JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **49,** 259–293 (1998)
ARTICLE NO. PC981442

A Unified Framework for Instruction Scheduling and Mapping for Function Units with Structural Hazards[1]

Erik R. Altman

IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598
E-mail: erik@watson.ibm.com

R. Govindarajan

Supercomputer Education and Research Center, Indian Institute of Science, Bangalore, 560 012, India
E-mail: govind@csa.iisc.ernet.in

and

Guang R. Gao

Department of Electrical Engineering, University of Delaware, Newark, Delaware 19716
E-mail: ggao@ee.udel.edu

Software pipelining methods based on an ILP (integer linear programming) framework have been successfully applied to derive rate-optimal schedules under resource constraints. However, like many other previous works on software pipelining, ILP-based work has focused on resource constraints of simple function units, e.g., "clean pipelines"—pipelines without structural hazards. The problem for architectures beyond such clean pipelines remains open. One challenge is how to represent such resource constraints for unclean pipelines, i.e., pipelined function units, but having structural hazards.

In this paper, we propose a method to construct *rate-optimal* software pipelined schedules for pipelined architectures with structural hazards. A distinct feature of this work is that it provides a unified ILP framework for two challenging and interrelated aspects of software pipelining—the scheduling of instructions at particular times and the mapping of those instructions to specific function units. Solving both of these aspects is essential to finding schedules which will work both on VLIW machines which map instructions to fixed function units and on dynamic out-of-order superscalars. We propose two ILP formulations to solve the integrated scheduling and mapping problem. Both adopt principles of graph

[1]This work was supported by research grants from NSERC (Canada) and MICRONET—Network Centers of Excellence (Canada).

0743-7315/98 $25.00

coloring in an ILP framework, and one uses *forbidden latencies* in an elegant extension of classical hardware pipeline control theory.

We have run experiments on four variations of our proposed formulations. As input we used a set of 415 "unique" loops taken from several benchmark suites, and we targeted an architecture whose function units contain many structural hazards. All four of our variations did well, with the best finding a rate-optimal schedule for 65% of the loops. This compares favorably with a leading heuristic, Huff's *Slack Scheduling*—the ILP approaches found a schedule with smaller initiation interval for over 50% of the loops, with a mean improvement of almost 30%. Finally, we have found that reusing pipeline stages—and thus adding hazards—results in only a 10% drop in performance, while permitting significant savings in area.

1. INTRODUCTION

Software pipelining overlaps operations from different loop iterations in an attempt to fully exploit instruction level parallelism. To be successful on real machines, the function unit constraints of those machines must be taken into account. A variety of heuristic-based software pipelining algorithms [1, 2, 7, 11, 16, 18, 20–22, 24, 26, 33, 35, 36] have been proposed which operate under resource constraints. An excellent survey of these algorithms can be found in [25]. More recently, integer linear programming (ILP)-based approaches for finding an optimal periodic schedule on machines with simple resource usage have been proposed [8, 12, 13]. Optimal enumeration-based methods have also been proposed [4, 34].

Real machines sometimes have function units with structural hazards, that is function units which reuse one or more stages of their pipelines during execution of an instruction. Because of the difficulty in modeling such pipelines, schedulers sometimes assume that no pipelining is possible, thus significantly reducing the throughput of the machine. For example, this often happens with `divide`. Were a good scheduling model available for such pipelines, more such function units might be designed with hazards: designing function units with structural hazards often allows them to be smaller. The architect is then left with the choice of having many slower function units versus fewer faster ones.

Despite significant progress in resource-constrained software pipelining with *clean* pipelines, i.e., pipelines with no structural hazards, finding optimal schedules for *unclean* pipelines has remained open. The problem is further complicated if the target machine is a VLIW (or has a VLIW microarchitecture). Such machines often associate opcodes positions in the instruction word with particular function units. In such cases, it is essential not only that the proper *number* of function units be available, but that the *particular* function units needed by each operation be available. In the software pipelined version of a loop, this in turn requires that operations execute on the same function unit in each iteration. We illustrate these points with an example in Section 3.

The main contributions of this paper come in addressing the problem of scheduling for function units with hazards and scheduling for machines requiring a fixed mapping of instructions to function units:

- We propose two ways in which earlier integer programming approaches can be modified to simultaneously deal with structural hazards and the fixed mapping require-

ment. The first such approach models individual pipeline stages in a straightforward fashion. The second adapts theory developed for hardware pipelines, so as to model only the function units and thus results in improved efficiency.

