ELSEVIER

Journal of Systems Architecture 43 (1997) 557–586

JOURNAL OF SYSTEMS ARCHITECTURE

Constrained software synthesis for embedded applications

Rajesh K. Gupta [a,*], Giovanni De Micheli [b]

[a] *Information and Computer Science, University of California, Irvine, CA 92697, USA*
[b] *Department of Electrical Engineering, Stanford University, Stanford, CA 94305, USA*

Received 1 July 1996; revised 2 October 1996; accepted 7 November 1996

Abstract

Embedded systems are composed of interacting hardware components such as general-purpose processors and application-specific circuits and software components that execute on the general-purpose hardware. The software component consists of application-specific routines that must deliver the required system functionality under constraints on timing and memory storage available. In this paper, we consider two main problems in the synthesis of the software component in embedded system designs: (a) generation of software and (b) conditions to ensure correct behavior of the generated software from an HDL-modeled input. Generation of embedded software requires operation linearization under constraints to ensure timely interaction with concurrent hardware. We describe our procedure to achieve constrained software generation and the utility of our approach by examples. Experimental results show that the proposed algorithm is substantially faster than conventional methods and yields efficient schedules for the embedded software.

Keywords: Software synthesis; Embedded software; Multithreaded software; Operation linearization; Operation serializability

1. Introduction

Embedded systems are targeted for specific applications under constraints on the relative timing of their actions. In recent years there has been a surge in embedded system designs that use commodity processors along with memory and logic chips [1–5]. The primary motivation for using predesigned processors is to reduce the design cost and time by using software to implement system functionalities. However, a purely software implementation often fails to meet the required timing performance for embedded systems. Thus, dedicated hardware is needed to implement the time-constrained portions of system functionality.

While there have been several advances in the design of individual hardware and software components, the combined design of systems using these components to achieve a specific functionality is still pretty much a manual and time-consuming task. This work considers a synthesis approach to systematic

* Corresponding author. Email: rgupta@ics.uci.edu.

PII S1383-7621(96)00125-7

exploration of mixed system designs that is driven by the constraints. It is built on the high-level synthesis techniques for digital hardware [6,7] and the program compilation techniques. An overview of the sub-problems and our approach can be found in [5]. In this paper, we focus on the problem of software synthesis under constraints in a co-synthesis framework.

This paper is organized as follows. We briefly define the input description and target architecture followed by a cost model of the processor in Section 2. We then review the steps used in software synthesis. In Section 3, we present our model of software that is useful in determination of satisfaction of the timing constraints. In Section 4, we present an algorithm for software linearization under timing constraints. We address the issue of correctness in software generation in Section 5. In Section 6, we present results from the implementation of our software synthesis procedure. We summarize contributions and discuss open issues in Section 7.

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* [8]. Our choice of an HDL for system description is due to its ability to specify detailed timing relationship between concurrent sets of operations. In addition, the simpler programming model using only static data types makes it possible to do complete operation dependency analysis that is essential to synthesis tasks. This input description is compiled into a flow graph model [9] consisting of a set Φ of acyclic polar graphs $G(V, E, \chi)$ with unique source and sink vertices. The vertex set V represents *language-level* operations and special *link* vertices are used to encapsulate hierarchy. A link vertex induces (single or multiple) calls to another flow graph model that may, for instance, be body of a loop operation. The edge set E in flow graph represents dependencies between the operation vertices. Function χ associates a Boolean (enabling) expression with every edge. The enabling expression for a condition vertex refers to the condition under which its successor nodes are enabled. In case of a multiple in-degree vertex, the vertex is enabled by evaluating an input expression consisting of logical AND and OR operations over the enabling expression of its fanin edges. We consider here only *well-formed* flow graphs where the enabling expressions use either AND or OR operations, but not both, in the same expression. Such a flow graph is a *bilogic* flow graph where a vertex may have conjoined or disjoined inputs/outputs. Bilogic flow graphs occur frequently in program control graphs [10]. A flow graph with only conjoined vertices is referred to as a *unilogic* flow graph.

Operations in a flow graph present either a fixed or variable delay during execution. The variation in delay is caused by the dependence of operation delay on either the *value* of the input data or on the *timing* of the input data. Example of operations with value-dependent delay are loop (link) operations with a data-dependent iteration count. In a unilogic flow graph only the link operations present value-dependent delays, whereas in a bilogic flow graph any operation can present a value-dependent delay. The second category of operations with variable delay are operations 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 with a block semantics is the *wait* operation that models synchronization events at system ports. The loop (link) and synchronization operations introduce uncertainty over the precise delay and order of operations. Due to concurrently executing flow graph models, these operations affect the order in which various operations are invoked. A system model containing such operations is called a *non-deterministic* model [11] and

operations with variable delays are termed as *non-deterministic* delay or $\mathcal{ND}$ operations.

We define a *rate of an execution* ρ_k at invocation k of an operation as the inverse of the time interval between its k and $(k-1)$-th execution. Thus, the rate of execution of an operation is defined only at times when the operation is executed. For a flow graph we define a *rate of reaction*, ϱ, as the rate of execution of its source operation. The reaction rate of a flow graph represents the frequency at which the graph is executed and is typically used to determine the throughput at its ports.

The timing constraints are of two types: (a) bounds on the time interval between initiation of any two operations (min/max delay constraints), and (b) bounds on the successive initiation interval of the same operation (rate constraints). For constraint analysis, the rate constraints are translated into one or more min/max delay constraints using 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 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.

It is important to note that while min/max timing constraints can be used to capture the *durational* and *deadline* timing constraints [12], [1] our constraint analysis procedure does not guarantee satisfaction of deadline constraints. Due to the non-deterministic delay operations, our timing constraint analysis only considers bounds relative operation intervals that are verified in view of the operation delay estimates. Also, in the presence of both minimum and maximum delay constraints between a pair of operations, the validity of constraint satisfiability analysis is limited by the accuracy of the delay estimation procedure.

2.1. Flow graph 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 $\mathcal{T}$ to enable the 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 is more complex and is discussed further in next section.

The size attributes refer to the physical size and pinout of implementation of operations and graphs. The hardware size, S, of an operation refers to its size as a sum of sizes of hardware resources required to implement the operation, the associated control logic and the storage registers. The size of a graph model is computed as a bottom-up sum of the size of its operations. 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. The size of a software implementation refers to the size of the program and the data portions.

In general, it is a difficult problem to accurately estimate the size of the resulting hardware or software from flow graph models, since the process of compilation and synthesis consists of optimizations at several levels of abstractions not captured by the flow graph model. The purpose of estimation in our

[1] For a given operation, v, release time, $r(v)$ is indicated by a min delay constraint of $r(v)$ from source to v. Similarly, a deadline of $d(v)$ is indicated by a max delay constraint of $d(v) - \delta(v)$ from v to source operation where $\delta(v)$ is the execution delay of the operation v.

context is to determine the relative sizes for different implementations of different flow graphs, to be used in making trade-offs between hardware and software implementations of a flow graph model. This allows for simplifications in the estimation procedure as discussed later.

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

2.2. Capturing the memory side-effects of a software implementation

A graph model captures the functionality of a system and the system behavior on its ports. The operational semantics of the graph model requires use of an *internal storage* to describe multiple assignments in the HDL model. Whereas additional variables can be created that avoid multiple assignments to the same variable, because of their structural nature ports must still be kept 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. Thus, during the execution of operations in a graph model, the internal storage may be modified more than once. 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 the 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.

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 a specific memory location or an on-chip register. This static storage is, in general, upper bounded by the size of the variables in $M(G)$ mentioned above. To estimate the software size, a flow graph model is not enough. In addition, knowledge of the processor to be used and the type of runtime system used is needed 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 required since a port may be bound to multiple input/output operations in a flow graph model.

