Constrained Software Generation for Hardware-Software Systems

Rajesh K. Gupta

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

Giovanni De Micheli

Departments of EECS
Stanford University
Stanford, CA 94305.

Abstract

Mixed systems are composed of interacting hardware components such as general-purpose processors, application-specific circuits and software components that executes on the general purpose hardware. The software component consists of application-specific routines that must deliver the required system functionality and a runtime environment. Further, the software is required to deliver functionality under constraints on timing and memory storage available. In this paper, we consider methods to achieve software generation under imposed constraints and demonstrate the utility of our approach by examples.[1]

1 Introduction

In recent years there has been a surge of interest in digital system implementations that use microprocessors, along with memory and logic chips [1, 2, 3, 4]. The primary motivation for using *predesigned* processors is to reduce the design time and cost by implementing functionality using a program on the processor. However, a purely software implementation often fails to meet required timing performance which may be on the time to perform a given task, and/or on the ability to sustain specified throughput at system ports. Therefore, dedicated hardware is often needed to implement time-constrained portions of system functionality [4].

This work considers a constraint-driven synthesis approach to explore mixed system designs. It is built upon high-level synthesis techniques for digital hardware [5, 6]. There are several sub-problems that have to be addressed in order to achieve the synthesis goal for mixed systems. For an exposition of the related problems the reader is referred to [4]. Here we focus on the problem of software generation under constraints, given that the portion of system functionality to be implemented in software has been identified.

We briefly describe the inputs followed by a cost model of the processor in Section 2. We then present an overview of the steps in generation of the software component. In Section 3 we present a model of software that ensures satisfaction of timing constraints while still allowing for a serial execution of all operations by suitably overlapping input/output operations to computations. Estimation of software performance in view of the cost model of the target processor is presented in Section 4. Software performance is affected by the allocation and management of storage for program and data portions of the software. We present a scheme for estimation of software storage and its effect on software performance. We conclude in Section 5 by presenting a summary of the contributions and open issues.

2 The Inputs and Synthesis Flow

The input to our co-synthesis system consists of a description of system functionality in a hardware description language (HDL) called *HardwareC* [7]. The choice of this language is due to practical reasons and other languages such as VHDL, Verilog can be used as well. (Though additional language semantic considerations for synthesis may be needed, see [8]). The input description is compiled into a flow graph model [9] that consists of a set Φ of acyclic polar graphs $G(V, E)$. The vertex set V represents *language-level* operations and special *link* vertices used to encapsulate hierarchy. The link vertices induce a (single or multiple) calls to an another flow graph model which may, for instance, be body of a loop operation. The unified semantics of loop and call operations makes it possible to perform efficient constraint analysis to ensure existence of a feasible hardware or software implementation [9]. The edge set E in flow graph represents dependencies between operation vertices.

Operations in a flow graph present either a fixed or variable delay during execution. The variation in delay is caused by dependence of operation delay on either the *value* of input data or on the *timing* of the input data. Example of operations with value-dependent delay are loop (link) operations with data-dependent iteration count. An operation presents a timing-dependent delay only if it has *blocking* semantics, for example, the

[1] Supported by a grant from the AT&T Foundation and a grant from NSF-ARPA No. MIP 9115432.

wait operation that models communication and synchronization events at system ports. Since loop (link) and synchronization operations introduce uncertainty over the precise delay and order of operations, these operations are termed as *non-deterministic* delay or $\mathcal{ND}$ operations.

The timing constraints bound either the time interval between initiation of any two operations (min/max delay constraints), or the successive initiation interval of the same operation (rate constraints). For constraint analysis purposes the rate constraints are translated into one or more min/max delay constraints by means of constraint propagation techniques [9]. The resulting min/max constraints are represented as additional weighted edges on the flow graph model. The flow graph model with constraint edges is called a *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 weight $\delta_{ij} \in \Delta$ on edge (v_i, v_j) defines constraint on the operation start times t_k as $t_k(v_i) + \delta_{ij} \leq t_k(v_j)$ for all invocations k.

2.1 Implementation Attributes

A hardware or software implementation of a flow graph G refers to assignment of delay and size properties to operations in G and a choice of a runtime scheduler Υ that enables execution of source operation in G. For non-pipelined hardware implementations, the runtime scheduler is trivial as the source operation is enabled once the sink operation completes. For software implementation the runtime scheduler may be more complex and is discussed further in next section.

