in a flow graph is as follows. At any time, an operation may be waiting for its execution, presently executing or having completed its execution. Correspondingly, we define the *state*, ς, of a vertex to be one of $\{s_r,\ s_e,\ s_d\}$ where s_r refers to the reset state, s_e to the enable state and s_d to the done state. An operation is enabled for execution once all its predecessors have completed execution in the case of a input-conjoined vertex; and once any of its predecessors has completed execution in the case of a input-disjoined vertex. The state of a vertex is changed from *done* to *reset* if all its successors are either reset or done. This semantics is general, and can support both pipelined and non-pipelined implementations of the graph model.

Example 2.3. Execution of process graph model `example` in Figure 4.

Non-pipelined

1	2	3	4	5	6	7	8	9	10	11	12	13
-	-	-	-	-	-	-	-	-	-	-	-	-
e	-	-	-	-	-	-	-	-	-	-	-	-
d	e	e	-	-	-	-	-	-	-	-	-	-
-	d	d	e	-	-	-	-	-	-	-	-	-
-	-	-	d	e	-	-	-	-	-	-	-	-
-	-	-	-	d	-	e	-	-	-	-	-	-
-	-	-	-	-	-	d	e	-	-	-	-	-
-	-	-	-	-	-	-	e	e	-	-	-	-
-	-	-	-	-	-	-	e	d	e	e	-	-
-	-	-	-	-	-	-	e	-	d	d	e	-
-	-	-	-	-	-	-	d	-	-	-	d	e
e	-	-	-	-	-	-	-	-	-	-	-	d
d	e	e	-	-	-	-	-	-	-	-	-	-
-	d	d	e	-	-	-	-	-	-	-	-	-
-	-	-	d	-	e	-	-	-	-	-	-	-
-	-	-	-	-	d	e	-	-	-	-	-	-
-	-	-	-	-	-	d	e	-	-	-	-	-
-	-	-	-	-	-	-	e	e	-	-	-	-
-	-	-	-	-	-	-	e	d	e	e	-	-

Pipelined

1	2	3	4	5	6	7	8	9	10	11	12	13
-	-	-	-	-	-	-	-	-	-	-	-	-
e	-	-	-	-	-	-	-	-	-	-	-	-
d	e	e	-	-	-	-	-	-	-	-	-	-
-	d	d	e	-	-	-	-	-	-	-	-	-
-	-	-	d	e	-	-	-	-	-	-	-	-
-	-	-	-	d	-	e	-	-	-	-	-	-
-	-	-	-	-	-	d	e	-	-	-	-	-
e	-	-	-	-	-	-	e	e	-	-	-	-
d	e	e	-	-	-	-	e	d	e	e	-	-
-	d	d	e	-	-	-	e	-	d	d	e	-
-	-	-	d	e	-	-	d	-	-	-	d	e
-	-	-	-	d	-	e	-	-	-	-	-	d
-	-	-	-	-	-	d	e	-	-	-	-	-
e	-	-	-	-	-	-	e	e	-	-	-	-
d	e	e	-	-	-	-	e	d	e	e	-	-
-	d	d	e	-	-	-	e	-	d	d	e	-
-	-	-	d	-	e	-	d	-	-	-	d	e
-	-	-	-	-	d	e	-	-	-	-	-	d
-	-	-	-	-	-	d	e	-	-	-	-	-

This table shows a particular sequence of operation executions for a given data input. Symbols 'e' and 'd' indicate enable and done states respectively. A dash '-' indicates the reset state. Note that the source vertex can be enabled for execution in the same step as the completion of the same vertex without changing the execution semantics. No assumption about timing of the operations is made, that is, consecutive rows in the table above can be spaced arbitrarily over the time axis. Thus, the execution of a flow graph progresses as a *wavefront* of operations are enabled for execution. The operations may complete at different times depending upon the delay of the individual operations.

The table on the left shows the *non-pipelined* execution of the graph model, that is, the source vertex is enabled again only after the completion of all operations in the graph model. On the contrary, an execution is considered *pipelined* if the source operation is enabled before completion of all operations. Therefore, in a pipelined implementation, there is more than one wavefront of enabled operations that progresses through the graph model at any time. In general, pipelining of flow graphs requires generation of pipeline stall and bypass control needed to accommodate pipelining of variable delay and synchronization operations. In this work, we consider restricted pipelining using buffers only in the context of software synthesis. For this pipelined execution, the minimum number of steps before the source operation can be enabled is determined by the maximum number of the steps taken by any operation. □

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. This actual assignment of values is related to the hardware or software implementation of operations in G. For non-pipelined hardware implementations, the runtime-scheduler is trivial, the source operation is enabled once its sink operation completes (and the graph enabling condition is true for conditionally invoked graphs). 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 4.

Size attributes refer to the physical *size* and pinout of implementations of operations and graphs. A hardware implementation consists of hardware resources (also called data-path resources), control logic, registers, and communication structures likes 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.

In general, it is a difficult problem to accurately estimate the size of the hardware required from flow graph models. Indeed, the size of implementation is one of the metrics that hardware synthesis attempts to minimize. Estimation in this context really refers to *relative* sizes for implementations of different flow graphs, rather than an absolute prediction of the size of the resulting hardware as formulated in [11, 12].

The effect of resource usage constraints is to limit the amount of available concurrency in the flow graph model. The more constraints on available hardware resources, the more operation dependencies are needed to ensure constraint satisfaction. The effect of timing constraints, on the other hand, is to explore alternative implementations *at a given level of concurrency.* We assume that the expressed concurrency in flow graph models can be supported by available hardware resources. That is, serialization required to meet hardware resource constraints has already been performed. This is not a strong assumption, since the availability of major resources like adders and multipliers are usually known in advance.

Capturing memory side-effects of a software implementation. 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* in order to facilitate multiple-assignments in HDL descriptions. Whereas additional variables can be created that avoid multiple assignments to the same variable, assignments to ports must still be multiply assigned 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). Further, M, need not be the *minimum* storage required for correct behavioral interpretation of a flow graph model.

Timing properties. 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. 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 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 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.

[2]Recall that loop vertices represent 'repeat-until' type operations. The length computation treats the loop operation as a call operation.

The elements of $\underline{\ell}$ are the lengths of the longest paths that are mutually-exclusive. No particular ordering of elements in $\underline{\ell}$ is assumed.

The length computation for a flow graph proceeds by a bottom-up computation 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 $\delta_u \odot \delta_v \overset{\triangle}{=} \delta_u + \delta_v$; (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 $\delta_u \otimes \delta_v \overset{\triangle}{=} \max(\delta_u,\ \delta_v)$; (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 $\delta_u \oplus \delta_v \overset{\triangle}{=} (\delta_u,\ \delta_v)$. Clearly, a disjoined composition of two delays leads to a 2-tuple delay since the two operations belong to mutually exclusive paths. With this definition, the composition operators, $\odot$, $\otimes$ and $\oplus$ form a simple algebraic structure called commutative monoid, on the the power set of positive integers, Z^+ with 0 as an identity element. This composition of delays is generalized to composition of paths as follows. In case of a sequential composition of two path lengths, $\underline{\ell}_u$ and $\underline{\ell}_v$ with cardinality n and m respectively, the resulting path length contains $n \times m$ elements, consisting of sum over all possible pairs of elements of $\underline{\ell}_u$ and $\underline{\ell}_v$. In case of a conjoined composition, the resulting path length is of cardinality $n \times m$ and consists of maximum over all possible pairs of elements. Finally, in case of a disjoined composition, the resulting path length is of cardinality $n + m$ and contains all elements of $\underline{\ell}_u$ and $\underline{\ell}_v$.

Rate of execution. The *instantaneous rate of execution*, $\widetilde{\rho}_i(t)$ of an operation v_i is the marginal number of executions n of operation v_i at any instant of time, t.

$$\widetilde{\rho}_i(t) \overset{\triangle}{=} \frac{dn}{dt} = \lim_{\Delta t \to 0} \frac{\Delta n}{\Delta t} \qquad (\text{sec}^{-1}) \tag{1}$$

Due to the discrete nature of executions (i.e., $n \in Z^+$), we define

$$\widetilde{\rho}_i(t) \overset{\triangle}{=} \begin{cases} \dfrac{1}{t - t_k(v_i)} & k \text{ such that } t_k(v_i) < t < t_{k+1}(v_i) \\ 0 & t \le t_1(v_i) \end{cases}$$

where $t_k(v_i)$ refers to the start time of the k^{th} execution of operation v_i. Assuming a synchronous execution model with cycle time τ, the (lattice) rate of execution at invocation k of an operation v_i is given as the inverse of the time interval between its current and previous execution. That is,

$$\begin{aligned} \rho_i(k) &\overset{\triangle}{=} \widetilde{\rho}_i(t)\ |_{t\,=\,t_k(v_i)} \\ &= \frac{1}{t_k(v_i) - t_{k-1}(v_i)} \quad (\text{sec}^{-1}) \\ &= \frac{\tau}{t_k(v_i) - t_{k-1}(v_i)} \quad (\text{cycle}^{-1}) \end{aligned} \tag{2}$$

By convention, the instantaneous rate of execution is 0 at the first execution of an operation ($t_0 \to -\infty$). Note that ρ_i is defined only at times when operation v_i is executed whereas $\widetilde{\rho}_i$ is a function of time and defined at all times. In our treatment of execution rates and constraints on rates, only rates at times of operation execution are of interest. Hence we use the definition of ρ as the rate of execution.

For a graph model, G, its *rate of reaction*, is defined as the rate of of execution of its source operation, that is,

$$\varrho_G(k) \overset{\triangle}{=} \rho_0(k) \tag{3}$$

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

$$\varrho_G(k)^{-1} = \lambda_k(G) + \gamma_k(G) \tag{4}$$

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. For a pipelined implementation, the *degree* of pipelining determines the reaction rate of G. As the number of pipestages increases, the reaction rate of the graph model increases. With appropriate choice of pipeline buffers, it is possible to accommodate different rates of execution for operations in a graph model.

2.3 Non-determinism, Execution Rate and Communication

A flow graph model consists of operations that present 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. In bilogic flow graphs, 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 operating flow graph models, these operations affect the order in which various operations are invoked. Due to this uncertainty, a system model containing these operations is called a *non-deterministic* [13] 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 $\mathcal{ND}$ operations, the rate of execution of all operations is the same and is independent of input data. Therefore, the reaction rate of the graph, G,

$$\varrho_G(k) = \rho_{v_i}(k) \qquad \text{for all} \quad v_i \in V(G) \quad \text{and} \quad \text{for all} \quad k \geq 0$$

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 2.4. Single rate graph models.