- Both methods make use of the fact that the mapping constraint is an instance of the *circular-arc graph coloring* problem [15] and can be represented by integer linear constraints and smoothly integrated in the overall ILP framework.
- We have implemented two variants of both methods and have performed tests on 415 unique loops from a variety of standard benchmarks: `SPEC92`, the `NAS kernels`, `linpack`, and the `livermore loops`. Our experiments were conducted for two target arhitectures whose function units contain moderate to high degree structural hazards. All four of our variations did well, with the best finding a rate-optimal schedule for 65% of the loops. This compares favorably with a leading heuristic, Huff's *Slack Scheduling*—our ILP approaches found a schedule with smaller initiation interval for over 50% of the loops, with a mean improvement of almost 30%.
- We found that reusing pipeline stages—and thus adding hazards—results in only a 10% drop in performance, while permitting significant savings in area. Alternatively, their smaller size could allow more such function units, thus boosting performance.

Register constraints can also be integrated in this framework as demonstrated in [4, 7, 12, 22]. A variety of techniques have been proposed which essentially convert loops with conditionals into basic blocks. Among these are `if-conversion` [3], `hierarchical reduction` [20], and `trace scheduling` [9]. The output of any of these techniques can be used as input to our proposed framework. However, as both register constraints and conditionals are orthogonal to the main thrust of our work, we do not dwell further upon them.

The rest of this paper is organized as follows. In Section 3 we use examples to motivate our approach to the scheduling and mapping approach for *unclean pipelines*. Section 4 provides a brief review of earlier work which used an ILP framework to software pipeline loops for target architectures with *clean* pipelines. It then proposes extensions to that framework so as to handle structural hazards. Finally it shows how the mapping problem can be solved by introducing coloring into the ILP formulation. Section 5 introduces an alternative mechanism for software pipelining in the presence of hazards. As mentioned earlier, this alternative uses theory developed for hardware pipeline control. Experimental results using 415 unique loop kernels are reported in Section 6. These results not only contrast our two proposed alternatives, but provide a comparison to *Slack Scheduling* as well. We survey related work in Section 7 and conclude in Section 8.

2. BACKGROUND AND MOTIVATION

Software pipelining has been proposed as an efficient method for loop scheduling for high-performance VLIW and superscalar architectures. Software pipelining derives a static parallel schedule—a periodic pattern—that overlaps instructions from different iterations of a loop body. The performance of a software-pipelined schedule is measured by the *initiation rate* of successive iterations.

The problem of finding, under resource constraints, an optimal schedule (i.e., with the fastest possible or "rate-optimal" initiation rate) is NP-hard. Various heuristic solution methods [11, 18, 28, 35] have been proposed to solve the problem efficiently. In this paper,

we are interested in solution methods based on ILP. In contrast to heuristic scheduling algorithms which find approximate (or suboptimal) solutions, ILP methods are based on a formulation where an objective optimality function is specified.

Since finding an optimal schedule is NP-hard, it is of course also true that finding an optimal solution to an ILP problem is NP-Hard. Hence the use of such a method in a practical compiler may be questioned. However, we feel that a clearly stated optimality objective in the problem formulation is important. A given loop is likely to have many good schedules from which to choose, and optimality criteria are essential to guide the selection of the best ones. It is also useful to compiler designers to establish an optimal bound using our ILP-based methods so as to compare and improve their heuristic methods of software pipelining [30]. Furthermore, such an approach can be provided as a compiler option for those users who want to explore the best possible schedules for their (performance-critical) code on a given machine even if this may mean a substantially longer compile time. Likewise, such an approach can be useful in hardware synthesis where the one-time cost of finding an optimal schedule is borne over many chips and uses of those chips. Finally, a large number of loop bodies are repeated in many programs. For example, the 415 unique loops on which we report here were culled from 1008 total loops. This being the case, it may be feasible to maintain a database of loops and their optimal schedules and make use of that in practical compilers.

We note that significant progress has been made in developing efficient ILP algorithms. We believe that ILP methods for software pipelining should take advantage of such developments and further improve their performance. In particular, analyzing the structure of the constraints in the ILP approach to the software pipelining problem appears to be an important direction to pursue, although it is beyond the scope of this paper.

In the rest of this paper, we discuss how handle hardware pipelines with structural hazards can be handled in an ILP framework.

3. SCHEDULING PIPELINES WITH STRUCTURAL HAZARDS: BASICS

In this section, we highlight the issues encountered in formulating the software pipelining problem in the presence of structural hazards. A structural hazard is caused by one or more stages of the pipeline being used at multiple times during the execution of an instruction. For example, a **shifter** may be used to normalize inputs at the start of a floating point divide and be reused again at the end of the divide to normalize the output.

3.1. Motivating Example

We use the simple example loop in Fig. 1 as a running example throughout. Both C language and instruction level representations are given in Fig. 1b while the data dependence graph (DDG) is depicted in Fig. 1a. Note that `vr33` is the loop index, while `vr32` is the loop invariant size of each element in the array being accessed.