The size attributes refer to the physical size and pinout of an implementation. 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. In general, it is a difficult problem to accurately estimate the size of an implementation from the graph models. Estimation in this context refers to *relative* sizes for implementations of different flow graphs, rather than an absolute prediction of the size of the resulting hardware or software.

Memory Side-effects: The operational semantics of the graph model requires use of an *internal storage* in order to facilitate multiple-assignments in HDL descriptions [10]. The resulting memory side-effects created by graph models are captured by a set $M(G)$ of variables that are defined and used by the operations in a graph model G. $M(G)$ is independent of the cycle-time of the clock used to implement the corresponding synchronous hardware and does not include storage specific to structural implementations of G (e. g., control latches). Further, $M(G)$ need not be the *minimum* storage required for correct behavioral interpretation of G.

The size $S(G)$ of a software implementation consists of the program size and the static storage to hold variable values across machine operations. The static data storage can be in the form of specific memory locations or on-chip registers. This static storage is, in general, upper bounded by the size of variables in $M(G)$ defined above. Estimation of software size requires, in addition to the flow graphs, knowledge of the processor and the runtime system to be used, as discussed in the next section. Pinout, $P(G)$ refers to the size of inputs and outputs in units of words or bits. A pinout does not necessarily imply the number of *ports* used. A port may be bound to multiple input/output operations in a flow graph.

The flow graph model is input to a set of partitioning transformations that generate set of flows graphs to be implemented in hardware and software [10]. The hardware implementation is carried out by high-level synthesis tools [5]. The objective of software implementation is to generate a sequence of processor instructions from the set of flow graph models. Due to significant differences in abstraction levels of the graph model and processor instructions, this task is performed in steps as shown in Figure 1. A detailed discussion of these steps can be found in [10]. We first create a linearized set of operations collected into program *threads*. The dependencies between program threads is built into the threads by means of additional enabling operations for *dependent* threads. Further, overhead operations are added to execute program threads either by means of subroutine calling, or as coroutine transfers. Finally, the routines are compiled into machine code. We assume that the processor is a predesigned general-purpose component with available compiler and assembler. Therefore, the important issue in software synthesis is generation of the source-level program. Most of this paper is devoted to this step of software synthesis. For details on assembly, linking and loading issues the reader is referred to [10]. For the choice of processor, we assume the DLX processor [11] for which simulation and compilation tools have been integrated into our synthesis system. However, the analysis routines can use any other processor abstracted by means of a cost model described next.

2.2 Processor Cost Model

To a compiler, 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* machine with only explicit operands in an instruction (i.e., there is no

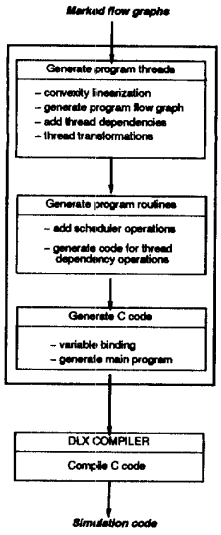


Figure 1: *Generation of the software component.*

accumulator or stack). We also assume that the memory model uses a ***byte-level addressing*** scheme. This is consistent with the prevailing practice in the organization of general-purpose computer systems.

A processor instruction consists of an ***assembly language*** operation and a set of operands on which to perform the specified operation. While the actual instruction sets for different processors are different, a commonality can be established based on the ***types*** of instructions supported. For our purposes we assume a ***basic*** set of instructions consisting of memory operations (load, store), ALU operations and control transfer operations. It is important to note that some of these instructions may refer to *macro*-operations that may not be available as single machine instructions but as a group of instructions, for example, call and return. These (macro-)assembly instructions are often needed for compilation efficiency and to preserve the ***atomicity*** of certain operation in the flow graph. These operations also help in software delay estimation by providing additional information which may not be readily available purely from looking at the instruction set of a processor.