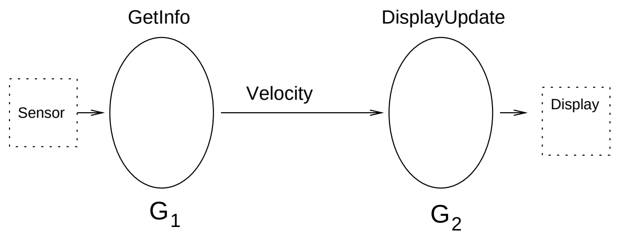


Figure above shows part of a vehicle cruise controller that consists of two single-rate graphs G_1 and G_2 implemented in hardware and software respectively. The latencies of the respective implementations are, $\lambda(G_1) = 25 \cdot \tau_h$ sec with $\gamma(G_1) = 0$ and $\lambda(G_2) = 665 \cdot \tau_s \cdot$ sec with $\gamma(G_2) = 85$ cycles. 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. A multi-rate model has a bounded reaction rate if the model does not contain $\mathcal{ND}$ operations, else it is unbounded.

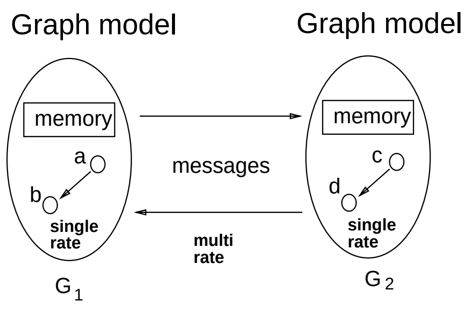


Figure 5: Graph model properties.

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 ensuring 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.

Example 2.5. Use of message passing for communication across two graph models.

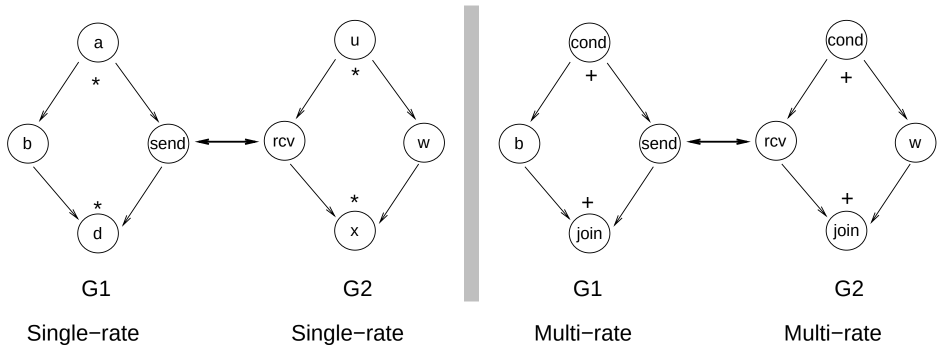


Figure 6: Communication across models.

Figure 6 shows communication across two models, G_1 and G_2. In the first case, G_1 and G_2, are single rate. Since the `send` and `receive` operations are invoked for each execution of the respective graph models, therefore, the rates of execution of operations `b` and `w` are identical. In the second case the execution of synchronization is conditionally invoked. Hardware-software implementations of G_1 and G_2 benefit by this synchronization operation since it allows G_1 and G_2 to run at their reaction rates and synchronize only when a communication is indicated. □

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. However, it is more efficient to implement the communication based on shared-memory. This is because a shared memory communication

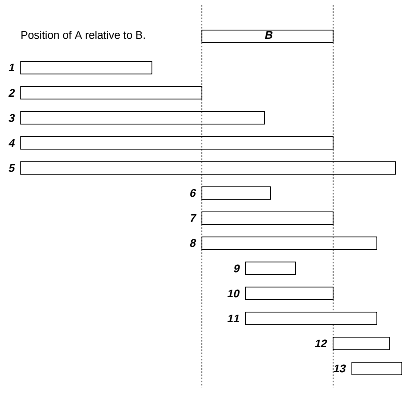


Figure 7: All possible binary timing relations.

uses much less overhead both in operation delay as well as control complexity. In this context, a completely non-blocking message-passing communication can be thought of as a shared memory communication.

In principle, communication between two single-rate models can be accomplished by means of 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 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. Such a situation arises in implementation of a loop body and the loop link operation in the calling graph, where communication takes place by means of shared storage.

3 Timing Constraints and Constraint Analysis

Constraints on timing performance are an important part of embedded systems and depending upon applications different types of constraints are used. For systems used in control applications, real-time response time constraints are most important, while to systems used in on-line transactions and data-processing, synchronization and consistency constraints are of most importance. In general these performance constraints are too abstract to be handled directly on a system model that is described at the level of individual operations. In this work, we first describe timing constraints that apply to the level of individual operations, and a runtime system to support the operation of hardware-software systems and finally develop a relationship of operation-level timing schema to system performance parameters in the context of the runtime system environment in Section 4.

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 is defined as:

$$t_k(v_j) \geq t_k(v_i) + l_{ij} \tag{5}$$

Similarly a *maximum timing constraint*, $u_{ij} \geq 0$ from v_i to v_j is defined by the following inequality:

$$t_k(v_j) \leq t_k(v_i) + u_{ij} \tag{6}$$

We note that operation delay constraints are general and can be used to capture durational and deadline constraints in specifying real-time systems [14] and all possible binary timing relationships between operations [15] as shown in Figure 7.

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 referred to as *data rate constraints*. A *minimum data rate constraint*, r_i (cycles^{-1}), on an input/output operation defines the lower bound on the execution rate of operation v_i. Similarly, a *maximum data rate constraint*, R_i (cycles^{-1}), on an I/O operation defines the upper bound on the execution rate of operation v_i.

$$\begin{array}{rcll} \rho_{v_i}(k) & \leq & R_i & \forall\, k > 0 \qquad \text{[max rate]} \\ \Rightarrow \quad t_k(v_i) - t_{k-1}(v_i) & \geq & \tau \cdot {R_i}^{-1} & \forall\, k > 0 \end{array} \tag{7}$$

Similarly,

$$\begin{array}{rcll} \rho_{v_i}(k) & \geq & r_i & \forall\, k > 0 \qquad \text{[min rate]} \\ \Rightarrow \quad t_k(v_i) - t_{k-1}(v_i) & \leq & \tau \cdot {r_i}^{-1} & \forall\, k > 0 \end{array} \tag{8}$$

In general, when considering 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 constraint* of 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,

$$r_i^G \leq \rho_{v_i}(k) \leq R_i^G \tag{9}$$

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.

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. 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

$$t_k(v_i) \geq \max_{j \ni v_j > v_i} [t_k(v_j) + \delta(v_j)] \tag{10}$$

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 (5, 6, 7 and 8). Clearly, the satisfaction of timing constraints is related to the choice of the scheduling function. 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 10. A schedule is considered minimum if $|t_k(v_i) - t_k(v_o)|$ is minimum for all $v_i \in V$. For each invocation of G, since the start times of all operations are fixed for all executions of G (that is, for all k), 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 [16] for bilogic flow graphs.

A 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 *relative schedule function* maps vertices to a *set* of integers representing *offsets*. An offset $\theta_{v_j}(v_i)$ of vertex v_i with respect to vertex v_j is defined as the delay in starting execution of v_i after completion of operation v_j. Offsets are determined relative to vertices which the execution of v_i (transitively) depends upon. That is,

$$t_k(v_i) \geq t_k(v_j) + \delta(v_j) + \theta_{v_j}(v_i) \quad \text{if} \quad v_j >^* v_i$$

where $>^*$ represents transitive closure of the dependency relation $>$. For a given vertex, v_i a set, $\mathcal{A}(v_i)$ of *anchor* vertices is defined as the set of conditional ($\mathcal{CD}$) and loop, wait ($\mathcal{ND}$) vertices that have a path to v_i:

$$\mathcal{A}(v_i) = \{v_j \in V : v_j >^* v_i,\ v_j \text{ is } \mathcal{ND} \text{ or } \mathcal{CD}\} \tag{11}$$

A relative schedule function, Ω_r is defined as a set of offsets for each operation such that operation start time satisfies the following inequality:

$$t_k(v_i) \geq \max_{a \in \mathcal{A}(v_i)} [t_k(a) + \delta(a) + \theta_a(v_i)] \tag{12}$$

Since the quantity $\delta(a)$ is known only at runtime, the operation start time under relative schedule is determined only at the runtime.

Inequality 12 can be derived from the inequality 10 by expressing the latter over the transitive closure, $G^{>*}$, of G and then adding the known operation delays, δ, as offsets from unknown delay operations. Clearly, a solution to Inequality 12 will also satisfy Inequality 10 if the offsets, $\theta_{v_j}(v_i) \geq \ell(v_j,\ v_i)$, where $\ell(v_j, v_i)$ refers to the path length from vertex v_j to vertex v_i. Finally, a relative schedule is minimum if it leads to minimum values of all offsets for all vertices.

One of the interesting properties of a relative schedule is that it attempts to express the (spatial) uncertainty associated with conditional invocations of an operation ($\mathcal{CD}$) as its temporal uncertainty by treating it as an unbounded delay ($\mathcal{ND}$) operation. Thus, a conditional operation is same as a data-dependent loop operation where operations on its branches are invoked a variable number of times (0 or 1) depending upon data values. For the purposes of relative scheduling, variable delay operations are treated as unknown delay operations in [17]. The relative scheduler is extended to bilogic relative scheduler by using bounds on the variable delay operations due to conditionals. Depending upon the actual branches taken, this schedule may not be the minimum in the sense of relative scheduling described earlier. However, it reduces the number of $\mathcal{ND}$ operations, thus making it easier to perform the constraint analysis. Also, the cost of implementing control for a bilogic relative scheduler lies somewhere between the control costs for static and relative schedulers.

A *bilogic relative schedule* treats an operation offset as a *vector* $\underline{\theta}_{v_j}(v_i)$ representing the (finite) set of possible delays. A bilogic schedule, Ω_{br} then computes the offset vectors such that

$$t(v_i) \geq \max_{a \in \mathcal{A}_b(v_i)} [t_a + \delta(a) + |\underline{\theta}_a(v_i)|_\infty] \tag{13}$$

where $|\cdot|_\infty$ refers to the largest element (or the infinity norm) of the vector. The bilogic anchor set is defined as $\mathcal{A}_b(v_i) = \{v_j \in V : v_j >^* v_i,\ v_j \text{ is } \mathcal{ND}\}$. Once again, the Inequality 13 can be derived from Ineq 10 for bilogic flow graphs. Thus a solution to Ineq. 13 will also satisfy Ineq. 10 provided $|\underline{\theta}_a(v_i)|_\infty \geq \ell_M(a, v_i)$.

3.2 Constraint satisfiability.

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.