In this paper we focus on *modulo scheduling*, which, given a loop, tries to find a software pipelined schedule with a fixed initiation interval **II**, i.e., a schedule which initiates a new loop iteration every **II** cycles. Given a fixed **II**, we can express in linear form, the time at which instruction x executes in each iteration j:

$$j \cdot \mathbf{II} + t_x.$$

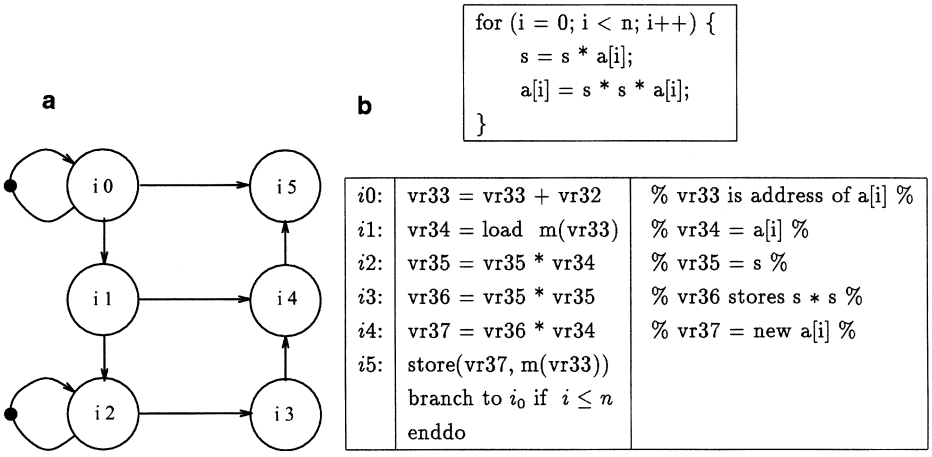

FIG. 1. An example loop. (a) Dependence graph and (b) program representation.

The value of t_x is constant and reflects that time at which instruction x first executes. The reciprocal of **II** is the *initiation rate* of the schedule. For more background information on linear scheduling, readers are referred to the survey paper by Rau and Fisher [25]. Clearly, the smaller the initiation interval **II**, the better the schedule.

Luckily there are lower bounds for **II** that turn out to be quite tight in practice. Modulo scheduling normally begins by trying to find a schedule at the lower bound value of **II**. If no schedule is found—either because none exists or because the modulo scheduling heuristic failed to find it—**II** is incremented by one. This continues until a schedule is found. How then is a lower bound for **II** obtained? In two steps [5, 18, 20, 25, 26]:

1. Loop-carried dependences (or recurrences) yield one lower bound, **RecMII**, on **II**. The value of **RecMII** is determined by the critical (dependence) cycle(s) in the loop [29]. Specifically

$$\textbf{RecMII} = \left\lceil \frac{\text{sum of instruction execution times}}{\text{sum of dependence distances}} \right\rceil$$

along the critical cycle(s). For the DDG in Fig. 1a, **RecMII** = 3 corresponding to the self loop at instruction i_2. Instruction i_2, a **Multiply**, has latency 3, and the single dot on the arc indicates that the value produced in this iteration is used in the immediately following iteration for a dependence distance of 1.

2. Another lower bound **ResMII** on **II** is enforced by resource constraints. Consider an architecture consisting of three **integer units** with a latency 1, two **FP units** with a latency 3, and two **load/store unit** with a latency 3. Further, assume that the **FP** and **load/store units** have structural hazards. In order to make the discussion simple, assume both the FP unit and the load/store unit to be 3-stage pipelines with the following simple reservation tables [19]:

Reservation Table for **FP unit**

	Time steps		
	0	1	2
Stage 1	x		
Stage 2		x	
Stage 3		x	x

Reservation Table for **load/store unit**

	Time steps		
	0	1	2
Stage 1	x		x
Stage 2		x	
Stage 3			x

As can be seen, the reservation table indicates the times at which each pipeline stage is used during the execution of an instruction. If a particular function unit has s pipeline stages and a latency of d, then the reservation table has s rows and d columns.

Consider (1) there are N_r operations in the loop that use function unit type r (e.g., there are three **FP** instructions in the example loop), (2) there are F_r units of type r in the architecture (in our architecture, there are two **FP units**), and (3) $d_{\max}(r)$ is the maximum number of time steps any stage of a function unit of type r is used in executing an instruction. For example, $d_{\max}$(**FP unit**) is 2 since stage 3 is used in two cycles. It can than be seen that the number of available function units imposes a lower bound **ResMII** on **II**,

$$\textbf{ResMII} = \max_r \left\lceil \frac{d_{\max}(r) * N_r}{F_r} \right\rceil,$$

where the maximum is taken over all function unit types r. In our example the **ResMII** bound is given by both the **FP unit** and the **load/store unit**. That is,

$$\textbf{ResMII} = \max\left(\left\lceil \frac{1}{3} \right\rceil, \left\lceil \frac{2*3}{2} \right\rceil, \left\lceil \frac{2*2}{2} \right\rceil\right) = \max(1,\ 3,\ 2) = 3.$$