Based on this understanding of processor and instruction set architecture we represent the target processor as, $\Pi = (\tau_{op}, \tau_{ea}, t_m, t_i)$ where the execution time function, τ_{op}, maps ***assembly operations*** to positive integer delays. The address calculation function, τ_{ea}, maps a memory addressing mode to the delay (in cycles) encountered by the processor in computing the effective address. When generating programs from HDL descriptions only a limited number of addressing modes are used. For example, a computed reference (register indirect) usually occurs when the data value is created dynamically or a local variable (stack) is referred to by means of a pointer or an array index. These conditions can generally be avoided when generating code from HDL. t_m represents memory access time. The interrupt response time, t_i, is the time that processor takes to become aware of an external hardware interrupt in a single interrupt system (that is, when there is no other maskable interrupt running).

Storage alignment is a side-effect of the byte-level addressing scheme assumed for the processor/memory architecture. Because the smallest object of a memory reference is a byte, references to objects smaller than a byte must be aligned to a byte. Further, for memory efficiency reasons, the objects that occupy more than a byte of storage are assigned an integral number of bytes, which means their addresses must also be aligned. For example, address of a 4-byte object (say integer) must be divisible by 4. In case of a ***structure*** declaration, the size is determined by the total of size requirements of its members. In addition, the structure must ***end*** at an address determined by the most restrictive alignment requirement of its members. This may result in extra storage for 'padding'. Variables with size greater than 32-bits, are bound to multiple variables represented by an array. The size and alignment requirements are then multiplied by the number of array elements.

Example 2.1. Variable storage assignments.

The following shows the set of variables used in the definition of a flow graph and the corresponding storage assignments in the software implementation of the graph (as generated by VULCAN).

```
a[1], b[2], c[3], d[4], e[5]        struct{ a:1; b:2; c:3; d:4; f:5 }
f[33]                               int f[2]
```

Minimum storage used in the flow graph model is 8 bytes. However, due to alignment requirements the actual data storage is 12 bytes.□

3 Software & Runtime System Model

Most synthesized hardware uses static resource allocation and binding schemes, and a static or relative scheduling technique [7]. 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. Consequently, there is no need for a runtime system in hardware. Similarly, in software, the need for a runtime system depends upon the whether the resources and tasks (and their dependencies) are determined at compile time or runtime.

Since our target architecture contains only a single resource, namely, 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

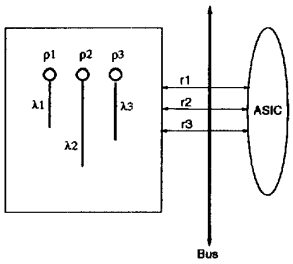


Figure 2: *Software model.*

resources and tasks and their dependencies are all statically known. However, in presence of $\mathcal{ND}$ operations in software, a complete serialization of operations may make determination of constraint satisfiability impossible [4]. A solution to this problem is to construct software as a set of ***concurrent program threads*** as sketched in Figure 2. 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 of a thread is defined as sum of the delay of its operations without including the initial $\mathcal{ND}$ operation whose delay is accounted for in the delay due to the runtime scheduler, Υ.

In this model of software, satisfiability of constraints on operations belonging to different threads can checked for marginal or deterministic satisfiability [9], assuming a bounded delay for runtime scheduling associated with $\mathcal{ND}$ operations. Constraint analysis for software depends upon first arriving at an estimate of the software performance discussed next.

4 Estimation of Software Performance

When deriving timing properties from programs, several problems are encountered due to the fact that popular programming languages provide an inherently asynchronous description of functionality, where the program output is independent of the timing behavior of its components and of its environment. Attempts have been made to annotate programs with relevant timing properties [12, 13]. Syntax-directed delay estimation techniques have been tried [14, 15] 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.

We perform delay estimation on flow graph models for both hardware and software using bottom-up estimation. A software delay consists of two components: delay due operations in the flow graph model, and delay due to the runtime environment. The effect of runtime environment on constraint satisfiability is evaluated in terms of operation *schedulability* for a given type of runtime environment such as preemptive and prioritized [10] and is out of the scope of this paper. It suffices to say that the effect of runtime can be modeled as a constant overhead delay to each execution of the flow graph. Here we focus on the first component, that is, the delay of a software implementation of the operations in the flow graph model. For this purpose, it is assumed that a given flow graph is to be implemented as a single program thread. Multiple program thread generation is achieved similarly by first identifying subgraphs corresponding to program threads [10]. Software delay then depends upon the delay of operations in the flow graph ***and*** operations related to storage management. Calculations of storage management operations is described in Section 4.4.