Figure 8 shows the flow of the constraint analysis. The given flow graph model with an implementation and a set of min/max delay and execution rate constraints is input to deterministic constraint analysis that

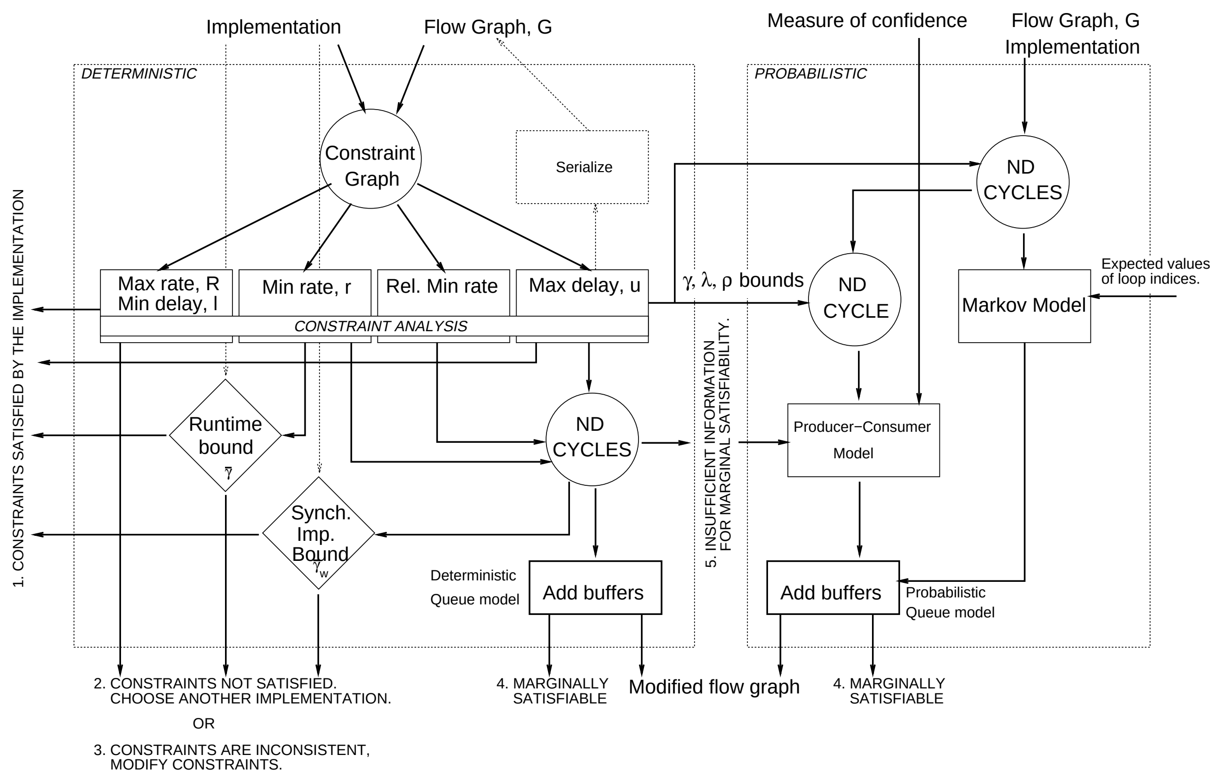


Figure 8: General flow of constraint analysis.

relies on a constraint graph model to determine if the constraints are satisfiable. If the constraints are satisfiable (answer 1 in Figure 8), then the choice of hardware or software implementation is acceptable and the constraint analysis is complete. This means that there exists a possible detailed implementation of the graph model in hardware or software for which the constraints can be satisfied. Conversely, a given set of constraints may be violated by an implementation (answer 2), for example, the operation delays may not be fast enough for the choice of hardware or software. If the constraints are not satisfiable by either hardware or software implementations, there is a possibility that constraints may be inconsistent (answer 3). Constraint analysis in all these cases is complete. On the other hand, constraint analysis may be inconclusive, implying the need for alterations in the *style* of implementation. For example, alternative implementations of the *wait* operation can be explored or buffering can be used to meet execution rate constraints. Such cases are identified by cycles with $\mathcal{ND}$ operations in the constraint graph model.

In presence of cycles with $\mathcal{ND}$ operations in the constraint graph model, constraints may be treated as marginally satisfiable if certain bounds on delay of $\mathcal{ND}$ operations are observed. These (positive) bounds are developed from available slack assuming that the constraints are satisfied (answer 4). In the case of marginally satisfiable constraints, alternative implementations of $\mathcal{ND}$ operations can be explored that improve these bounds. In the last case (answer 5), we need additional information about a measure of confidence (for example, acceptable probability of error) in order to carry out probabilistic analysis. We now summarize the results in constraint analysis.

3.3 Deterministic Constraint Analysis

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 timing constraints are specified in the input HDL description as annotations using tags on individual operations. In general, given

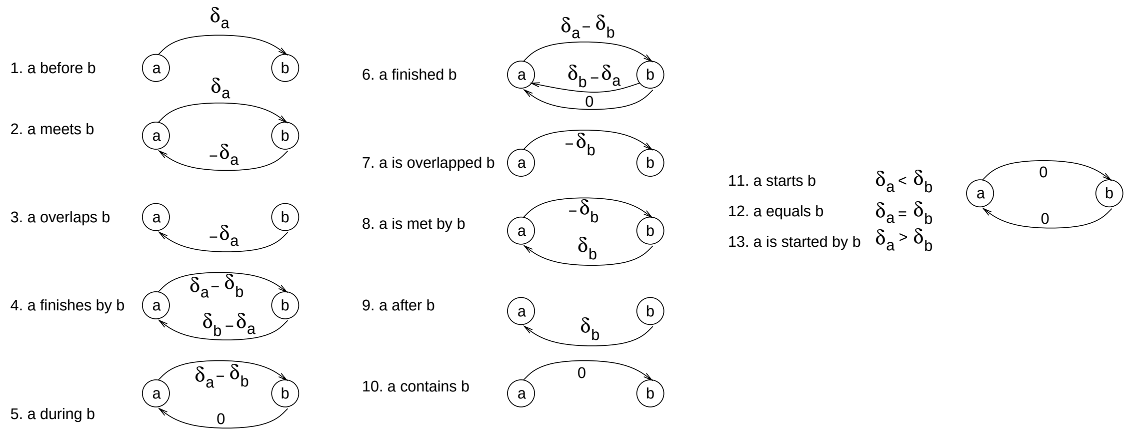


Figure 9: Representation of binary timing constraints.

two operations a and b with non-zero delays δ_a and δ_b respectively, there are a total of thirteen possible binary timing relationships as shown in Figure 7 that are captured using the constraint graph model. The constraints and corresponding constraint graphs are shown in Figure 9.

In the presence of $\mathcal{ND}$ operations, satisfiability 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. For a (bilogic) relative scheduler, a minimum delay constraint is always satisfiable since from any solution that satisfies Inequality 12 or 13 a solution can be constructed such that $\theta_{v_j}(v_i) \geq \max(\ell(v_j, v_i), l_{ji})$ for each constraint l_{ji}. This solution satisfies both Inequalities 10 and 5. On the contrary, 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 [16] lays out a necessary and sufficient condition for to determine the satisfiability of constraints in presence of $\mathcal{ND}$ operations.

Theorem 3.1 (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.*

Execution rate constraints are constraints on the time interval between invocations of the same operation. In general, this interval can be affected by pipelining techniques since pipelining allows one to initiate an operation sooner than what the total latency of the graph model will allow. For deterministic constraint analysis, we consider here only non-pipelined implementations of the flow graph models. (Limited pipelining of operations in the context of $\mathcal{ND}$-cycles discussed in discussed later). Therefore, operations in the graph model are enabled for next iteration only after completion of the previous iteration. We state without proof conditions for checking satisfiability of minimum and maximum rate constraints. For proofs the reader is referred to [18].

Theorem 3.2 (Maximum rate constraint) *A max-rate constraint, R_i, in G is satisfied if $\ell_m(G) \geq R_i^{-1}$.*

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

> **Example 3.1**. Maximum rate constraints.
>
> For the process graph model `example` shown in Figure 4 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. Thus 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. In order to determine constraint satisfiability we need additional input on how frequently the condition is true. For deterministic analysis purposes, we take a two step approach to answering constraint satisfiability:

1. Answer about 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 frequency of 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 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.

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 *overhead* $\gamma_k(G)$ as the delay $[t_{k+1}(v_0(G)) - t_k(v_N(G))]$ and can be thought of as an additional delay operation in series with the sink operation, $v_N(G)$. If G is not a root-level flow graph, then there exists a parent flow graph G_+ that calls G by means of a link operation, say v. It can be shown that $\gamma_k(G)$ is upper bounded by $\overline{\gamma}$ defined as follows:

$$\gamma_k(G) \leq \overline{\gamma}(G) \triangleq [\ell_M(G_+) + \overline{\gamma}(G_+)] - \ell_m(G) \tag{14}$$

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, infact, be unbounded. We summarize the satisfiability of a minimum rate constraint in the following theorem (see [18] for a proof).

Theorem 3.3 (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 minimum available overhead for the runtime scheduler, γ_{avail} is greater than the maximum delay $\overline{\gamma}_M$ offered by the chosen runtime scheduler. That is,*

$$\overline{\gamma}_{\text{avail}} \triangleq \frac{\tau}{r_i} - \ell_M(G) - \sum_{G_i=G_+}^{G_o} [\ell_M(G_i) - \ell_m(G_i)] - [\ell_m(G_o) - \ell_m(G)] \geq \overline{\gamma}_M \tag{15}$$

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 the 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 a $\mathcal{ND}$-cycle in the constraint graph. According to Theorem 3.1, 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 implications of developing bounds on the delay of these operations is as follows:

- Let us first consider synchronization related $\mathcal{ND}$ operations. Since there are multiple ways of implementing a synchronization operation, the effect of the bound is to choose those implementations which are *most likely* to satisfy the minimum rate constraint. Thus, a bound on the delay of the synchronization refers to a bound on the delay offered by the *implementation* of the $\mathcal{ND}$ operation. The implementation delay of a synchronization operation is referred to as 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. For such an implementation, the minimum rate constraint is interpreted as the rate supportable by an implementation. With this interpretation, the $\mathcal{ND}$ operations are considered non-$\mathcal{ND}$ operations with a fixed delay, $2 \cdot \gamma_w$.

- Next, 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 model contains at most one 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 16 that is derived later. 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.

For a relative minimum rate constraint constraint relative to G, the overhead term $\overline{\gamma}(G)$ in Equation 16 is assigned zero value. In general, however, 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_+. 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. The following theorem lists the condition for checking constraint satisfiability.

Theorem 3.4 (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*

$$\overline{x}_v \triangleq \left\lfloor \frac{\tau r_i^{-1} - \overline{\gamma}(G) - \ell_M(G) + \mu(v)}{\ell_M(G_v)} \right\rfloor + 1 \tag{16}$$

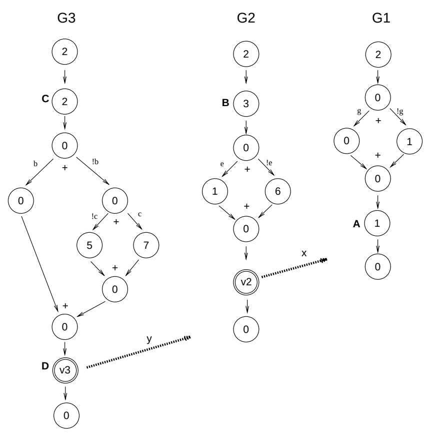
where $\mu(v)$ refers to the mobility *of operation v and is defined as the difference in length of the longest path that goes through v and ℓ_M.*[3] *G_v refers to the graph model called by the $\mathcal{ND}$ operation v and the overhead bound, $\overline{\gamma}(G)$ is defined by Equation 14.*

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 16 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

[3]The mobility is computed in $O(|E(G)|)$ time as the difference in 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.