2.2.1. Synthesis flow

The flow graph model is input to a set of partitioning transformations that generates a set of flows graphs to be implemented in hardware and software. The hardware implementation is carried out by high-level synthesis tools [6]. The objective of software implementation is to generate a sequence of processor or machine instructions from the set of flow graph models. Due to significant differences in processor abstractions at the levels of graph model and machine instructions, this task is performed in steps. The task of synthesis of software from flow graphs is divided into the following four steps as shown in Fig. 1. We first create a linearized set of operations collected into *program threads*. The dependencies between the program threads is built into the threads using (additional) enabling operations for

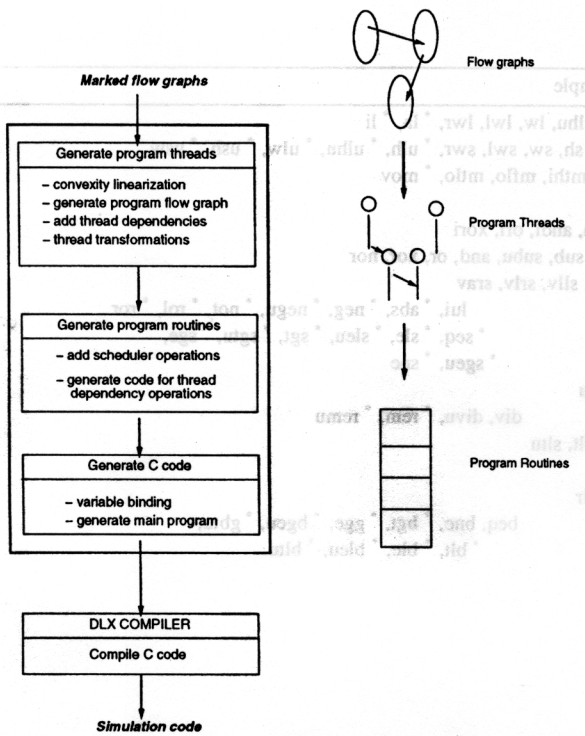


Fig. 1. Steps in generation of the software component.

dependent threads. Further overhead operations are added to implement the concurrency between the program threads either by subroutine calling, or as co-routines or as a program with case descriptions. Finally, the routines are compiled into machine code using compiler for the processor. 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 [13].

2.2.2. Target architecture

The target architecture for hardware–software implementation consists of a single processor assisted by application-specific chips and a single-level memory. The presence of a memory hierarchy leads to a significant variability in the timing properties of the program. An analysis of software performance in the presence of an instruction cache is presented in [14]. For the choice of a processor, we assume the DLX processor [15] for which simulation and compilation tools have been integrated into our synthesis system.

As mentioned earlier, an implementation of a flow graph in software is characterized by the assignment of delays to operation vertices and choice of a runtime scheduler. The delay of an operation is dependent on the set of processor instructions and delays associated with these instructions. The instruction set and associated delays are determined by the choice of the processor. We capture processor-specific information into a *cost model* described below. This model is general and provides flexibility to use processors other than the DLX.

2.3. Processor cost model

A processor can be described at different levels of detail such as the internal machine organization of blocks, or its instruction set architecture (ISA) which is commonly used by software compilers. We take the latter view for its ease and applicability in the context of our flow graph model even though it sacrifices accuracy in our estimations. Thus, a processor is characterized by its instruction set architecture which consists of its instructions and the memory model. We make the following assumptions on the ISA:

- The processor is a general-purpose *register* machine with only explicit operands in an instruction (no implied accumulator or stack). All operands must be named. This refers to the most general model for code generation. A general-purpose

Table 1
Basic instruction set

Instruction type	Meaning	DLX Example
load	Load from memory	lb, lbu, lh, lhu, lw, lwl, lwr, * la, * li
store	Store to memory	sb, sh, sw, swl, swr, * ulh, * ulhu, * ulw, * ush, * usw
move	Move registers	mfhi, mthi, mflo, mtlo, * mov
xchange	Exchange registers	–
alu	ALU operations	addi, addiu, andi, ori, xori
		add, addu, sub, subu, and, or, xor, nor
		sll, srl, sra, sllv, srlv, srav
		lui, * abs, * neg, * negu, * not, * rol, * ror
		* seq, * sle, * sleu, * sgt, * sgtu, * sge,
		* sgeu, * sne
mpy	Integer multiply	mult, multu
div	Integer divide	div, divu, * rem, * remu
comp	Compare	slti, sltiu, slt, sltu
call	Call	–
jump	Jump	j, jal, jr, jalr
branch	Branch	beq, bne, * bgt, * gge, * bgeu, * gbtu,
		* blt, * ble, * bleu, * bltu
bc_true	Branch taken	–
bc_false	Branch not taken	–
return	Call return	–
seti	Set interrupt	–
cli	Clear interrupt	–
int_response	Interrupt response	–
halt	Halt	–

* Synthesized/macro instruction.

register machine is the most dominant architecture in use today and is expected to remain so in foreseeable future [15]. [2]

- The memory addressing is based on *byte-level addressing*. 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. These instructions usually correspond to instructions in the processor instruction set. While the actual instruction sets for different processors are different, a commonality can be established based on the *types* of instructions supported. We assume a *basic* set of instructions listed in Table 1. This set of basic instructions groups together functionally similar operations. In addition, it also contains *macro*-operations that may not be available as single instructions but as a group of processor instructions, for example, call and return. These macro-assembly instructions are often needed for compilation efficiency and to preserve the *atomicity* of certain operations in the flow graph model. These operations also help in software delay

[2] Traditionally accumulator-based instruction sets have also lately adopted this programming model, with the exception of Intel floating-point which continues to use a stack-based programming model due to compatibility reasons.

estimation by providing additional information which may not be readily available purely from looking at the instruction set of a processor.

There is a significant variation in the types of operands supported on different processors. Following the taxonomy in [15] we distinguish processors by either a Load-Store (LS) ISA where memory operations are confined to only two instructions, or Register-Memory (RM) ISA where all instructions may have at most one memory operand or by Memory-Memory (MM) ISA which allows all operands to be memory operations.

Thus, the target processor is described using the following cost model,

$$\Pi = (\tau_{op}, \tau_{ea}, t_m, t_i), \quad (1)$$

where the execution time function, τ_{op}, maps *assembly language* instructions to positive integer delays. The address calculation function, τ_{ea}, maps a memory addressing mode to the integral delay (in cycles) encountered by the processor in computing the effective address. An addressing mode specifies an immediate data, register or memory address location. In the last case, the actual address used to access the memory is called the effective address. When generating programs from HDL descriptions only limited 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. Neither of these conditions occur when generating code from HDL. Further, not all the addressing modes may be supported by a given processor. For example, the DLX processor supports only immediate and register addressing modes. t_m represents the memory access time. Finally, the interrupt response time, t_i, is the time that the 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). For the cost model, this parameter can be specified as a part of the operation delay function (as shown by entry `int_response` in Table 1).

We note our delay estimation procedure is limited by the choice Π for the processor cost model at the instruction level. A more detailed model would be needed to include machine organization-specific effects such as pipeline interference. To circumvent the problems due to inaccuracy in estimations, the Vulcan system also provides a mechanism to analyze the compiler output and feedback *aggregate* program delays for program constraint analysis.

3. 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.

Against this general framework, most synthesized hardware uses static resource allocation and binding schemes, and static or relative scheduling techniques [8]. Due to this static nature, operations that share

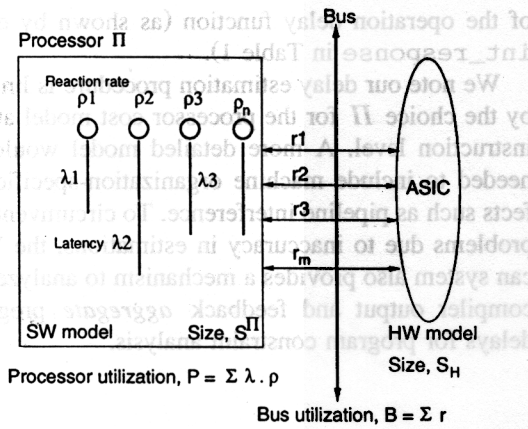


Fig. 2. Software model to avoid creation of $\mathcal{ND}$ cycles.

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. Consequently, there is no runtime system in hardware. Similarly, in software, the 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 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 that in our case both 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 lead to creation of $\mathcal{ND}$ cycles in the constraint graph, that is, cycles containing $\mathcal{ND}$ operations, which would make satisfiability determination impossible [5]. One way to avoid this problem is to ensure that the constraints do not span $\mathcal{ND}$ operations. However, this condition is restrictive and it may not always be possible to guarantee a linearization that meets this condition. As an alternative we construct software as a set of concurrent program threads as sketched in Fig. 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 λ_i of a thread i is the sum of the delay of its operations without including the initial $\mathcal{ND}$ operation whose delay is merged into the delay of 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:

(1) Processor utilization, $\mathcal{P}$ indicates utilization of the processor. It is defined as

$$\mathcal{P} \triangleq {}^{n}\sum_{i=1} \lambda_i \cdot \varrho_i. \tag{2}$$

In general, $\mathcal{P}$ is upper-bounded by unity. However, under certain runtime conditions, a lower upper bound may be necessary. These are discussed in Section 4.5.