4.1 Software Implementation Delay

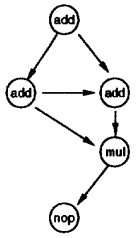
Each operation v in the flow graph is characterized by a number of read accesses, $m_r(v)$, a number of write accesses, $m_w(v)$ and a number of assembly-level operations, $n_o(v)$. The software operation delay function, η, is computed as:

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

where the operand access time, m_i, is the sum of effective address computation time and memory access time for memory operands. Note that often not all possible combinations of ALU and memory operations are allowed and often the two operations are overlapped for some instructions. Due to this non-orthogonality, the delay function may overestimate the operation delays.

Example 4.1. Software delay estimation.

For the graph model shown below, assuming addition delay 1 cycle, multiplication delay is 5 cycles and memory delay 3 cycles.



Assuming that each non-NOP operation produces a data, that is, $m_w(v) = 1$ and that the number of memory read operations is given by the number of input edges, the software delay associated with the graph model is $3 \times t_+ + t_* + (5+4) \times m_i = 35$ cycles. By comparison, the VULCAN generated code, when assembled, takes 38 cycles (not including setup operations that are accounted for later) that includes an additional 3 cycle delay due to return conventions.□

Use of operation fanin and fanout to determine memory operations provides an approximation for processors with limited number of general purpose registers. Most processors with load-store instruction set architectures feature a large number of on-chip registers. We consider the effect of register usage in these processors in Section 4.4.

$\mathcal{ND}$ operations

Wait operations in a graph model induce a synchronization operation in the corresponding software model. This delay is characterized by a *synchronization overhead* related to the program implementation scheme used. One implementation of a wait operation is to cause a *context switch* in which the waiting thread is switched out in favor of an enabled thread. We assume that the software is computation intensive and thus the wait time of a program thread can always be overlapped by the execution another program thread. A thread is resumed by the runtime scheduler using a specific scheduling policy. Alternatively, the completion of synchronization operation associated with wait operation can also be indicated by means of an interrupt to the processor. In this case, the synchronization delay is estimated as $\eta_{intr}(v) = t_i + t_s + t_o$, where t_i is interrupt response time, t_s is interrupt service time, which is typically the delay of the service routine that performs input read operation and t_o is concurrency overhead which constitutes a 19 cycle delay for the simplified coroutine implementation on the DLX processor [10].

Finally, the link operation are implemented as call operations to separate program threads corresponding to bodies of the called flow graphs. Thus the delay due to these operations is accounted for as the delay in implementation of control dependencies between program threads.

4.2 Estimation of Software Size

The size of a software implementation S^{Π} refers to the size of program S_p^{Π} and of static data S_d^{Π} necessary to implement the corresponding program on a given processor Π. For a system model Φ,

$$S^{\Pi}(\Phi) = \sum_{G_i \in \Phi} S^{\Pi}(G_i) = \sum_{G_i \in \Phi} [S_p^{\Pi}(G_i) + S_d^{\Pi}(G_i)] \quad (2)$$

We postpone the discussion on estimation of program size to later in this section. S_d^{Π} represents the storage required to hold variable values across operations in the flow graph and across the machine operations. 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,

$$S_d^{\Pi}(G) \leq |M(G)| + |P(G)| \quad (3)$$

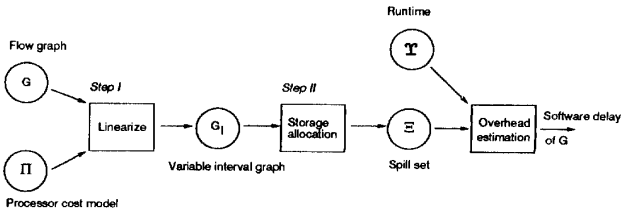


Figure 3: *Software delay estimation flow.*

This inequality is because not all variables need be *live* at the execution time of an instruction. A variable is considered live if it is input to an future instruction (Section 4.4). In case, $S_d^{\Pi}(G)$ is a proper subset of the variables used in software implementation of G, that is, $M(G)$, *additional* operations (other than the operation vertices in G) are needed to perform data transfer between variables and their mappings into $S_d^{\Pi}(G)$. In case, $S_d^{\Pi}(G)$ 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 $V(G)$ is constant under any schedule. However, the data set size $S_d^{\Pi}(G)$ 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 edges induce a (conflict) relation between simultaneously live variables. It has been shown that this problem is NP-complete for general graphs [16]. Most popular heuristics for code generation use a specific order of execution of successor nodes (e.g., left-neighbour first) in order to reduce S_d^{Π} [17].