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 3.2. 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:

$$\begin{aligned}
\overline{\gamma}(G_2) &= \Delta\ell(G_3) + \overline{\gamma}_M + \ell_m(G_3) - \ell_m(G_2) \\
&= 13 + 0 + 13 - 9 = 17 \\
\overline{x}_2 &= \left\lfloor \frac{\tau r_B^{-1} - \overline{\gamma}(G_2) - \ell_M(G_2) + \mu(v_2)}{\ell_M(G_1)} \right\rfloor + 1 \\
&= \left\lfloor \frac{50 - 17 - 15 + 0}{4} \right\rfloor + 1 = 5.
\end{aligned}$$

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.

Delay constraints across graph models. So far we have considered constraint satisfiability analysis by analyzing one flow graph at a time. We now examine timing constraint between operations that belong to two separate flow graphs G_1 and G_2. The satisfiability constraints that span across flow graphs is affected by the relationships between the flow graphs. As shown in Figure 10 there are following three types of relationships between flow graphs:

Case A: G_1 and G_2 are concurrent. This refers to the case when invocation paths to G_1 and G_2 are disjoint. If the graphs G_1 and G_2 share the parent process graph, there are referred to as single-rate models ($\natural$), otherwise the reaction rate of the graphs can be multi-rate and is indicated by ($||$). For

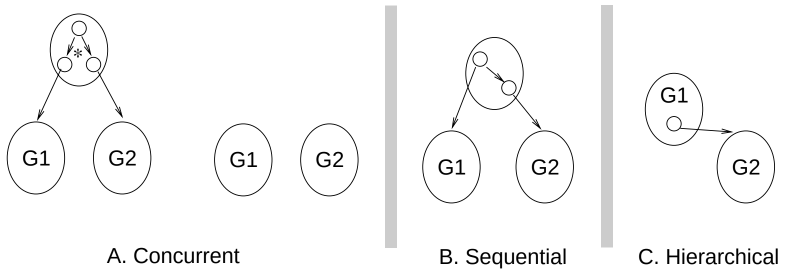


Figure 10: Relationships between flow graphs

operations with multi-rate executions, operation delay constraints are considered between all execution events of the respective operations. Verification of such constraints is a difficult problem. Such constraints are not allowed in our formulation of hardware-software cosynthesis.

For operations with single-rate executions, a composite graph model is constructed by merging the respective source and sink vertices of G_1 and G_2 into a single source or sink vertex of G_{12} respectively.

Figure 11 shows an example where $G_{1*}||G_{2*}$. Since G_1 and G_2 are process graphs no constraints across different hierarchies are supported. However, constraints across G_{21} and G_{22} are analyzed by the constraint graph obtained by composing G_{21} and G_{22} in parallel. Note that disjoined forks lead to mutually exclusive paths, therefore, by definition there can not be constraints on operations that belong to separate conditional paths.

Case B: G_1 and G_2 have a sequential dependency. In this case, a composite constraint graph is constructed either as a serialization from G_1 to G_2 or vice-versa depending upon the ordering relation between G_1 and G_2. A composite constraint graph construction $G_{1;2}$ as a serialization from G_1 to G_2 is carried out by adding an edge from sink of the predecessor graph G_1 to the source of the successor graph G_2. The inter-graph constraints are then added and constraint analysis is carried out on the composite constraint graph model. For example, G_{11} and G_{12} in Figure 11 are sequentially related.

Case C: G_1 and G_2 belong to the same hierarchy. Verification of these constraints is carried out by propagating these constraints upwards until these are applicable to the operations in the same graph model. For example, $G_1 \succ\!\!- G_{11} \succ\!\!- G_{111}$ and $G_1 \succ\!\!- G_{12}$. Similarly for G_2. These graphs belong to the same control hierarchy. Constraints across graph models are considered to be constraints on respective link operations in the parent graph. For instance a constraint that applies to an operation, v_i in G_1 and another operation in G_{11} is treated as a constraint across operations v_i and the link operation v_{11} in G_1.

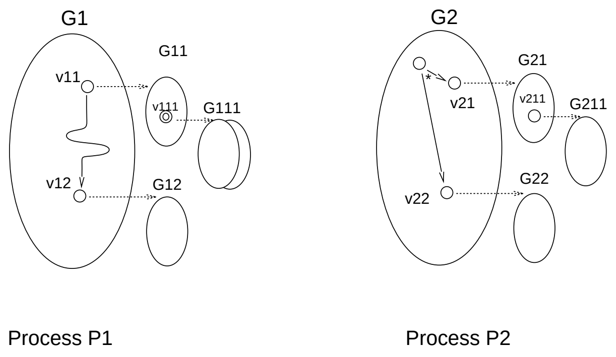


Figure 11: Graph model hierarchy example.

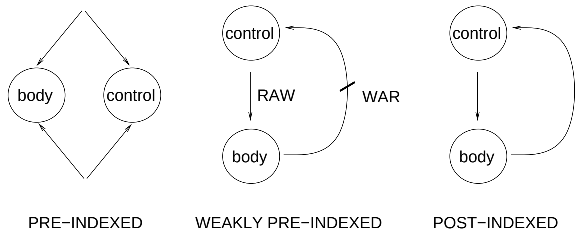


Figure 12: Types of loop operations.

3.4 Constraint Analysis and Loop Operations

A loop is defined by the loop link operation v in the calling graph G and a called graph G_v. The called graph $G_v = (V, E)$ consists of two sets of operations: those relating to loop control and those relating to loop body: $V = V_b \cup V_c$. Operations in V_c are *user specified* as a part of the loop control operations and typically consists of operations relating to loop condition evaluation, loop register loads and (possibly) modification of the loop index. An execution of G_v consists of a finite number of iterations of V_c and V_b. We assume that each loop is controlled by a single variable index. A loop index value, x_v, marks execution of loop body until the loop exit condition becomes true. The loop index x_v is defined as a shared storage between G and G_v. An operation in V_b either reads the loop index, or writes the loop index or is independent of the loop index. That is, the operations in the loop body can be partitioned into $V_b = V_{br} \cup V_{bw} \cup V_{bn}$ where V_{br} is the set of operations that read the loop index; V_{bw} is the set of operations that modify the loop index and V_{bn} is the set of operations that do not read or modify loop index. As shown in Figure 12 there are following three types of loop operations:

Pre-indexed. If $V_{br} = V_{bw} = \emptyset$. That is, loop body operations do not affect the loop index.
Weakly pre-indexed. If $V_{bw} = \emptyset$. That is, loop body operations use but do not modify the loop index.
Post-indexed. If $V_{bw} \neq \emptyset$. That is, the loop index is modified by the loop body.

The number of invocations of pre-indexed loops are marked by a loop index variable that is assigned a value at run time but *before* the execution of the loop body is started. This is in contrast to post-indexed loops where the number of iterations of loop body are determined by the body of the loop operation.

For a graph G with loop link operation v, a minimum execution rate constraint on any operation *(other than v)* in G will cause an $\mathcal{ND}$-cycle in the corresponding constraint graph of G. In addition, a maximum delay constraint in G may also cause an $\mathcal{ND}$-cycle. We saw earlier that constraint satisfiability for $\mathcal{ND}$-cycle leads to a bound, $\overline{x}$, on the number of times the loop body G_v can be invoked for each invocation of the loop operation, v. We now consider the ways in which this bound can be improved by altering the implementation of the loop operation.

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., $M(G) \cap M(G_v)$. For preindexed loop operations, as shown in Figure 13, 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. Based on these bounds, an algorithm is presented that first determines a bounded interval over which this queue is guaranteed to be empty (that is, number of productions equals number of consumptions). It then finds a bound on the queue depth based on the transient behavior of the queueing system over this finite interval. Recently Kolks *et. al.* [20] have proposed use of implicit state enumeration techniques to determine size of buffers between communicating finite state machines. The procedure is based on representing the buffer as a finite state machine by modeling it as a counter. State reachability analysis on the network of interacting finite state machines is performed to determine the maximum value of the

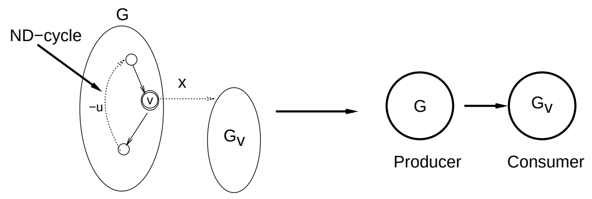


Figure 13: Modeling an $\mathcal{ND}$ loop as a producer-consumer system

counter used and thus the minimum size of the buffer is determined. This approach is elegant when all parts of a system design can be conveniently modeled as finite state machines. Like the approach in [19] it also considers worst case bounds by examining worst case data values.

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 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 3.3. In order 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,

$$\begin{aligned} \ell'_m &\geq x_v \cdot \ell_M(G_v) \\ \Rightarrow \quad x_v &\leq \frac{\ell'_m}{\ell_M(G_v)} \end{aligned} \tag{17}$$

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:

$$B_1 = \left\lfloor \frac{\ell'_m}{\ell_M(G_v)} \right\rfloor \tag{18}$$

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. We now consider a way of extending this bound by altering the structure of the loop operation.

Use of buffers to extend bounds on loop index. 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

$$B_k = \left\lfloor \frac{k \cdot \ell'_m}{\ell_M(G)} \right\rfloor \tag{19}$$

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 in order 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. 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. We assume software implementation such 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.

3.5 Marginal Satisfiability and Probabilistic Constraint Analysis

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 can be addressed effectively by relating the possibility of constraint violation to the possibility of exceeding a specified bound on the delay of the $\mathcal{ND}$ operations. For a given $\mathcal{ND}$-cycle, a constraint violation occurs if the delay, δ, offered by the $\mathcal{ND}$ operation exceeds a (deterministic) bound, $f(u)$, where δ is a random variable. 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 would be to find supportable rate constraints that minimize some measure (absolute, mean-square, etc.) of the violation error.

An alternative approach is to determine supportable rate constraints that limit the probability of constraint violation below some acceptable limit, ϵ. Or, as is possible in the case of preindexed loop operations, find an appropriate size of the buffer that contains probability of a given constraint violation below a given limit. This notion 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 *marginally satisfiable*. We take this interpretation to solving satisfiability problem for delay and rate constraints.

For illustration purposes, let us consider a max-delay constraint of Equation 6, $t_k(v_j) - t_k(v_i) \le u_{ij}$. 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 $\epsilon, 0 \le \epsilon \le 1$,

$$\Pr\{t_k(v_j) - t_k(v_i) > u_{ij}\} \le \epsilon$$

That is, for each k, the check for constraint satisfiability is considered a trial. A marginally satisfiable constraint is then found to be satisfied 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. The answer depends upon the choice of the probability distribution function for the random variable. Based on the distribution of loop index, the buffer depth and the probability of buffer being full is derived. 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 a preindexed $\mathcal{ND}$ 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.