(2) *Bus utilization*, $\mathcal{B}$ is 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,

$$\mathcal{B} \triangleq \sum_{j=1}^{m} r_j, \tag{3}$$

where r_j is the inverse of the minimum time interval between two consecutive samples for variable j. This interval is determined by the rate constraint on the input/output operation associated with the variable. In general, $\mathcal{B}$ is limited by the available bus bandwidth as a function of the bus cycle time and memory access time.

The utilization parameters $\mathcal{P}$ and $\mathcal{B}$ are global functions that are used to select operations for implementation into software. The resulting set of opera-

tions organized as marked flow graphs (Fig. 1) is implemented into multiple program threads by the linearization algorithm described in Section 4.3. The use of multiple concurrent program threads instead of a single program to implement the software avoids a complete serialization of all operations which may create unbounded (i.e., $\mathcal{ND}$) cycles. In this model of software, satisfiability of constraints on operations belonging to different threads can be checked for marginal or deterministic satisfiability [9], assuming a bounded delay on 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. Estimation of software performance

A program is compiled into a sequence of machine instructions. Therefore, timing properties of a program are related to the timing properties of the machine instructions to which it is eventually translated. Any variability in machine instruction timings is reflected on the variability of timing of programming-language statements. One approach to software estimation would be to generate such estimates directly from synthesized and compiled machine instructions for a given graph model. However, the process of compilation of high-level language programs is time intensive and may not be a suitable step when evaluating trade-offs among software and hardware implementations. Further, analysis of a compiled program can give timing properties of the entire programs, but not at the level of individual operations which is needed to exploit scheduling freedom in generating software. Therefore, alternative methods are sought for estimating software timing properties directly from programs written in high-level languages. Attempts have been made to annotate programs with relevant timing properties [16,17]. Syntax-directed delay estimation techniques have been tried [18,19] which provide quick estimates based on the language constructs used. However, syntax-directed delay estimation techniques rely solely on the structure of the programming languages, and thus lack timing information that is relevant in the context of the execution semantics of operations. (This distinction is, however, subtle since it is always possible to encode the context dependent timing information by choice appropriate syntactical structures.) More importantly, in our co-synthesis system, since flow graphs are central to constraint analysis, and hardware/software partitioning tradeoffs, therefore, we develop delay estimation on flow graph models using the semantic interpretation of flow graphs in our estimation procedures. A software delay consists of two components: delay of the operations in the flow graph model, and delay of the runtime environment. The effect of runtime environment on constraint satisfiability is considered in terms of operation *schedulability* for different types of runtime environments discussed further in Section 4.5. For now it suffices to say that the effect of runtime can be modeled as a constant overhead delay to each execution of the flow graph. We first focus on the delay due 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 [13]. Software delay then depends upon the delay of operations in the flow graph model *and* operations related to storage management. Calculations of storage management operations are described in Section 4.4.

4.1. Operation delay in a software implementation

Estimation of software performance is done under simplifying assumptions that tradeoff modeling accuracy against speed. We assume that the system bus is

always available for instruction/data reads and writes and that all memory accesses are aligned. The effect of storage alignment is considered in Section 4.4.1. Each operation v in the flow graph is characterized by the read accesses, $m_r(v)$, the write accesses, $m_w(v)$ and the assembly-level operations, $n_o(v)$. The software operation delay function, η, is computed as follows:

$$\eta(v) = \sum_{i=1}^{n_o(v)} t_{op_i} + (m_r(v) + m_w(v)) \times m_i, \quad (4)$$

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 overestimated from real execution delays. Nevertheless, in the one-level memory model, the number of read and write accesses of an operation can be estimated from its fanin and fanout (see Example 4.1).

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. A model of register usage and its effect is described later in Section 4.4. Based on this model, we determine the memory access operations and their contribution to the software delay.

4.1.1. $\mathcal{ND}$ operations

Wait operations in a graph model induce a synchronization operation in the corresponding software model. Thus, the software delay of wait operations is estimated by the *synchronization overhead* which is related to the program implementation scheme being used. One implementation of a synchronization operation is to cause a *context switch* in which the waiting program thread is switched out in favor of another program thread. It is assumed that the software component is computation intensive, and thus the wait time of a program thread can always be overlapped by the execution of another program thread. After the communication operation associated with the wait operation is complete, the waiting program thread is resumed by the runtime scheduler using a specific scheduling policy in choosing among available program threads. Alternatively, the completion of communication operation associated with wait operation can also be indicated by an interrupt operation to the processor. In this case, the synchronization delay is computed as follows:

$$\eta_{\text{intr}}(v) = t_i + t_s + t_o, \quad (5)$$

where t_i is interrupt response time, t_s is interrupt service time, which is typically the delay of the service routine that performs the input read operation and t_o is concurrency overhead [13] which constitutes a 19-cycle delay for the simplified co-routine implementation on the DLX processor. Both schemes have been implemented in our co-synthesis system.

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

Example 4.1. For the graph model shown in Fig. 3, 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 are given by the number of input edges, the software delay associated with the graph model

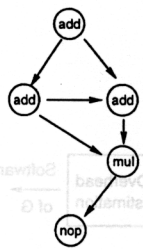


Fig. 3.

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.

4.2. 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. 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,

$$S^{\Pi}(\Phi) = \sum_{G_i \in \Phi} S^{\Pi}(G_i) = \sum_{G_i \in \Phi} \left[S_p^{\Pi}(G_i) + S_d^{\Pi}(G_i) \right]. \tag{6}$$

We postpone the discussion on estimation of program size to later in this section. S_d^{Π} consists of storage required to hold variable values across operations in the flow graph and across the machine operations. [3] This storage can be in the form of specific memory locations or the on-chip registers. In general, $S_d^{\Pi}(G)$ would correspond to a subset of the variables used to express a software implementation of G, that is,

$$S_d^{\Pi}(G) \leq |M(G)| + |P(G)|, \tag{7}$$

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. Interpretation of variables in relation to flow graph is discussed in 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 do the data transfer between variables and their mappings into the set $S_d^{\Pi}(G)$. 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 $V(G)$ would be constant under any schedule. However, the data set $S_d^{\Pi}(G)$ of G would vary according to scheduling technique used. Accordingly, the number of operations needed to do 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 [20]. Hence, heuristics solutions are commonly used. Most popular heuristics for code generation use a specific order

[3] To be precise, S_d^{Π} also includes the initialized data storage.

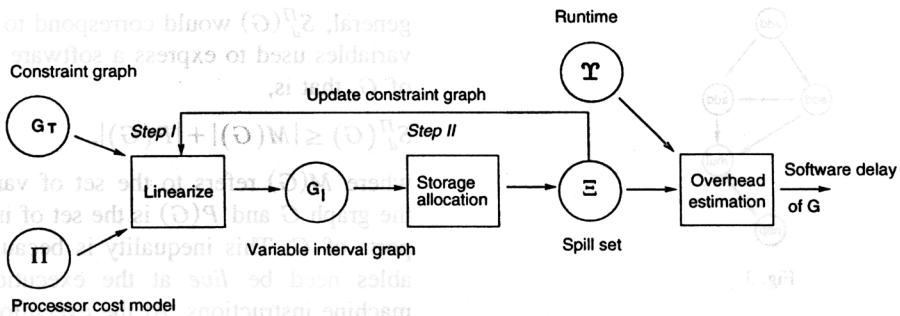


Fig. 4. Software delay estimation.

of execution of successor nodes (e.g., left-neighbor first) to reduce the size of S_d^{Π} [21].

In contrast to the register assignment in conventional software compilers which carry out simultaneous register assignment and operation linearization, we devise a two-step approach to help estimate the effect of data storage and software delays resulting from additional memory operations. We first linearize operations followed by an estimation of the register/memory operations. This two-step approach is taken to preserve the flexibility in operation scheduling which must take into account timing constraints not present in traditional software compilers. Fig. 4 illustrates the steps in estimation of software performance. Since spill operations add delay to a linearized set of operations, the timing analysis must ensure constraint satisfiability in view of the spill-related operations. It is also possible to combine the two steps into a single procedure where vertex linearization estimates for possible spill operations based on a given constraint on available registers and variable life-times. For sake of clarity we first describe the two steps individually.