In contrast to the register assignment in software compilers which perform simultaneous register assignment and operation linearization, we use a two-step approach as shown in Figure 3. We first linearize operations followed by an estimation of the data transfer operations for a given linearization. This two-step approach is taken in view of the timing constraints not present in traditional software compilers.

4.3 Operation Linearization

In the presence of timing constraints, operation linearization can be reduced to the problem of 'sequencing of variable length tasks with release times and deadlines' which is shown to be NP-complete in the strong sense [16]. It is also possible that there exists no valid linearization of operations. An exact ordering scheme

is described in [7] that explores all possible orders in order to find an optimum linearization. In [18], 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 (root) vertex and outputs it. The input to the algorithm is a constraint graph model. The linearization algorithm uses a data structure, Q, in which candidate vertices for elimination at any step are stored. The algorithm consists of following three steps:

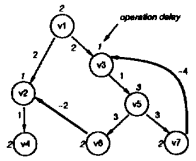
I. Select a root operation to add to the linearization,
II. Perform timing constraint analysis to determine if the addition of the selected root operation to the linearization constructed thus far leads to a feasible order, else select another vertex from Q,
III. Eliminate selected vertex. Update Q.

The (heuristic) selection of a vertex to be output from among a number of zero in-degree vertices 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, that the selected vertex does not create any additional dependencies or increase weights on any of the existing dependencies in the constraint graph. For two, we associate a measure of *urgency* with each root 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., schedule with a single resource) is that the set of operations have a schedule under timing constraints *assuming unlimited resources*. A feasible schedule using unlimited resources 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, the graph may contain cycles due to the backward edges created by the timing constraints. A feasible schedule under timing constraints is obtained by using the slacks to adjust the path delays to operations such that all constraints are satisfied. This is accomplished by applying an iterative algorithm based on [19] 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. If this procedure fails, 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.

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 is necessary but not sufficient, therefore, the heuristic may fail to find any feasible linearization when there may exist a valid ordering.

Example 4.2. Operation linearization.



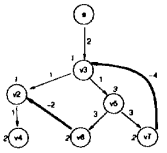
Consider the flow graph shown in the figure above. Initialize: $Q = \{v_1\}$, $\delta(s) = 0$. By applying the cycle detection procedure on this graph, the assignment of operation urgency labels, σ is (0 4 2 5 3 6 6) for vertices v_1 through v_7.

Iteration 1:

▷I: $v = v_1$. Add edge (s, v_1) with weight = 0.
▷II: No positive cycle. Feasible.
▷III: $\boxed{\textbf{output} = v_1}$. $Q = \{v_2, v_3\}$. $\delta(s) = 0 + \delta(v_1) = 2$. Q is sorted based on σ to be $Q = \{v_3, v_2\}$ where the first element represents the head of Q.

Iteration 2:

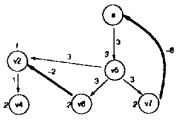
▷I: Candidate $v = v_3$. The new constraint graph G'_T is shown below:



▷II: No positive cycle. Feasible. The new assignment of the urgency labels is same as the previous one.
▷III: $(v_7, v_3) \in E_b \Rightarrow \delta(v_7, v_3) = -6$. $Q = \{v_2, v_5\}$. $\boxed{\textbf{output} = v_3}$. $\delta(s) = 2 + \delta(v_3) = 3$. Urgency, $\sigma(v_2) = 4, \sigma(v_5) = 3$. Q is sorted as $Q = \{v_5, v_2\}$.

Iteration 3:

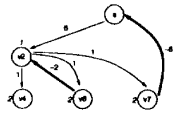
▷I: Candidate $v = v_5$. Add edge (s, v_5) with weight $= \delta(s) = 3$.Constraint graph G'_T is shown below.



▷II: No positive cycle. Feasible. $\sigma(s) = 0, \sigma(v_2) = 6, \sigma(v_4) = 7, \sigma(v_5) = 3, \sigma(v_6) = \sigma(v_7) = 6$.
▷III: $Q = \{v_2, v_6, v_7\}$. $\boxed{\textbf{output} = v_5}$. $\delta(s) = 3 + \delta(v_5) = 6$. $\sigma(v_2) = \sigma(v_6) = \sigma(v_7) = 6$.