[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.

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 preindexed $\mathcal{ND}$ 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 $\epsilon \rightarrow 0$, 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 has the least information (or highest entropy) and is therefore the most random law that can be used and thus the most conservative approach [21]. 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], $f_e(x) = \mu e^{-\mu x}$ with expected value, $E_e[X] = \frac{1}{\mu}$ and variance $\sigma_e^2 = \frac{1}{\mu^2}$.

Theorem 3.5 *For a given error probability, ϵ, the following express the bounds on buffer depth, k*

$$N_e \quad = \quad \left\lceil \frac{\ln \epsilon}{\ln(\frac{B_1}{\overline{x}}) - \ln(e^{B_1/\overline{x}} - 1)} \right\rceil \approx \left\lceil \frac{-\ln \epsilon}{\frac{B_1}{\overline{x}} - \ln(\frac{B_1}{\overline{x}})} \right\rceil \tag{20}$$

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

Its proof can be found in [18]. Note that as $\epsilon \rightarrow 0$, $N_e \rightarrow \infty$. Figure 14 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 20 is complex [18] and not very useful as a general formula. As an alternative, the Equation 20 can be solved numerically using a solver like Maple [22] 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 present the notion of a runtime system and its application in hardware-software implementations followed by the generation of software from the flow graph model.

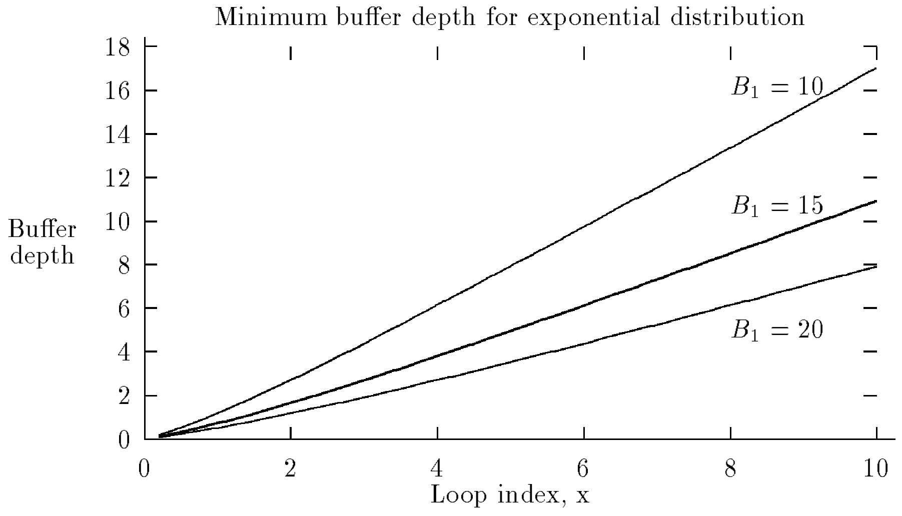


Figure 14: Buffer depth for exponential distributions ($\epsilon = 0.01\%$)

4 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 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 can be thought of consisting 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. The results of the scheduling and resource management tasks are interdependent, that is, a choice of a schedule affects allocation/binding and vice versa. Depending upon the nature and availability 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 15. A thread is defined as a linearized set of operations that may or may not begin by 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 defined as sum of the delay of its operations without including the initial $\mathcal{ND}$ operation whose

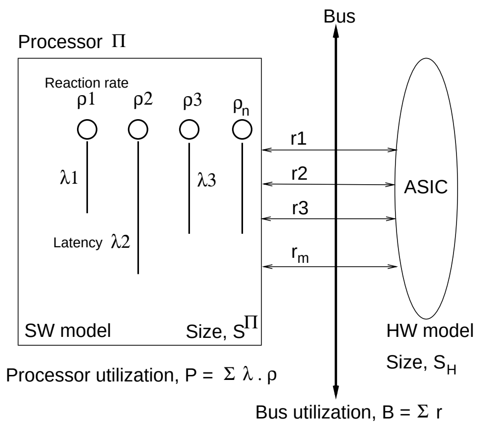


Figure 15: Software model to avoid creation of $\mathcal{ND}$ cycles.

delay is merged 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

$$\mathcal{P} \triangleq \sum_{i=1}^{n} \lambda_i \cdot \varrho_i \tag{21}$$

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,

$$\mathcal{B} \triangleq \sum_{j=1}^{m} r_j \tag{22}$$

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. In general, the runtime scheduler is one of the following two types:

- **Non-preemptive runtime scheduler.** Here a program thread executes either to its completion or to the point when it detaches itself voluntarily (for example, to observe dependence on another program thread). Most common examples of non-preemptive schedulers are first-in-first-out (FIFO) or round-robin (RR) schedulers. FIFO schedulers select a program thread strictly on the basis of the time when it is enabled. A RR scheduler repetitively goes through a list of program threads arranged in a circular queue. A non-preemptive scheduler may also be *prioritized* or *non-prioritized.* Priority here refers to the selection of program threads from among a set of enabled threads. Both FIFO and RR maintain the order of data arrival and data consumption and, therefore, avoid starvation. A prioritized discipline may, however, lead to starvation.

- **Preemptive runtime scheduler**. These schedulers provide the ability to preempt a running program thread by another program thread. Preemption generally leads to improved response time to program threads at increased cost of implementation. This ability to preempt is tied to an assignment of priorities to program threads. The primary criterion in design of a preemptive scheduling scheme is in selection of an appropriate priority assignment discipline that leads to a feasible schedule. Priority selection can be *static* where the priorities do not change at run time, for example rate-monotonic priorities [23], or *dynamic*, for example deadline-driven priorities [24].

We have so far implemented only non-preemptive runtime scheduling techniques. The reason for this choice is to keep the synthesized software simple. The implementation of multiple program threads in a preemptive runtime environment leads to additional states (in addition to being *detached* or *running*) for the program threads which adds to the overhead delay caused by the runtime scheduler. Use of preemptive runtime schedulers makes the detailed timing constraint analysis developed in this work inapplicable.

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. This observation has been used in analysis of several runtime scheduling algorithms and has been formalized by Mok [25]. 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.,

$$\mathcal{P} \leq 1 \tag{23}$$

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 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:

$$\frac{1}{\varrho} \geq \sum_{\forall \text{ threads } k} \lambda_k \tag{24}$$

This inequality follows from the case when all the program threads are enabled simultaneously. In this case, 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 Inq. 24. From Inq. 24, a sufficient condition for software reaction rate satisfiability is to ensure that

$$\frac{1}{\varrho_{\max}} \geq \sum_{\forall \text{ threads } k} \lambda_k \tag{25}$$

where $\varrho_{\max} = \max_i \varrho_i$ defines the maximum reaction rate over all program threads. It is interesting to note that the above condition for worst case reaction rate also applies for RR schedulers, though the average case performance differs.

Prioritized runtime scheduler. A prioritized FIFO scheduler consists of a *set* of FIFO buffers that are prioritized such that after completion of a thread of execution the scheduler goes through the buffers in the order of their priority. The program threads are assigned a specific priority ψ and are enqueued in the corresponding FIFO buffer. Thus, among two enabled program threads, the one with the higher priority is selected. The effect of this priority assignment is to increase the *average* reaction rates for the program threads with higher priority at the cost of decrease in the *average* reaction rate for the low priority threads. Recall that in a non-prioritized scheduler the supportable reaction rate is fixed for all threads as the inverse of the sum over all thread latencies. Unfortunately, the worst case scenario gets considerably worse in case of a prioritized scheduler and it is possible that a low priority thread may never be scheduled due to starvation.

The average case performance improvement can be intuitively understood by the following analysis. Consider a software of n threads, $T_1, T_2, \ldots, T_n$ with reaction rates, $\varrho_1, \varrho_2, \ldots, \varrho_n$ respectively. Let $\psi(T_i)$ be the priority assignment of one of the levels from 1 to l, with l being the highest priority and 1 being the lowest priority. For each thread, let us define *background processor utilization* as

$$\eta(T_i) = \sum_{\forall\ T_k\ \ni\ \psi(T_k)<\psi(T_i)} \lambda_k \cdot \varrho_k \tag{26}$$

That is, $\eta(T_i)$ provides a measure of the processor utilization by the set of program threads with priority strictly lower than $\psi(T_i)$. Thus the *available* processor utilization for threads with priorities greater than equal to $\psi(T_i)$ is given by $1 - \eta(T_i)$. On an average, this can be seen as an extension in the latency of program threads. Let us define an *effective latency*, λ' as

$$\lambda'_i = \frac{\lambda_i}{1 - \eta(T_i)} \tag{27}$$

The average case feasibility condition is now defined as

$$\frac{1}{\varrho(T_i)} \geq \sum_{\text{thread } T_k \ni \psi(T_k) \geq \psi(T_i)} \lambda'_k \tag{28}$$

For this model of software, the satisfiability of constraints on operations belonging to different threads is checked for marginal or deterministic satisfiability, assuming a bound delay on the scheduling operations associated with $\mathcal{ND}$ operations. Constraint analysis for software depends upon first arriving at an estimate of the software performance and size of register/memory data used for the software. We discuss these two issues next.

4.1 Estimation of 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,

$$S^{\Pi}(\Phi) = \sum_{G_i \in \Phi} S^{\Pi}(G_i) = \sum_{G_i \in \Phi} [S_p^{\Pi}(G_i) + S_d^{\Pi}(G_i)] \tag{29}$$

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[5] 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, $S_d^{\Pi}(G)$ would correspond to a subset of the variables used to express a software implementation of G, that is,

$$S_d^{\Pi}(G) \leq |M(G)| + |P(G)| \tag{30}$$

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 an 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/reallocation operations*. Due to a single-processor target architecture, the cumulative operation delay of $V(G)$ would be constant under any schedule. However, the data set $\mathcal{S}_d^{\Pi}(G)$ of G would vary according to scheduling technique used. Accordingly, the number of operations needed to perform the requisite data transfer would also depend 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 requires solution to the vertex coloring problem for a conflict graph where the vertices correspond to variables and an edge indicates simultaneously live variables. The number of available colors corresponds to the number of available machine registers. It has been shown that this problem is NP-complete for general graphs [26]. Hence heuristics

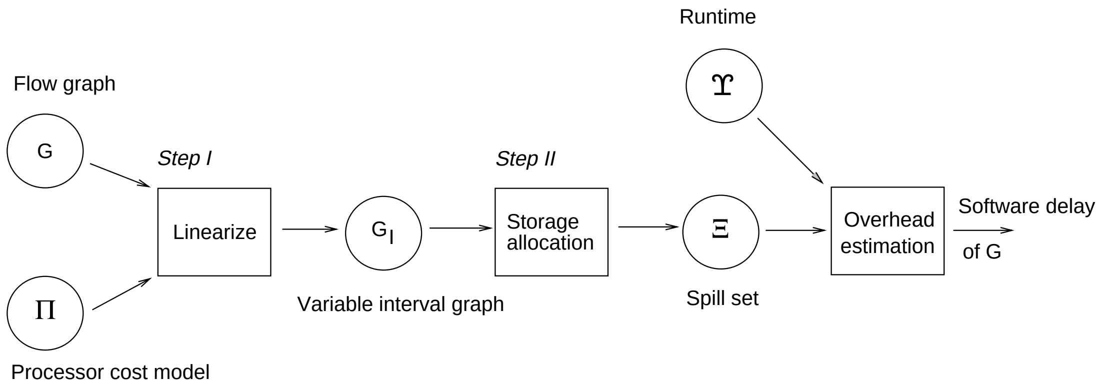


Figure 16: Software delay estimation flow.

solutions are commonly used. Most popular heuristics for code generation use a specific order of execution of successor nodes (e.g., left neighbour first) in order to reduce the size of $\mathcal{S}_d^{\Pi}$ [27].