4.3. Operation linearization

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. This complete order corresponds to a schedule of operations on a single resource, that is, the processor. 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 [20]. It is also possible that there exists no linearization of operations that satisfies all timing constraints. An exact ordering scheme under timing constraints is described in [8] that considers all possible valid orders to find the one that meets the imposed timing constraints. It also suggests a heuristic ordering technique that solves all-pair shortest path for each iteration of the linearization operation. In [22] 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 following procedure *linearize* outlines the algorithm. The input to the algorithm is a constraint graph model consisting of forward and backward edges as mentioned earlier. Recall, a backward edge represents a maximum delay constraint between the initiation times of two operations, whereas a forward edge represents a minimum delay constraint between the operation initiation times. By default, a non-zero operation delay leads to a minimum delay constraint between the operation and its immediate successors.

The algorithm consists of following three steps indicated by the symbol (▷):

```
linearize(G = (V, E_f ∪ E_b)) {
   δ(s) = 0; Q = {source vertex of G};
   if positive_cycle(G)
      exit;
   repeat {
▷I    v = extract head(Q);
      G' = G;
      add edge (s, v) in G' with weight = δ(s);
      for all w ∈ Q and w ≠ v
         add edges (v, w) in G' with weight = δ(v);
      {determine spill set for the linearized set of ops}
▷II   if positive_cycle(G')
         mark and move v to tail of Q;
      else {
▷III     if v is head of an edge, (u, v) ∈ E_b
            δ(u, v) = δ(u, v) − δ(s);
         for all w ∈ succ(v) s.t. pred(w) = ∅
            Q = Q + {w};
         remove v from Q; output v;
         δ(s) = δ(s) + δ(v);
         G = G';
         sort Q by urgency labels;
      }
   } until Q = ∅;
}
```

(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

(1) Select a root operation to add to the linearization,

```
/* initialize */
/* no valid linearization exists */
/* vertex with smallest urgency label */
/* construct new constraint graph */
/* select a candidate vertex */
/* linearize candidates' siblings */
/* discussed later */
/* not a feasible linearization */
/* discard candidate */
/* we have a good candidate */

/* update Q with new root vertices */
/* delete vertex v and its successor- */
/* -edges in G' */
```

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 has a schedule under timing constraints *assuming unlimited resources*. A feasible schedule under no resource constraints corresponds to an assignment of operation start times according to 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 because of 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 the Liao–Wong algorithm [23] 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 is indicated by the procedure *positive_cycles()* that 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 linearization 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 as a last resort. The following example graphically illustrates the linearization procedure (Fig. 5).

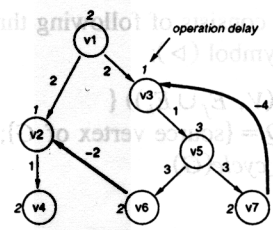


Fig. 5.

Example 4.2. Consider the flow graph shown in Fig. 5. We initialize $Q = \{v_1\}$ and $\delta(s) = 0$. Successive iterations of the linearization algorithm are shown in Fig. 6. At each iteration, operation label σ is obtained by application of procedure *positive_cycle()* on the constraint graph. The operation linearization returned by the algorithm is $v_1, v_3, v_5, v_7, v_2, v_6, v_4$.

The time complexity of *linearize()* is dominated by *positive_cycle()* that takes $O(|V|^2 \cdot k)$ time where k is the number of backward edges. Since after each iteration a vertex is removed, therefore, the complexity is $O(|V|^3 \cdot k)$ where k is typically a small number.

4.4. Estimation of register, memory operations

The number of read and write accesses is related to the amount and allocation of static storage, $S_d^{\Pi}(G)$. 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 $n_i =$

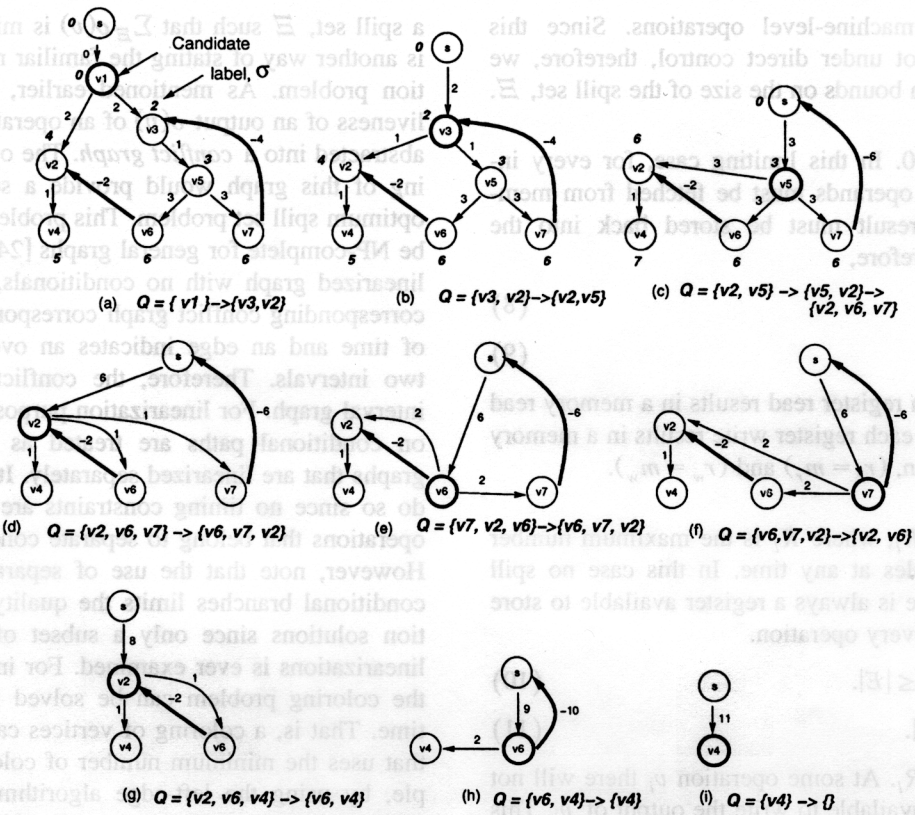


Fig. 6. Example of operation linearization under timing constraints.

$|\{\text{source vertices}\}|$ and $n_o = |\{\text{sink vertices}\}|$. Let r_r and r_w be the number of register read and write operations, respectively. Finally, recall that m_r represents the number of memory read operations and m_w represents the number of memory write operations.

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, which results 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 1. $R = 0$. 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,

$$m_r = |E|, \tag{8}$$

$$m_w = |V|. \tag{9}$$

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 2. $R \geq R_l$, 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.

$$m_r = n_i \leq |V| \leq |E|. \tag{10}$$

$$m_w = n_o \leq |V|. \tag{11}$$

Case 3. $R < R_l$. 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 is chosen for spill.

$$m_r = n_i + \sum_{\Xi} o(v_i) \leq \sum_{V} o(v_i) = |E|. \tag{12}$$

$$m_w = n_o + |\Xi| \leq |V|. \tag{13}$$

Clearly, the choice of the spill set determines the actual number of memory read and write operations needed. The optimization problem is then to choose a spill set, Ξ such that $\Sigma_{\Xi} o(v)$ 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 [24]. In case of a linearized graph with no conditionals, nodes in the corresponding conflict graph correspond to intervals of time and an edge indicates an overlap between two intervals. Therefore, the conflict graph is an interval graph. For linearization purposes, operations on conditional paths are treated as separate subgraphs that are linearized separately. It is possible to do so since no timing constraints are supported on operations that belong to separate conditional paths. However, note that the use of separate graphs for conditional branches limits the quality of linearization solutions since only a subset of the possible linearizations is ever examined. For interval graphs, the coloring problem can be solved in polynomial time. That is, a coloring of vertices can be obtained that uses the minimum number of colors, for example, by using the left-edge algorithm. However, a problem occurs when the number of registers available is less than the minimum number of registers needed. In this case, outputs from a set of vertices should be spilled to memory and the conflict graph modified accordingly so that the new conflict graph is colorable. We use the following heuristic to select operations for the spill. First, a conflict graph, G_l for a given schedule is built by drawing an edge between v_i and v_j if any of the output edges of v_i span across v_j. From this conflict graph, we select a vertex, v with outdegree less than R. This vertex is then assigned a register different from its neighbors. From this we construct a new conflict graph G_l' by removing v and its fanout edges from G_l. The procedure is then repeated on G_l' until we have a vertex with outdegree greater than or equal to R. In this case, a vertex is chosen for spilling and the

process is continued. Example 4.3 illustrates the procedure.