Iteration 4:

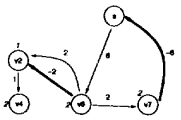
▷I: Candidate $v = v_2$. Add edge (s, v_2) with weight $= \delta(s) = 6$. Constraint graph G'_T is shown below:



▷II: Positive cycle. Mark and move v_2 to tail of Q. $Q = \{v_6, v_7, v_2\}$.

Iteration 5:

▷I: Candidate $v = v_6$. G'_T is shown below:



▷II: Linearization using v_6 at this step leads to a positive cycle. Hence v_6 is moved to the tail of Q. $Q = \{v_7, v_2, v_6\}$.

Iteration = 6:

▷I: Candidate $v = v_7$. The constraint graph in successive iterations are shown in Figure 4.
▷II: No positive cycle. Feasible. The assignment of the urgency labels: $\sigma(s) = 0, \sigma(v_2) = 8, \sigma(v_4) = 9, \sigma(v_6) = 8, \sigma(v_7) = 6$.
▷III: $Q = \{v_2, v_6\}$. **output = v_7**. $\delta(s) = 6+2 = 8$. Urgency, $\sigma(v_2) = \sigma(v_6) = 8$.

Iteration = 7:

▷I: Candidate $v = v_2$.
▷II: no positive cycles. Feasible. The assignment of urgency labels: $\sigma(s) = 0, \sigma(v_2) = 8, \sigma(v_4) = 9, \sigma(v_6) = 9$.
▷III: Since $(v_6, v_2) \in E_b \Rightarrow \delta(v_6, v_2) = -2-8 = -10$. **output = v_2**. $Q = \{v_6, v_4\}$.

Iteration = 8:

▷I: Candidate $v = v_6$.
▷II: no positive cycles. Feasible. $\sigma(s) = 0, \sigma(v_6) = 10, \sigma(v_4) = 12$.
▷III: $Q = \{v_4\}$, $\delta(s) = 9 + 2 = 11$. **output = v_6**.

Iteration = 9:

▷I: Candidate $v = v_4$.
▷II: no positive cycles. Feasible.
▷III: $Q = \emptyset$. **output = v_4**.

Thus, the linearization returned by the algorithm is $v_1, v_3, v_5, v_7, v_2, v_6, v_4$. □

The time complexity of linearization algorithm is dominated by the cycle detection algorithm and is given by $O(|V|^3 \cdot k)$ where k is the number of backward edges and typically a small number.

4.4 Estimation of Memory Operations

Memory operation in a software implementation is related to the amount and allocation of static storage $S_d^{II}(G)$. Since it is difficult to determine actual register allocation and usage, some estimation rules must

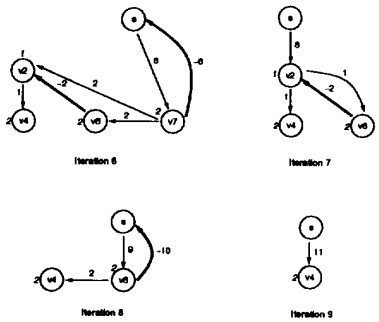


Figure 4: *Linearization example.*

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. 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 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 registers available (not including any registers needed for operand storage). In case R is small, 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. The actual number of spills can be determined exactly given a schedule of *machine* 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, $m_r = |E|$ and $m_w = |V|$. Note that each register read results in a memory read operation and each register write results in a memory write operation, $(r_r = m_r)$ and $(r_w = m_w)$.

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

Case III: $R < R_l$ At some operation v_i there will not be a register available to write the output of v_i. This implies that some register holding the out-

put 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 live. Let $\Xi \subset V$ be the set of live operations that are chosen for spill. Therefore,

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

Clearly, the choice of the spill set Ξ determines the actual number of memory read and write operations needed. In software compilation, the optimization problem is then to choose Ξ such that $\sum_{\Xi} o(v)$ is minimized. This is another way of stating the familiar register allocation problem [20]. We use the following heuristic to select operations for the spill. From the conflict graph, G_I for a given schedule, select a vertex v with outdegree less than R. This vertex is then assigned a register different from its neighbours. From this we construct a new conflict graph G'_I by removing v and its fanout edges from G_I. The procedure is then repeated on G'_I until we have a vertex with outdegree greater than or equal to R. In this case, a vertex is chosen for spill and the process is continued.