In contrast to the register assignment in conventional software compilers which perform simultaneous register assignment and operation linearization, we peform operation linearization that takes into account detailed timing constraints. We explain this procedure in two steps as shown in Figure 16:

- Estimation of register/memory **data transfer operations**.
- **linearize operations**, that is, find a schedule of operations.

We now present the estimation of register/memory operations in the context of software delay estimation next.

4.2 Estimation of Software Performance

Previous work in software estimation has been to annotate programs with relevant timing properties [28, 29]. Syntax-directed delay estimation techniques have been tried [30, 31] which provide quick estimates based on the language constructs used. However, syntax-directed delay estimation techniques lack timing information that is relevant in the context of the semantics of operations such as loop and conditionals. We perform 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. We consider the delay due of a software implementation of the operations in a flow graph model. 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:

$$\eta(v) = \sum_{i=1}^{n_o(v)} t_{op_i} + (m_r(v) + m_w(v)) \times m_i \tag{31}$$

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 function of some operations is often

[5]To be precise, $\mathcal{S}_d^{\Pi}$ also includes the initialized data storage or the bss.

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, some estimation rules must be 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, that is, $v_i \succ v_j$. Vertices with no predecessors are called source vertices and vertices with no successors are defined as sink vertices. Let $i(v), o(v)$ be the indegree and outdegree of vertex v. Let $n_i = |\{\text{source vertices}\}|$ and $n_o = |\{\text{sink vertices}\}|$. 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 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 RM/MM 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, Ξ.

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,

$$m_r = |E| \tag{32}$$
$$m_w = |V| \tag{33}$$

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.

$$m_r = n_i \leq |V| \leq |E| \tag{34}$$
$$m_w = n_o \leq |V| \tag{35}$$

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 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.

$$m_r = n_i + \sum_{\Xi} o(v_i) \leq \sum_{V} o(v_i) = |E| \tag{36}$$
$$m_w = n_o + |\,\Xi\,| \leq |V| \tag{37}$$

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. As mentioned earlier, the notion of liveness of an output $o(v)$ of an operation, v can be abstracted into a *conflict graph.* The optimum coloring of this graph would provide a solution to the optimum spill set problem. This problem is shown to be NP-complete for general graphs [32].

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* [27] 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. (Note that in the case of write and logic operations, the left-hand of the assignment also generates an rvalue that is subsequently assigned to an lvalue.) 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.

4.3 Operation Linearization

We now consider the issue of generation of a program thread from the flow graph model. Program thread consists of linearization of operations in the flow graph. A flow graph model 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.* A thread must be enabled first in order to run. In case of concurrent program threads in a single-processor software implementation, concurrency between program threads is achieved by using an interleaved execution model. 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 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. Typically, we consider a variable that is tested also as a used variable. However, a distinction between the two is made if the variable in question is used solely as an argument of a condition testing.

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. There are two situations when a flow graph is not serializable. We discuss these conditions and transformations below. First condition when this may happen is 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;
3. Find a linearization of the new flow graph assuming each simple operation to be an atomic operation.

Example 4.1 below shows the transformation.

Example 4.1. 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 the figure below.

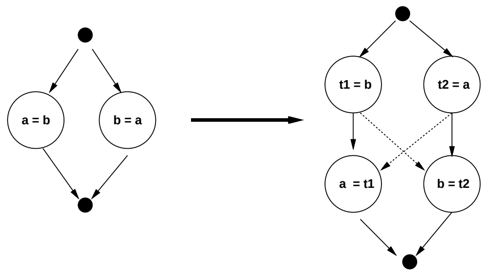


Figure 17: Creation of serializable graphs.

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

A second situation arises in the case of dedicated loop operations. A hardware implementation of a loop operation is by means of shared memory [33]. The operations in body of the loop have access to the storage defined in the calling graph. In this case, the program thread corresponding to the loop body may have access to storage in the program thread corresponding to the parent graph. This situation can be simplified by making explicit all data transfers between the loop graph and the parent graph as shown in Figure 18. Depending upon the data transfer, the parent graph may or may not block pending execution of the loop operation.

Finally we note that since wait operation is allowed only on a single input signal, it is possible that a software implementation of waiting on multiple signals may lead to a deadlock. For this reason, only single-variable wait operations are currently supported.

Linearization Techniques. Linearization of G 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:

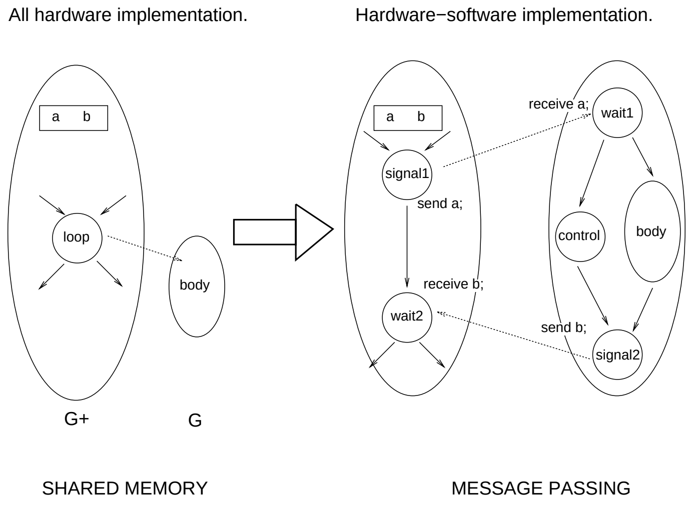


Figure 18: Shared-memory versus message-passing implementations of loop operation.

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* $\mid \Xi \mid$ *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 [26]. 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 [9] that considers all possible valid orders in order to find one that meets the imposed timing constraints. In [34] 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 subset of vertices based on a two-part criteria. One criterion is that the selected vertex does not create any additional dependencies or does not modify weights on any of the existing dependencies in the constraint graph. For the second criterion, we associate a measure of *urgency* with each source operation and select the one with the least value of the urgency measure. 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 based on Liao-Wong algorithm [35] 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.

The two criteria for vertex selection are applied in reverse order if a linearization fails. At any time, if a vertex being output creates a serialization not in the original flow graph, a corresponding edge is added in the constraint graph with weight equals delay of the previous output vertex. With this serialization, the constraint analysis is performed to check for positive cycles, and if none exists, the urgency measure for the remaining vertices is recomputed by assigning the new start times, else the algorithm terminates without finding a feasible linearization.

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 [36].

5 Partitioning

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. Fortunately, to a great extent partitioning at various levels of abstraction can be carried out independently so long as the objectives of partitioning are kept distinct. 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 by no means a substitute for 'conceptual' partitioning usually carried out at higher levels of abstractions. Indeed, it attempts to supplement the conceptual design process by providing the system designer a means to handle the complexity associated with a detailed design description consisting of language-level operations. Therefore, the problem of partitioning here refers specifically to a partition of the system functionality represented by a flow graph model. A flow graph model contains a hierarchy of flow graphs that implements a function or a process. This partitioning can be developed as a collective set of *transformations* on the flow graph model that achieve the correct separation of functionality as embodied by the flow graphs. 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.

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.

5.1 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 15. The software performance and size parameters were discussed in the 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 [37].

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,*

$$f(\varpi) = a_1 \cdot S_H(\Phi_H) - a_2 \cdot S^{\Pi}(\Phi_S) + b \cdot \mathcal{B} - c \cdot \mathcal{P} + d \cdot |m| \qquad (38)$$

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.

$$S_H(\Phi_H) = \sum_{G_i \in \Phi_H} S_H(G_i) = \sum_{G_i \in \Phi_H} \sum_{v \in V(G_i)} S(v) \qquad (39)$$

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 in order to ensure correct functionality of the resulting partitioned graph models. The effect of a move of an operation on data storage is to change the 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

$$S_d^{\Pi}(v) = 0 \Rightarrow S^{\Pi}(\Phi_S) = \sum_{G_i \in \Phi_S} \sum_{v \in V(G_i)} O(w(v)) \tag{40}$$

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. For our implementations, 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 as explained by the following pseudo-code.

```
▷I     construct initial partition, $\varpi \leftarrow \varpi_0$
       compute $f(\varpi_0)$
       repeat {
▷II        select a group of operations to move
           create new partition, $\varpi'$
▷III       compute $f(\varpi')$
           if $f(\varpi') < f(\varpi)$
               accept move: $\varpi \leftarrow \varpi'$
       } until no more operations to move
```

Before describing the heuristics, 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 [38] or Fidducia-Matheysis [39], 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.

Local versus global properties. We need to capture not only the effects of *sizes* of hardware and software parts but also the effect of *timing* behavior of these portions, as captured by processor and bus utilization, into the partition cost function. During 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. Part of the problem lies in the fact that timing properties are usually *global* in nature, thus making it difficult to make incremental computations of the partition cost function which is essential in order to develop effective partition algorithms. 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, there exists a spectrum of techniques in partitioning, and use of timing properties in driving the partitioning search process. On one end of the spectrum are partitioning schemes for hardware circuits which are mostly focussed on optimizing area (and pinout) of resulting circuits and do not use timing properties. On the other end of the spectrum are partitioning schemes in software world where the objective of partitioning is to create a set of programs to execute on multiple processors. These schemes do make extensive use of statistical timing properties in order to drive the partitioning algorithm [40]. 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.[6] 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 in terms of a set of program threads as shown in Figure 15 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 system functionality between 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.

5.2 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 off 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 presence of multi-rate data transfers, feasibility of hardware-software partition is determined by the fact 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 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 buids 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:

[6] Partitioning for unscheduled flow graphs was considered in [41] 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.

1. for all $G_i \in \Phi_H$, G_i is hardware feasible,
2. for all $G_i \in \Phi_S$, G_i is software feasible,
3. For all program threads T_i in software, Φ_S, $\frac{1}{\max \varrho(T_i)} \geq \sum_k \lambda_k$. This condition also ensures that processor utilization is below unity.
4. Bus bandwidth, $\mathcal{B} \leq \overline{\mathcal{B}}$.