For these calculations, we assume that each v is implemented as a closed sequence of assembly language instructions, though it is possible that the source language compiler may rearrange operations. The effect of this rearrangement, however, can be assumed only to reduce the total register usage requirements.

4.4.1. Storage alignment

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.

Table 2 lists data types and alignment requirements which are taken into account in the determination of the data size. The size of a *structure* 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 (up to a maximum 3 bytes per member) for padding. In the case of a structure consisting entirely of bit fields, there is no padding if the total number of bits is less than 32 bits. In case of structure widths greater than 32 bits, additional 32-bit words are assigned and members that lie on the boundary of two words are moved to the subsequent word leaving a padding in the previous word. It is assumed that no member takes more than 32-bits. 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.

Table 2
Variable types and alignment used

Data type	Size	Address alignment
int	32	%4
short	16	%2
char	8	%1
pointer	32	%4
struct	*variable. see text.*	

Example 4.3. 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;
   e:5}
f[33]    int f[2]
```

Minimum storage used in the flow graph model is 6 bytes. However, due to alignment requirements the actual data storage is 12 bytes.

4.4.2. Effect of multiregister nodes

The register usage of compilers is determined by the generation of r-value and l-value [21] for each statement in the generated C code. The r-value is the result or value of evaluation of an expression (or simply the right-hand side of an assignment). In the case of logic nodes, the (recursive) organization of equations gives an estimate of the number of r-values needed. Note that logic operations frequently need shift operations. A shift operation is modeled as an additional r-value that is accounted for in computation of the total number of r-values associated with the logic operation. For most assignment statements, the left side generates a l-value and the right side generates a r-value. However, in case of pointer

assignments (such as those using variables bound to ports and multiword data, e.g., `f[]` in Example 4.3) and structure and indirect member references (common in logic nodes), the left-hand side also generates *r*-values which are subsequently assigned to the appropriate *l*-value also generated by the left side. This generation of a *r*-value by the left side happens in two cases: write and logic operations. In both cases, the left-hand side generates a *r*-value that is assigned to a left-hand generated *l*-value.

We extend our estimation procedure for the spill set in the case of operation vertices with multiple

```
Input: flow graph model, G(V, E)i
Output: S(G), static storage for a linear code implementation of G
single_thread_static_storage(G) {
  H = linearize(G)                                   /* linearize vertices */
  count = storage = 0;
  ∀u ∈ V(H) {                                        /* determine max live variables */
    ∀v ∈ succ(u)
      count = count + ω(u > v);                      /* add new registers */
    ∀v ∈ pred(u)
      count = count − ω(v > u);                      /* subtract registers for completed operations */
    storage = max(count, storage);
  }
  return storage
}
```

Finally, the program size, S_p^{Π} is approximated by the sum over *r*-values 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 *r*-values. This is only an upper bound since global optimizations in general may reduce the total number of instructions generated.

4.4.3. RM ISA architectures

Practical RM ISA architectures feature small register sets with non-orthogonal instructions. That makes register spills a very common occurrence in the compiled code. But more importantly, due to non-orthogonality of instructions, a substantial number (up to 27%) of *inter-register* data transfer instructions is generated to position data to appropriate registers from other registers [15]. These instructions do not affect the data storage, S_d^{Π}, but alter the size of the program thread and its delay. It is hard to model such instructions since these are dependent upon actual algorithms used in software compilation. At this time, to our knowledge, there is no analytic model available for this problem. As a first step, the following conjecture is suggested to estimate a bound on the additional operations.

register usage simply by using a weighted graph model where the weight of each vertex is given by the number of *r*-values it generates, that is, $G^D = (V, E, \omega)$ where $\omega(v) = |r\text{-}value(v)|$ and $\omega(e = v_i \succ v_j) = \omega(v_i)$. The above relations hold by replacing $|V| = \sum_V \omega(v)$ and $|E| = \sum_E \omega(e)$. The following procedure *single_thread_static_storage(G)* determines the maximum number of live variables, R_l, in a linearized graph model, *G*. We assume that each operation vertex requires at least one cycle, and hence any data transfer across operation vertices in the flow graph requires a *holding* register.

Conjecture 4.1. For a given graph model, $G = (V, E)$, the following sum

$$\Gamma = m_r + m_w + r_m = |V| + |E| \tag{14}$$

is constant for all architectures. Here, r_m represents the number of inter-register transfer operations.

The intuitive reasoning behind this conjecture is that for a machine with no general-purpose registers, there will be no inter-register operations since operands can be loaded directly into the required operand registers, and thus $r_m = 0$. This corresponds to the first case discussed earlier. In the other two cases, the total number of data movement operations can not be worse than the case with no registers. Based on this conjecture, we can estimate the inter-register transfer operations by first computing $m_r + m_w = f(R)$ as a function of the number of available registers, R and then applying Eq. (14) above.

Example 4.4. Consider the flow graph, G, shown in Fig. 7 consisting of 11 vertices and 12 edges with $n_i = 3$ and $n_o = 2$.

Each vertex produces an output variable that is named by the vertex label. Vertices a, b, c are input read operations, and vertices x, y are output write operations. The linearization results in the following order c, f, b, e, a, d, h, i, y, j, xwhich gives the maximum number of live variables, $R_l = 4$ according to algorithm *single_thread_storage*. R_l is the size of the largest clique in the interval graph shown in Fig. 8. In the interval graph, G_I, vertices represent

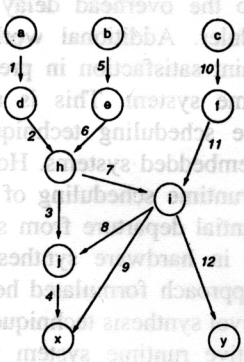


Fig. 7.

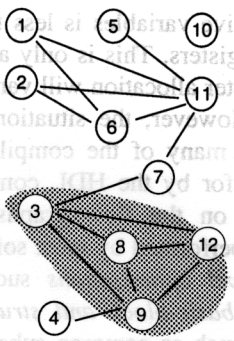


Fig. 8.

edges of G and an edge between two vertices in G_I indicates overlap between the corresponding edges in G.

For $R \geq R_l = 4$, register assignment can be done in polynomial time. The spill set, $\Xi = \varnothing$. Therefore, $m_r = n_i = 3$ and $m_w = n_o = 2$. Total number of memory operations = 5. For $R = 3$, the largest clique in G_I should be of cardinality 3 or less. For $\Xi = \{9\}$, the total number of memory operations = 7. For $\Xi = \{9, 3, 11\}$, the maximum number of live registers is reduced to 2, while the number of memory operations increases to 11. Note that in the worst case of static storage assignment for all variables in G, there would be 11 + 12 = 23 memory operations:

R_l	Spill set, Ξ	Memory operations, $m_r + m_w$
≥ 4	{ }	5
3	{9}	7
2	{9, 3, 11}	11

Note that we are not directly trying to minimize total register usage by the object program since operations at that abstraction level belong to the software compiler. The objective of spill set enumeration is to arrive at an estimate of memory operations assuming that the program is compiled by a reasonably sophisticated software compiler that achieves optimum register allocation when the maxi-

mum number live variables is less than the number of available registers. This is only an estimate since the actual register allocation will vary from compiler to compiler. However, the situation is not entirely hopeless since many of the compiler optimizations are accounted for by the HDL compiler itself, and thus reflected on the flow graphs. The common optimizations performed by most software compilers are: *source-level optimizations* such as procedure in-lining and *basic-block and structure-preserving optimizations* such as common subexpression elimination, constant propagation, tree-height reduction, dead-code elimination, variable renaming. All of these optimizations are taken care of by the HDL compiler and, hence, are reflected in the flow graphs. Among the remaining optimizations are copy propagation, code motion, induction variable elimination, machine register allocation, operator strength reduction, pipeline scheduling, branch offset minimization. Of these, the most significant is machine register allocation that accounts for the most increase in efficiency (20–50%) [15]. Register allocation is possible and most effective for local variables. These locals are defined by the storage $M(G)$ associated with the flow graph model. Therefore, much of our delay estimation is focused on understanding this aspect of compilation.

4.5. Effect of runtime scheduler

The runtime scheduler refers to the main program in software that integrates calls to various program threads implemented as co-routines [25]. As explained earlier, the runtime system used here mainly consists of a scheduler that invokes program threads at runtime.

We assume a *non-preemptive runtime scheduler* where 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. Alterations in scheduling discipline are then sought to ensure fairness, that is, the best prioritized discipline leads to least likelihood of a starvation.

A *preemptive* runtime scheduler provides 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 most feasible schedules.

We have so far implemented only non-preemptive runtime scheduling techniques. 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. Additional work is needed to ensure constraint satisfaction in presence of a preemptive runtime system. This is not to say that non-preemptive scheduling techniques are always sufficient for embedded systems. However, the very ability to do runtime scheduling of operations provides a substantial departure from static scheduling schemes used in hardware synthesis, and for the co-synthesis approach formulated here as an extension of high-level synthesis techniques, the choice of a non-preemptive runtime system provides a first step towards synthesizing embedded systems. In the

sequel, we consider only non-preemptive schedulers which may or may not be prioritized.

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., $\mathscr{P} \leq 1$. However, this condition is not sufficient. Consider, for example, the case when the software contains a program thread with a long latency but very low reaction rate. Such a program thread will bound the achievable reaction 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:

for thread, T_i

$$\frac{1}{\varrho_i} \geq \sum_{\text{thread } k=1}^{n} \lambda_k \Rightarrow \frac{1}{\max_i \varrho_i} \geq \sum_{\text{thread } k=1}^{n} \lambda_k, \tag{15}$$

where n is the total number of program threads. This condition is also necessary and sufficient for *independent* threads. In case of dependent program threads, only a subset of the total program threads is 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. 15. It is interesting to note that the same condition for worst case reaction rate also applies for RR schedulers, though the average case performance differs.

4.5.1. Prioritized runtime scheduler

A prioritized FIFO scheduler consists of a *set* of FIFO buffers that is prioritized such that after the 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 since it is possible that a low priority thread may never be scheduled due to starvation.

5. Correctness of the software synthesis procedure

As mentioned earlier in Section 3, the concurrency between the program threads is achieved by using an interleaved execution model. For a functionally correct execution in software it is essential that the execution of operations in a program thread yields the same result (modulo timing) as would the execution of the operations in the input flow graph model. We formulate this condition as a **serializability condition** on the flow graphs. Serializability is a necessary (but not sufficient) condition for generating a single-processor embedded software.

Serializability is a concern, for instance, when two operations in a program thread belong to two graphs or they are specified within the same graph using the "forced parallel" semantics in HardwareC (see example below).

We consider the interleaved executions of two program threads *speed independent* if the actions performed by a program thread have no influence on the rate at which it proceeds. The reaction rate of a program thread depends on the presence of other threads and the concurrency structures. A sufficient

condition for speed independence is that the program threads implement pure functions [26], that is, program threads be *functional*. However, the requirement for a functional behavior is only an abstraction and need not imply an implementation of program threads as functions. In practice, it is sufficient if the storage common to program threads is accessed sequentially. This introduces the notion of *critical sections* in program threads, that is, operations that access or modify the shared storage. One way to achieve the functional property for a thread is by blocking on critical sections such as by using semaphores.

From a practical point of view, the functionality property is not very useful due to excessive overhead incurred in implementation of blocking and the resulting loss of timing certainty in software. Therefore, we focus on methods to achieve speed independence of program threads by a close examination of the thread side-effects on storage and input/output. As mentioned earlier the operational semantics of a flow graph, G, uses a set of variables, $M(G)$, that are associated with the operations. An operation in $V(G)$ can take the following actions on a given variable, u in $M(G)$:

Defines u. 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⟩;

Uses u. When the operation corresponds when u appears in an expression or the right-hand side (rhs) of an assignment;

Tests u. 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. Example 5.1 illustrates the difference.

Example 5.1. Variable definition, use, and testing. Consider the HDL fragment below:

```
process vars (inp1,inp2,outp)
   in port inp1[size], inp2[size];
   out port outp[size];
{  int a,b;
   boolean c;
1:  a=read(inp1);
2:  b=read(inp1);
3:  ⟨
    3.1:  c=fun(a,b);
    3.2:  if (c) [
    3.2.1:  write outp=a;
    3.2.2:  write outp=b;
          ]
   ]
4: c=b+c;
}
```

The above HDL program has the following semantics:

- Steps 1, 2, 3, 4 are **data parallel**. These steps may execute in parallel, at different possible speeds, as long as all the data dependencies are not violated.
- Step 3 is **forced parallel**. This means that the sub-steps 3.1 and 3.2 must be initiated at the same time.
- Step 3.2 is **sequential**. The semantics of this statement is that sub-steps 3.2.1 and 3.2.2 must execute sequentially.

From the semantics of forced-parallel executions, note that the value of variable "c" tested in 3.2 may be different from the one computed by 3.1.

First two assignments (statement 1 and 2) define variables a and b whereas assignment 3.1 uses both these variables and defines variable c. Operation 3.2 tests variable c where as operation 4 uses and defines this variable.

We note that the distinction between variable use and test operations is new and specific to the problem of hardware–software co-synthesis. A variable that is only tested (and not used) is considered a control variable. This distinction between use and test is made to model different strategies for control and data transfer across a hardware–software partition [27].

5.1. Testing serializability

We assume that each operation (vertex) in G is implemented as an atomic operation in hardware or software. We explore conditions for operation serializability by examining the interactions of variables used in concurrent operations. 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.

From our discussion of speed independence of program threads, a sufficient condition for serializability is to check if program threads are functional such that there are no shared variables. In presence of shared variables the program threads will contain critical sections. Any interleaving of operations in the critical sections must ensure that the definition and use ordering relations on shared storage are not interfered by competing concurrent operations. For concurrent operations that both define and use a variable, the ordering between operations will affect the output, and thus make any interleaving of operations impossible.

Therefore, 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 operation. However, note that variables that are both used and tested can be shared between concurrent operations. Thus, the serializability of a graph is checked by examination of definition and use operations for its variables, to ensure that no shared variables are both defined and used by the same operation.

As mentioned earlier, for concurrent operations across process graph models, the communication is only by message-passing operations. In absence of any shared storage program threads created from separate flow graphs are always functional. For program threads created from the same flow graph, the condition of serializability requires examination of concurrent operations (i.e., operations without any transitive dependencies) that may belong to the same or different program threads.

5.2. Unserializable flow graphs

There are two situations when a flow graph is not serializable. First condition is when two operations are required to execute in parallel *regardless* of the 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 variable, concurrent read–write operations to the variable may refer to operations on two separate values of the variable corresponding to the master and slave portions of the flip-flop. Flow graphs with forced-parallel operations are made serializable using the following procedure:

1. *Decompose* concurrent operations into a 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 5.2 shows the transformation.

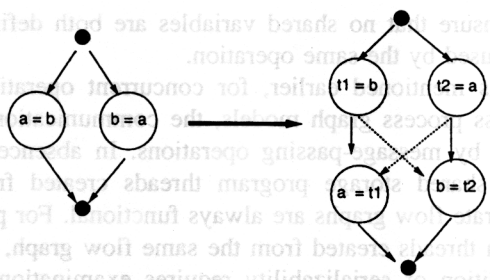


Fig. 9. Creation of serializable flow graphs.

Example 5.2. 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 Fig. 9. 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 use of shared memory [8]. The operations in the loop body have access to the storage defined in the calling graph. Here, the program thread corresponding to the loop body may have access to storage in the program thread corresponding to the parent graph. We address this case by making explicit all data transfers between the loop graph and the parent graph and then performing the serializability tests as in the case of concurrent processes.

6. Results

The linearization heuristic presented here was tried on a number of constraint graphs. The constraint graph input to the linearization algorithm is characterized by the number of min and max constraints. Among feasible linearizations, the quality of a linearization is defined by the size of the spill set, Ξ, that is, the number of variables that must be spilled to meet a given limitation on maximum number of registers available. Table 3 compares the efficiency of the proposed heuristic linearization and an exact linearization algorithm that uses backtracking to arrive at a feasible linearization. "Optimum" explores all possible linear order to generate a solution that uses a smallest spill set. Note that the exact method guarantees *a* feasible linearization if one exists, whereas "Optimum" method selects the best linearization from the set of all possible of feasible linearizations. In contrast, a heuristic linearization may result in no feasible solution when in fact there exists one. It is difficult to precisely characterize cases where this heuristic fails, but from our experiments we could derive a correlation between the topology of the constraints and the likelihood of a feasible solution. We call a timing constraint of *sequencing type* if the constraint is represented between two operations that are directly or transitively related. In contrast, *synchronization type* constraints are indicated between operations that do not have any dependency relation between them. The heuristic

Table 3
Comparison of efficiency of linearization algorithms

Constraints min, max	Exact	Heuristic	Optimum
8,4	31	6	78
14,2	152	9	585
12,2	35	9	266
19,1	74	11	738
19,3	643	11	2069
16,3	25	12	31
20,8	334	9	758
24,2	23	13	42
34,3	18168	18	144735
Spill set size	79	86	76

favors feasible solutions for "tall and thin" constraint graphs that are dominated by synchronization type constraints, and "short and obese" graphs that are dominated by sequencing type constraints. Overall the heuristic works well in practice, and the infeasible solution situation is rarely encountered in cases where the graphs are dominated by tight sequencing *and* synchronization constraints.

As shown in Table 3, the heuristic linearization based on operation urgency proposed in this paper provides substantial improvement in runtime over both exact and optimum linearization methods. Operations are selected for spilling based on a heuristic that the latest definitions in the partially linearized set of operations are spilled first. This choice reduces the likelihood of spilling variables that have been live over relatively longer times, thus reducing the fragmentation of a long live range into smaller ranges. Comparing the spill set size, the heuristic algorithm results in a spill set size that is 8–13% larger than the optimum.

We now consider practical design examples to illustrate the nature of the software component in co-synthesis designs.

6.1. Vehicle cruise controller

The cruise controller monitors vehicle cruising speed, a record of average speed, monitors fuel consumption and provides vehicle status and maintenance feedback to the driver through a display controller. Fig. 10 shows an overview of the controller.

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

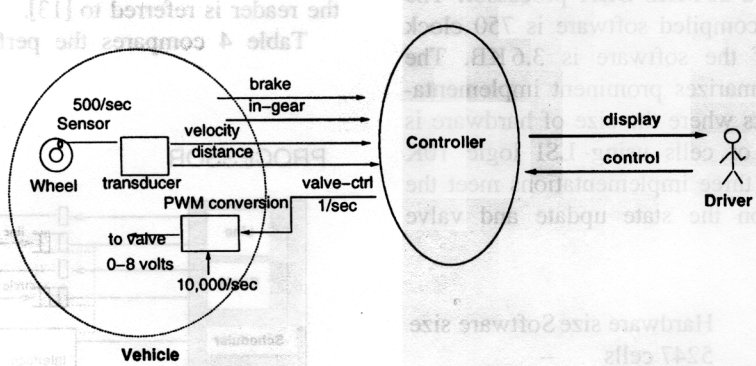


Fig. 10. Controller specification diagram.

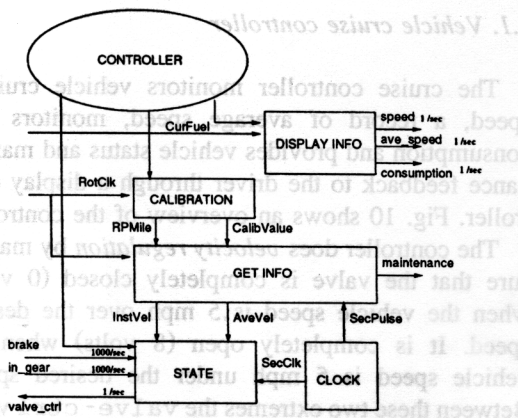


Fig. 11. Controller implementation block diagram.

less than 1 second, except for braking which must be as fast as possible (at most 1 ms).

This controller was described modularly as a collection of 7 blocks using a total of approximately 1000 lines of hardware description language (HDL) code. The final implementation through use of co-synthesis techniques is shown in Fig. 11. A hardware–software implementation that considers implementation of the STATE block into software while the rest is implemented into hardware. The worst case delay of the program estimated from the flow graph of the STATE block is 1968 cycles (362 cycles for best case) which corresponds to a software delay of 0.1 ms for a 20 MHz DLX processor. The actual delay of the compiled software is 750 clock cycles. The size of the software is 3.6 KB. The following table summarizes prominent implementation alternative points where the size of hardware is expressed in terms of cells using LSI logic 10K library of gates. All three implementations meet the timing constraints on the state update and valve control block.

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

6.2. Graphics controller

Fig. 12 shows a mixed implementation of a graphics controller for generating 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.

The mixed implementation of the system design consists of line and circle drawing routines in the software while the ASIC hardware performs the 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. 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. For details on interface synchronization the reader is referred to [13].

Table 4 compares the performance of different

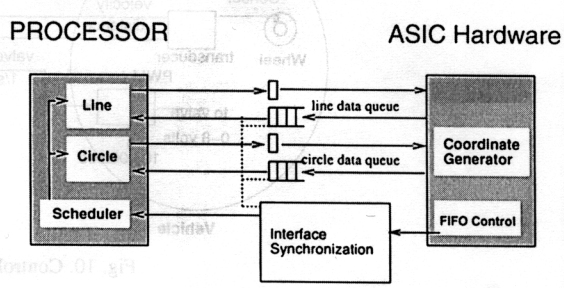


Fig. 12. Graphics co-processor implementation.

Table 4
System performance under different software implementation schemes

Scheme	Software size (bytes)	Runtime overhead	Input data rate^{-1} (cycles/coordinate)	Output data rate^{-1} (cycles/coordinate)			
				line		circle	
				ave.	peak	ave.	peak
Hardware interface	5972	19	81	535.2	330	76.4	30
Software interface	6360	19+84=103	95	651	407	94	31

program implementations using hardware and software interfaces. The hardware interface implementation uses a FIFO queue with associated control logic that is synthesized into 228 gates using LSI 10K library of gates. A software implementation uses predefined interrupt lines to input data to the processor. Such an implementation adds 388 bytes to the overall program size of the software component and increases the runtime overhead by 84 cycles. Note that data input and output rates have been expressed in terms of number of cycles it takes to input or output a coordinate. Due to a data-dependent behavior of the program threads, the actual data input and output rates would vary with respect to the value of the input data. In this example simulation, the input rate has been expressed for a simultaneous drawing of a line and 5 pixel radius with width of 1 pixel each and the results are accurate to one pixel. An