For most assignment statements, the left side generates a *lvalue* and the right side generates a *rvalue* [17], though in some cases (e.g., computed references and structures) left hand side also generates rvalues which are subsequently assigned to appropriate lvalue also generated by the left side. We observe that the number of instructions generated by the compiler is related to the number of rvalues. Therefore, the program size S_p^{II} is approximated by the sum over rvalues associated with operation vertices. This is only an upper bound since some global optimizations may reduce the total number of instructions generated.

Note that we do not directly try to minimize register usage by the object program since that optimization level belongs to the software compiler. The objective of spill set enumeration is to arrive at an estimate of memory operations assuming a reasonably sophisticated software compiler. Clearly this is only an estimate since the actual register allocation will vary with the choice of the compiler.

5 Summary

This paper presents important issues in software generation from our experience in building a co-synthesis system, VULCAN. The actual code generation is a fairly straightforward procedure and is omitted from discussions here. The main problem is due to presence of $\mathcal{ND}$ operations that may make it impossible to guarantee satisfaction of timing constraints. We have presented a model for the software and the runtime system that consists of a set of program threads which are initiated by synchronization operations. We then describe a linearization algorithm using a vertex elimination scheme. This algorithm uses a heuristic measure of urgency based on timing constraint analysis on an implementation *using unlimited resources.*

We conclude by noting that there exists a tradeoff between code and data size for a given graph implementation into software based on how the variables in $M(G)$ are *packed* into machine words (see Example 2.1). We leave this issue for a future discussion.

References

[1] D. E. Thomas *et al.*, "A Model and Methodology for Hardware-Software Codesign," *IEEE Design & Test of Computers*, pp. 6–15, Sept. 1993.

[2] M. Chiodo *et al.*, "Synthesis of mixed software-hardware implementations from CFSM specifications," in *Intl. Wkshp. on Hardware-Software Co-design*, Oct. 1993.

[3] R. Ernst *et al.*, "Hardware-Software Cosynthesis for Microcontrollers," *IEEE Design & Test of Computers*, pp. 64–75, Dec. 1993.

[4] R. K. Gupta, G. D. Micheli, "Hardware-Software Cosynthesis for Digital Systems," *IEEE Design & Test of Computers*, pp. 29–41, Sept. 1993.

[5] G. D. Micheli *et al.*, "The Olympus Synthesis System for Digital Design," *IEEE Design & Test of Computers*, pp. 37–53, Oct. 1990.

[6] A. Jerraya *et al.*, "Linking system design tools and hardware design tools," in *CHDL'93*, Apr. 1993.

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

[8] W. Wolf, R. Manno, "High-level modeling and synthesis of communicating processes using VHDL," *IEICE Trans. Information & Systems*, E76-D(9), pp. 1039–1046, Sept. 1993.

[9] R. K. Gupta, G. D. Micheli, "Specification and analysis of constraints for hardware-software systems," *To appear.*, 1994.

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

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

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

[13] A. Mok *et al.*, "Evaluating Tight Execution Time Bounds of Programs by Annotations," in *Proc. IEEE Wkshp. Real-Time Operating Systems & Software*, pp. 74–80, May 1989.

[14] C. Y. Park, A. C. Shaw, "Experiments with a Program Timing Tool Based on Source-Level Timing Schema," in *Proc. Real-Time Systems Symposium*, pp. 72–81, Dec. 1990.

[15] W. Hardt, R. Camposano, "Trade-offfs in HW/SW Codesign," in *Intl. Wkshp. on Hardware-Software Co-design*, Oct. 1993.

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

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

[18] P. Chou, G. Borriello, "Software scheduling in the Co-Synthesis of Reactive Real-Time Systems," in *Intl. Wkshp. on Hardware-Software Co-design*, Oct. 1993.

[19] Y. Liao, C. Wong, "An algorithm to compact a VLSI symbolic layout with mixed constraints," *Proc. IEEE Trans. on CAD/ICAS*, vol. 2, no. 2, pp. 62–69, Apr. 1983.

[20] G. J. Chaitin, "Register Allocation and Spilling via Graph Coloring," *SIGPLAN Notices*, 17(6), pp. 201–207, 1982.