Our procedure for partitioning a flow graph model is based on the iterative improvement procedure presented in Section 5.1. 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 is 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 partitioning feasible more, 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:

$$\Delta f = a_1 \, \Delta S_H - a_2 \, \Delta S^{\Pi} + b \, \Delta \mathcal{B} - c \, \Delta \mathcal{P} + d \, \Delta \, |m| \tag{41}$$

1. Hardware size S_H is reduced by the size attribute of the vertex according to Equation 39. 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 40. 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

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

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

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

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, $\Pi$, Bus bandwidth $\overline{\mathcal{B}}$, Runtime overhead, $\overline{\gamma}$
Output: Partitioned system graph model, $\Phi = \Phi_S \cup \Phi_H$

partition($\Phi$) {
    $\Phi_S = \Phi_H = \emptyset$;
    for each process graph in $G_i \in \Phi$ {
        graph_partition($G_i$);                     /* returns a tagged graph, $G_i$ */
        $(G_H, G_S)$ = partition_transformation($G_i$);   /* create separate graphs */
        $\Phi_S = \Phi_S \cup \{G_S\}$;
        $\Phi_H = \Phi_H \cup \{G_H\}$;
    }
}
graph_partition(G) {
    $V_H = V$;
    $V_S = \emptyset$;
 ▷I for $v \in V(G)$ {                               /* create initial partition */
        if $v$ is an $\mathcal{ND}$ link operation
            $V_S = V_S + \{v\}$;
    }
    create software threads ($V_S$);                 /* create initial $|V_S|$ routines */
    compute reaction rates, $\varrho$ 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 $v \in V_H$ and $v$ is not $\mathcal{ND}$ {   /* 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 $\Delta f > 0$
            $V_H = V_H - \{v\}$;                     /* move this operation to sw */
            $V_S = V_S + \{v\}$;
            $f_{\min} = f_{\min} - \Delta f$;
            update software threads;
            update thread reaction rate of the destination thread;
            for $u \in$ succ($v$) and $u \in V_H$      /* identify successor for move */
                move($u$);
    return $f_{\min}$;
}
check_feasibility($\Phi_H, \Phi_S$) {                 /* check partition feasibility */
    for all $G_i \in \Phi_H$                         /* check timing constraints for hardware */
        if not check_satisfiability($G_i$);
            return not feasible;
    for all $G_i \in \Phi_S$                         /* check timing constraint for software */
        if not check_satisfiability($G_i$);
            return not feasible;
    for all $T_i \in \Phi_S$
        if ($\frac{1}{\max \varrho(T_i)} < \sum_k \lambda_k$)   /* runtime scheduler */
            return not feasible;
    if $\mathcal{B} > \overline{\mathcal{B}}$                  /* check bus utilization */
        return not feasible;
    return feasible;
}
```

The markers $\triangleright$ 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 $O(|V|^4)$.

6 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, 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 [42] and DLX processor compilation and simulation tools [43], it provides a complete path for synthesis of hardware and software form *HardwareC* descriptions.

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

Example 6.1. 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|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>
```

□

This input is first compiled into a sequencing graph model (SIF) using the program HERCULES [44] by applying a series of compiler-like transformations. The model is translated into the bilogic flow graph model. The organization of data in Vulcan is shown in Figure 19. 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 model 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 routines automatically update the list of models after transformations and update the cost models and attributes with the result of analysis.

Vulcan provides an interactive menu-driven interface that is modeled after the Unix shell [42]. This interface provides the typical shell commands related to directory and file management, input/output redirection,

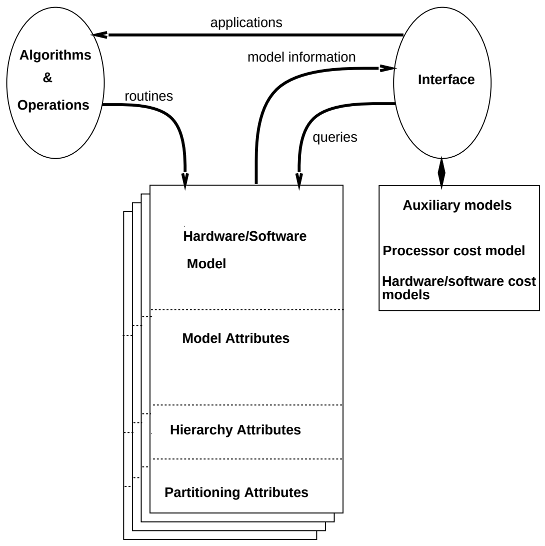


Figure 19: Data Organization in VULCAN

aliasing and history commands. Three levels of command complexity is supported and command abbreviation is provided for advanced users. This user interface is supported across all the tools in the Olympus Synthesis System, thus making it convenient for the user to move among the tools in the same session.

Figure 20 shows the organization of Vulcan subsystems and their relationship to hardware and software synthesis. Vulcan consists of following sub-systems. For each of these sub-systems a command menu is presented that lists the commands relevant to the subsystem. A history stack maintains the subsystems being used by the user. This allows for excursions into different subsystems as needed (for example, model manipulations related to constraint analysis may use synthesis results instead of using estimation routines).

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 21 shows the command flow in generation of the software component. The software synthesis also performs generation of scheduling routines (enqueue, dequeue) and hardware-software interface routines (transfer_to) 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.

As mentioned earlier, there are several issues that must be resolved in order to bring the target architecture described in Section 1 closer to a realization. Among the important issues in its implementation are use

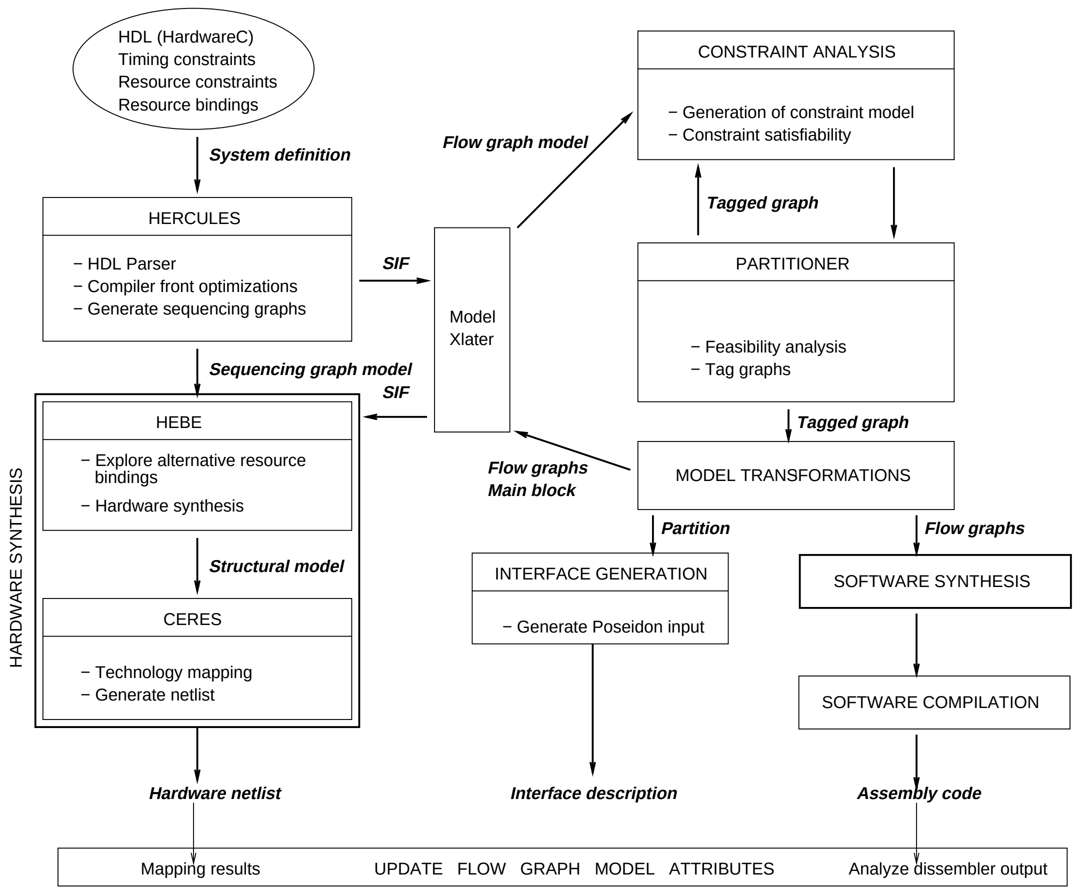
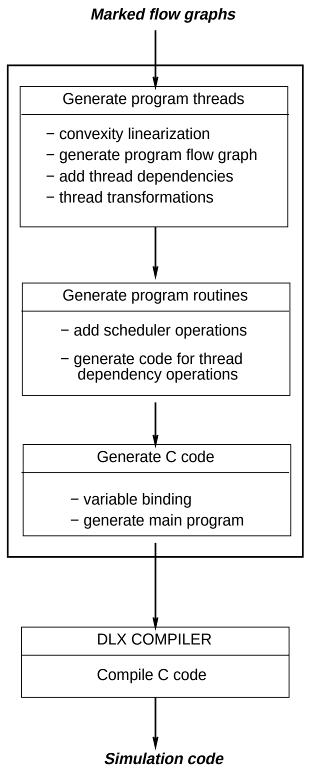


Figure 20: Vulcan subsystems and the Olympus Synthesis System



Marked flow graphs
Generate program threads
− convexity linearization
− generate program flow graph
− add thread dependencies
− thread transformations
Generate program routines
− add scheduler operations
− generate code for thread dependency operations
Generate C code
− variable binding
− generate main program
DLX COMPILER
Compile C code
Simulation code

Figure 21: Flow of software synthesis in Vulcan

of communication and synchronization mechanisms and the architecture of the interface between hardware and software components. We assume that the communication across hardware-software is carried out over a communication bus. We further assume that only the processor is the bus master, thus obviating a need for implementation of bus arbitration logic in the dedicated hardware.

We note that the target architecture presented in Section 1 is simple and leaves open many different possible ways of implementing the hardware-software interface and communication mechanisms. In the following, we present the architectural choices made by Vulcan and possible extensions and alternatives.

6.1 System synchronization

System synchronization refers to mechanism for achieving synchronization between concurrently operation hardware and software components. Due to pseudo-concurrency in the software component, that is, concurrency simulated by means of operation interleaving, a data transfer from hardware to software must be explicitly synchronized. Using a *polling* strategy, the software component can be designed to perform *pre-meditated transfers* from the hardware components based on its data requirements. This requires static scheduling of the hardware component so that the software is able to receive the data when it needs it. In cases where the software functionality is communication limited, that is, the processor is busy-waiting for an input-output operation most of the time, such a scheme would be sufficient. Further, in the absence of any $\mathcal{ND}$ operations, the software component in this scheme can be simplified to a single program thread and a single data channel since all data transfers are serialized. However, this would not support any branching nor reordering of data arrivals since dynamic scheduling of operations in hardware would not be supported.