```
#include "transfer_to.h"

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

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

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

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

int nextCoroutine;

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

POLLED PROGRAM THREADS

```
#include "transfer_to.h"

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

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

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

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

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

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

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

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

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

int nextCoroutine;

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

INTERRUPT–DRIVEN PROGRAM THREADS

Fig. 13. Graphics controller software component.

input rate of 81 cycles/coordinate corresponds to approximately 0.25 million samples/sec for a processor running at 20 MHz. Similarly, a peak circle output rate of 30 cycles/coordinate corresponds to a rate of 0.67 million samples/sec. Fig. 13 shows implementation of the *main* program that provides runtime scheduling of threads for the two cases of interface implementations.

7. Summary

The algorithms for software generation described here are implemented in C programming language as a part of the VULCAN co-synthesis system. This paper presents important issues in software generation from our experience in building this system. There are two main problems in generating software from flow graphs. First, since the program generation necessarily requires serialization of operations in the flow graph, one must ensure that it is indeed possible to preserve the HDL-modeled behavior through such serialization. We have developed conditions that are sufficient to ensure that such a serialization is possible based on variable definition and the use of analysis on the flow graphs. The actual code generation is a fairly straightforward procedure and is omitted from discussions here. The second problem relates to presence of $\mathcal{ND}$ operations. We have presented a model for the software and the runtime system that consists of a set of program threads which is initiated by synchronization operations. Use of multiple program threads avoids a complete serialization of operations which may otherwise create $\mathcal{ND}$ cycles in the constraint graph.

Under timing constraints, linearization of a part of a flow graph may also not be possible. We have presented a heuristic linearization algorithm to do operation linearization using a vertex elimination scheme. This scheduling scheme uses a measure of urgency based on timing constraint analysis on an unconstrained implementation. This heuristic is substantially faster than exact ordering search algorithms. Since the quality of a final solution is determined by the existence of a feasible schedule under timing constraints, the suggested heuristic is a better match for solving the linearization problem for mixed system designs.

Our current implementation does not consider the delay due to alignment operations in case of packed implementations which can be significant. Variable packing techniques are needed to minimize the alignment operations and their effect on the software performance. We are considering ways to do software estimation and code generation in view of the internal *organization* of the processor being used using a HDL model of the target processor to improve the effectiveness of code-generation for embedded systems.

Acknowledgements

This research was sponsored by a grant from the AT&T Foundation, and NSF CAREER Award MIP 95-01615 and grants from NSF EEC 89-43166 and NSF-ARPA No. MIP 9115432.

References

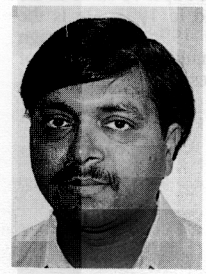
[1] D.E. Thomas, J.K. Adams and H. Schmit, A model and methodology for hardware–software code-sign, *IEEE Design and Test of Computers* (September 1993) 6–15.

[2] M. Chiodo, P. Giusto, A. Jurecska, A. Jurecska, A.S. Vincentelli and L. Lavagno, Hardware–software codesign of embedded systems, *IEEE Micro* 14(4) (August 1994) 26–36.

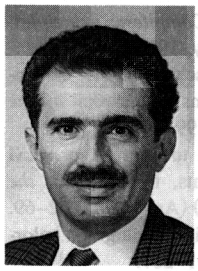
[3] P. Pfahler, C. Nagel, F.-J. Rammig and U. Kastens, Design of a VLIW architecture constructed from standard RISC chips: A case study of hardware/software codesign, in: *Proceedings of 19th EUROMICRO Symposium* (September 1993) 6–9.

[4] R. Ernst, J. Henkel and T. Benner, Hardware–software cosynthesis for microcontrollers, *IEEE Design and Test of Computers* (December 1993) 64–75.

[5] R.K. Gupta and G.D. Micheli, Hardware–software cosynthesis for digital systems, *IEEE Design and Test of Computers* (September 1993) 29–41.
[6] G.D. Micheli, D.C. Ku, F. Mailhot and T. Truong, The Olympus synthesis system for digital design, *IEEE Design and Test Magazine* (October 1990) 37–53.
[7] A. Jerraya, K. O'Brien and T.B. Ismail, Linking system design tools and hardware design tools, in: *International Conference on Computer Hardware Description Languages and their Applications – CHDL'93* (April 1993).
[8] D. Ku and G.D. Micheli, *High-Level Synthesis of ASICs under Timing and Synchronization Constraints* (Kluwer Academic Publishers, 1992).
[9] R.K. Gupta and G.D. Micheli, Specification and analysis of timing constraints for embedded systems, to appear in *IEEE Trans. on CAD*, 1996.
[10] V. Cerf, Multiprocessors, Semaphores and a graph model of computation, Ph.D. Thesis, UCLA, April 1972.
[11] D. Bustard, J. Elder and J. Welsh, *Concurrent Program Structures* (Prentice Hall, 1988) p. 3.
[12] B. Dasarathy, Timing constraints of real-time systems: Constructs for expressing them, method of validating them, *IEEE Trans. Software Engg.* SE-11(1) (January 1985) 80–86.
[13] R.K. Gupta, *Co-Synthesis of Hardware and Software for Digital Embedded Systems* (Kluwer Academic Publishers, Boston, 1995).
[14] Y.-T.S. Li, S. Malik and A. Wolfe, Performance estimation of embedded software with instruction cache modeling, in: *Proc. of the IEEE International Conference on Computer-Aided Design* (November 1995).
[15] J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach* (Morgan Kaufmann, 1990).
[16] A. Shaw, Reasoning about time in higher level language software, *IEEE Trans. Software Engg.* 15(7) (July 1989) 875–889.
[17] A. Mok et al., Evaluating tight execution time bounds of programs by annotations, in: *Proc. of the Sixth IEEE Workshop Real-Time Operating Systems and Software* (May 1989) 74–80.
[18] C.Y. Park and A.C. Shaw, Experiments with a program timing tool based on source-level timing schema, in: *Proc. of the 11th IEEE Real-Time Systems Symposium* (December 1990) 72–81.
[19] W. Hardt and R. Camposano, Trade-offs in HW/SW codesign, in: *International Workshop on Hardware–Software Co-Design* (October 1993).
[20] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness* (W.H. Freeman and Company, 1979).
[21] A.V. Aho, R. Sethi and J.D. Ullman, *Compilers: Principles, Techniques and Tools* (Addison Wesley, 1986).
[22] P. Chou and G. Borriello, Software scheduling in the cosynthesis of reactive real-time systems, in: *Proc. of the Design Automation Conference* (June 1994).
[23] Y. Liao and C. Wong, An algorithm to compact a VLSI symbolic layout with mixed constraints, in: *Proc. of the IEEE Transactions on CAD / ICAS* 2(2) (April 1983) 62–69.
[24] G.J. Chaitin, Register allocation and spilling via graph coloring, *SIGPLAN Notices* 17(6) (1982) 201–207.
[25] R.K. Gupta, C. Coelho and G.D. Micheli, Program implementation schemes for hardware–software systems, *IEEE Computer* (January 1994).
[26] P.B. Hansen, *Operating System Principles* (Prentice-Hall, 1973).
[27] S. Agrawal and R.K. Gupta, System partitioning using global data-flow, Memorandum UIUC DCS 1995, University of Illinois, October 1995.



Rajesh Gupta (Ph.D. Stanford '93, M.S. UC Berkeley '86) is an Assistant Professor in Computer Science at the University of California, Irvine. His current research focus is on system-level design and CAD issues. Prior to joining Irvine, Gupta worked as an Assistant Professor at the University of Illinois, Urbana-Champaign from 1994 through 1996. He is author of a book on "co-synthesis of hardware and software for digital embedded systems" published by Kluwer. From 1986 through 1989, Gupta was at Intel Corporation in Santa Clara, California where he worked as a member of design teams for three generations of microprocessor devices. He is co-author of a patent on PLL-based clock circuit. Gupta was nominated for the NSF Presidential Faculty Fellow Award by the University of Illinois in 1996. He is a recipient of the National Science Foundation CAREER Award in 1995, and two Departmental Achievement Awards at Intel and several fellowships at Stanford and Berkeley. He serves on the program committees of Great Lakes Symposium on VLSI, CODES workshop, ICCD, ICCAD and DAC.



Giovanni De Micheli is Professor of Electrical Engineering, and by courtesy, of Computer Science at Stanford University. Previously, he held positions at the IBM T.J. Watson Research Center, Yorktown Heights, New York, at the Department of Electronics of the Politecnico di Milano, Italy and at Harris Semiconductor, Melbourne, Florida. He holds a Nuclear Engineer degree (Politecnico di Milano, 1979), a M.S. and a Ph.D. degree in Electrical Engineering and Computer Science (University of California at Berkeley, 1980 and 1983). His research interests include several aspects of the computer-aided design of integrated circuits and systems, with particular emphasis on automated synthesis, optimization and validation. He is author of: *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994, co-author of: *High-Level Synthesis of ASICs under Timing and Synchronization Constraints*, Kluwer, 1992 and co-editor of: *Hardware/Software Co-Design*, Kluwer, 1995 and of *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation*, Martinus Nijhoff Publishers, 1986. He was also co-director of the NATO Advanced Study Institutes on Hardware/Software Co-Design, held in Tremezzo, Italy, 1995 and on Logic Synthesis and Silicon Compilation, held in L'Aquila, Italy, 1986. Dr. De Micheli is a Fellow of IEEE. He was granted a Presidential Young Investigator award in 1988. He received the 1987 IEEE Transactions on CAD/ICAS Best Paper Award and two Best Paper Awards at the Design Automation Conference, in 1983 and in 1993. He is the Program Chair (for Design Tools) of the 1996/1997 Design Automation Conference. He was Program and General Chair of the International Conference on Computer Design (ICCD) in 1988 and 1989, respectively.