In order to accommodate different rates of execution of the hardware and the software components, and due to $\mathcal{ND}$ operations, we look for a *dynamic* scheduling of different threads of execution. Such a scheduling is done based on the availability of data. This scheduling is by means of a *control FIFO* structure which attempts to enforce the policy that the data items are consumed in the order in which they are produced. The hardware-software interface consists of data queues on each channel and a FIFO that holds the identifiers

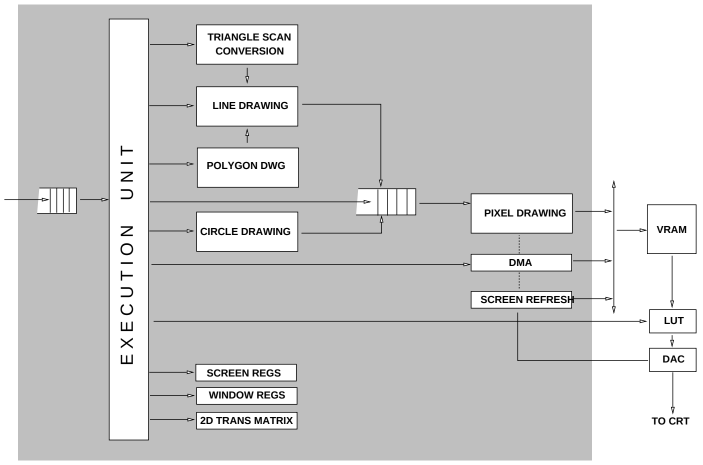


Figure 22: Example design block diagram.

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 would be the maximum number of threads in the system. Example 6.2 below shows an example of the interface between hardware and software based on two threads of execution.

Example 6.2. Specification of the control FIFO based on two threads of execution.

```
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;
```

In this example, two data queues with 16 bits of width and 1 bit of depth, `line_queue` and `circle_queue`, and one queue with 2 bits of width and 1 bit of depth `controlFIFO` are declared. The guarded commands specify the conditions on which the number 1 or the number 2 are enqueued – here, a '+' after a signal name means a positive edge and a '-' after the signal means a negative edge. The first 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`. □

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. The protocol governing the enqueue and dequeue operations to the control FIFO are described using guarded commands in a interface description file that is input to the system simulator [45].

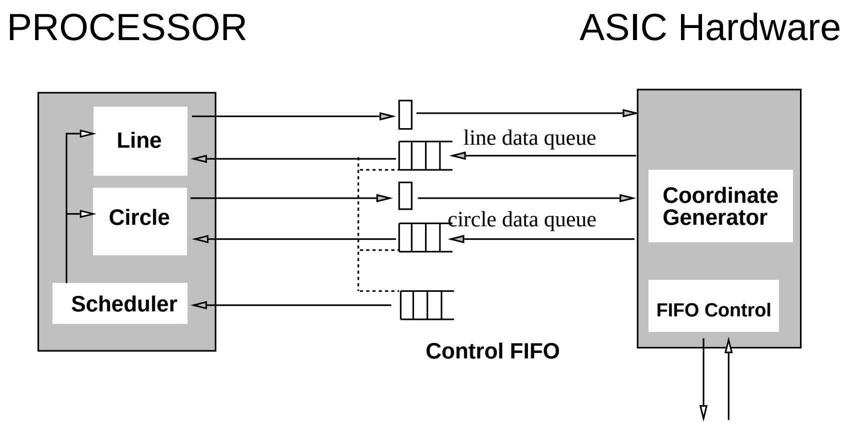


Figure 23: Controller implementation.

6.2 Example

Figure 22 shows block diagram of a graphics controller. The controller outputs pixel coordinates for specified geometries. The input to the controller is a specification of the geometry and its parameters, such as end points of a line. The current design handles drawing of lines and circles. However, it is a modular design where additional drawing capabilities can be added. 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. Typically the path from drawing buffer to the device runs at a rate of about 40 million samples per second.

As shown in Figure 23, a mixed implementation of the system design consists of line and circle drawing routines in the software 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 in addition to the runtime scheduler (or the main program). Both program threads generate coordinates that are used by the dedicated hardware. The data-driven dynamic scheduling of program threads is achieved by a 3-deep control FIFO buffer as a part of the interface synchronization block. Table 2 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.

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

Table 2: Graphics controller implementations.

7 Summary and Future Work

We have presented a co-synthesis approach to achieve computer-aided design of embedded systems. This methodology is driven by analysis of constraints on cost and 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. The graph model is devised to support implementations of graph models that execute at very different speeds by means of message-passing based communications between models. At the same time, the operations within a graph model communicate by means of shared memory, providing a way for efficient individual hardware or software implementations. Through this dichotomy of communication implementation, a hardware-software system is described at the level of individual graphs as being implemented in either hardware or software.

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 by means of 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 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 research in hardware-software co-synthesis 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 and synchronization. This work makes an attempt at developing the various sub-problems that are solved in an effort to develop an effective and practical co-synthesis approach. 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. As a result, we are able to put together a complete co-synthesis solution for embedded system designs that are modeled using a hardware description language and demonstrate the feasibility of achieving co-synthesis.

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. In particular, there is a need for supporting multiple levels of memory and multiple processors for use in an application. We plan to address performance analysis for these generalizations to architecture. 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.

8 Acknowledgements

This research was sponsored by a grant from the AT&T Foundation and a grant from NSF-ARPA No. MIP 9115432.

References

[1] K. P. Juliussen and E. Juliussen, *The* 6^th *Annual Computer Industry Almanac 1993.* The Reference Press, Austin, TX, 1993.

[2] M. L. Bader, "Market survey," *Bader Associates, Mountain View, California*, 1993.

[3] W. Wolf, "Hardware-Software Co-design of Embedded Systems," *IEEE Proceedings*, vol. 82, no. 7, pp. 965–989, July 1994.

[4] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, *Specification and Design of Embedded Systems.* Prentice Hall, 1994.

[5] D. Harel, "Biting the silver bullet," *IEEE Computer*, pp. 8–20, Jan. 1992.

[6] N. Halbwachs, *Synchronous programming of reactive systems.* Kluwer Academic Publishers, 1993.

[7] D. Scholefield, "The Formal Development of Real-Time Systems," technical report, (ch. 4), University of York, York, Heslington, Feb. 1992.

[8] A. Burns and A. Wellings, *Real-Time Systems and Their Programming Languages.* Addison-Wesley, 1990.

[9] D. Ku and G. D. Micheli, *High-level Synthesis of ASICs under Timing and and Synchronization Constraints.* Kluwer Academic Publishers, 1992.

[10] V. Cerf, *Multiprocessors, Semaphores and a Graph Model of Computation.* PhD thesis, UCLA, Apr. 1972.

[11] R. Jain, M. J. Mlinar, and A. Parker, "Area-time model for synthesis of non-pipelined designs," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, pp. 48–51, Nov. 1989.

[12] F. J. Kurdahi and C. Ramachandran, "Evaluating layout area tradeoffs for high level applications," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 1, no. 1, pp. 46–55, Mar. 1993.

[13] D. Bustard, J. Elder, and J. Welsh, *Concurrent Program Structures*, p. 3. Prentice Hall, 1988.

[14] 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.

[15] J. F. Allen, "Maintaining knowledge about temporal intervals," *Communications ACM*, vol. 26, no. 11, pp. 832–843, Nov. 1983.

[16] 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.

[17] D. Ku and G. D. Micheli, "Relative Scheduling under Timing Constraints," in *Proceedings of the* 27^{th}*Design Automation Conference*, (Orlando), pp. 59–64, June 1990.

[18] R. K. Gupta, *Co-synthesis of Hardware and Software for Digital Embedded Systems.* PhD thesis, Stanford University, Dec. 1993.

[19] T. Amon and G. Borriello, "Sizing Synchronization Queues: A Case Study in Higher Level Synthesis," in *Proceedings of the* 28^{th}*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 *Proceedings of the IEEE International Conference on Computer-Aided Design*, pp. 660–664, Nov. 1993.

[21] D. R. Cox and W. L. Smith, *Queues.* John Wiley and Sons, 1961.

[22] A. Heck, *Introduction to Maple.* Springer-Verlag, 1993.

[23] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

[24] J. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance Evaluation*, vol. 2, 1982.

[25] A. Mok, *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment.* PhD thesis, M.I.T., 1983.

[26] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[27] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools.* Addison Wesley, 1986.

[28] A. Shaw, "Reasoning about Time in Higher Level Language Software," *IEEE Trans. Software Engg.*, vol. 15, no. 7, pp. 875–889, July 1989.

[29] A. Mok and et. al., "Evaluating Tight Execution Time Bounds of Programs by Annotations," in *Proceedings of the Sixth IEEE Workshop Real-Time Operating Systems and Software*, pp. 74–80, May 1989.

[30] C. Y. Park and A. C. Shaw, "Experiments with a Program Timing Tool Based on Source-Level Timing Schema," in *Proceedings of the* 11^{th}*IEEE Real-Time Systems Symposium*, pp. 72–81, Dec. 1990.

[31] W. Hardt and R. Camposano, "Trade-offfs in HW/SW Codesign," in *International Workshop on Hardware-Software Co-design*, Oct. 1993.

[32] G. J. Chaitin, "Register Allocation and Spilling via Graph Coloring," *SIGPLAN Notices*, vol. 17, no. 6, pp. 201–207, 1982.

[33] D. Filo, D. C. Ku, and G. De Micheli, "Optimizing the control-unit through the resynchronization of operations," *INTEGRATION, the VLSI Journal*, vol. 13, pp. 231–258, 1992.

[34] P. Chou and G. Borriello, "Software scheduling in the Co-Synthesis of Reactive Real-Time Systems," in *Proceedings of the Design Automation Conference*, June 1994.

[35] Y. Liao and C. Wong, "An algorithm to compact a VLSI symbolic layout with mixed constraints," *Proceedings of the IEEE Transactions on CAD/ICAS*, vol. 2, no. 2, pp. 62–69, Apr. 1983.

[36] 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.

[37] 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.

[38] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Technical Journal*, vol. 49, pp. 291–307, Feb. 1970.

[39] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proceedings of the Design Automation Conference*, pp. 175–181, 1982.

[40] V. Sarkar, *Partitioning and scheduling parallel programs for multiprocessors.* MIT Press, Cambridge, Mass., 1989.

[41] R. Gupta and G. D. Micheli, "Partitioning of Functional Models of Synchronous Digital Systems," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, (Santa Clara), pp. 216–219, Nov. 1990.

[42] 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.

[43] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach.* Morgan-Kaufman, 1990.

[44] D. Ku and G. D. Micheli, "High-level Synthesis and Optimization Strategies in Hercules and Hebe," in *Proceedings of the European ASIC Conference*, (Paris, France), pp. 111–120, May 1990.

[45] R. K. Gupta, C. Coelho, and G. D. Micheli, "Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components," in *Proceedings of the* 29^{th}*Design Automation Conference*, pp. 225–230, June 1992.

List of Figures