A lower bound on the initiation interval **MII** is the maximum of **ResMII** and **RecMII**. However there may or may not exist a schedule with a period **MII** satisfying the given resource constraints. $\textbf{II}_{\min}$ (real-minimum-**II**) is the minimum period for which a schedule *exists*. Compare $\textbf{II}_{\min}$ to $\textbf{II}_{\text{act}}$ (or simply **II**) which is used to denote the initiation interval for which a schedule was *actually found* by the scheduling method. In short,

$$\textbf{MII} \leq \textbf{II}_{\min} \leq \textbf{II}.$$

A better initiation rate for the loop may be obtained by unrolling the loop a number of times. In such cases, the user may elect to unroll the loop prior to software pipelining. However for a fixed amount of unrolling, a schedule with initiation interval $\textbf{II}_{\min}$ is indeed optimal. Finally, we observe that most modern architectures have a separate branch unit which can function in parallel with other units. Since we are dealing with single basic block loops, unrolling is not necessary to minimize the overhead due to looping control.

Consider Schedule A in Table 1 for the DDG in Fig. 1a. The schedule is obtained from the linear schedule form $\textbf{II} \cdot i + t_s$, with $\textbf{II} = 3$, $t_{i0} = 0$, $t_{i1} = 1$, $t_{i2} = 4$, $t_{i3} = 8$, $t_{i4} = 12$, and $t_{i5} = 15$. Table 1 shows when an operation is initiated.

Table 1
Schedule *A* for Pipelines with Structural Hazards

	Time steps																	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Iteration = 0	i_0	i_1			i_2				i_3				i_4			i_5		
Iteration = 1				i_0	i_1			i_2				i_3				i_4		
Iteration = 2							i_0	i_1			i_2				i_3			
Iteration = 3										i_0	i_1			i_2				i_3
Iteration = 4													i_0	i_1			i_2	
Iteration = 5																i_0	i_1	

The resource requirement for a software pipelined schedule is modeled using the *modulo reservation table* (*MRT*) [18, 25, 28] where each stage of a function unit with structural hazard needs to be modeled as a resource. The MRT corresponding to the stages of the **FP Multiply Unit** is depicted in Table 2.

In order to verify that Schedule *A* satisfies the resource constraints, we need to check the resource requirement for each type of function unit. We concentrate only on the repetitive pattern since the resource requirement in the prolog and epilog is less than for any periodic pattern. Further, we concentrate only on the **FP unit**. The requirements for **integer** and **load/store units** can be estimated in a similar fashion. The resource usage for each stage of the **FP unit** can be derived from Table 1 and the corresponding rows in the **FP** reservation table. For example, in stage 2 of Table 2, the i_2 entry denotes the fact that i_2 from iteration 4 is using stage 2 at time step 17 (or equivalently time step 2 in the repetitive pattern).

Note that there is no overlap in the usage of stages 1 and 2 of the **FP** pipeline. However, stage 3 is used at most by two instructions in each time step. Since there are 2 **FP units** it looks like this is a valid schedule. However, as noted in the Introduction, some architectures or microarchitectures require a fixed mapping of operations to function units. For such architectures Schedule *A* has a problem. To illustrate the problem, assume

Table 2
Resource Usage Tables for the FP Unit

	Stage 1			Stage 2			Stage 3		
	Time Steps			Time Steps			Time Steps		
	15	16	17	15	16	17	15	16	17
Iter = 1	i_4				i_4			i_4	i_4
Iter = 2				i_3			i_3	i_3	
Iter = 3			i_3				i_2		
Iter = 4		i_2				i_2			i_2

that we assign instruction i_4 to **FP unit**-1 and instruction i_3 to **FP unit**-2. Instruction i_2 cannot be assigned to either of these function units since **FP unit**-2 is free only in time step 17 and **FP Unit**-1 is free only in time 15. This means that Schedule *A* requires instruction i_2 to migrate from one function unit to another during the course of execution, which is impossible. Hence the Schedule *A* is infeasible. In fact there exists no schedule for our example loop with **II** = 3 such that fixed function unit assignments can be made.

The second difficulty in instruction scheduling for function units with structural hazards is somewhat subtle and arises because an instruction executes exclusively on one pipeline. Architectures having completely pipelined or nonpipelined function units do not face the problem.

Consider Fig. 2. For a particular schedule of instructions W, X, Y, and Z, Fig. 2 depicts the resource usage tables for the **add** and **shift** stages of an **FP add unit**. One estimate

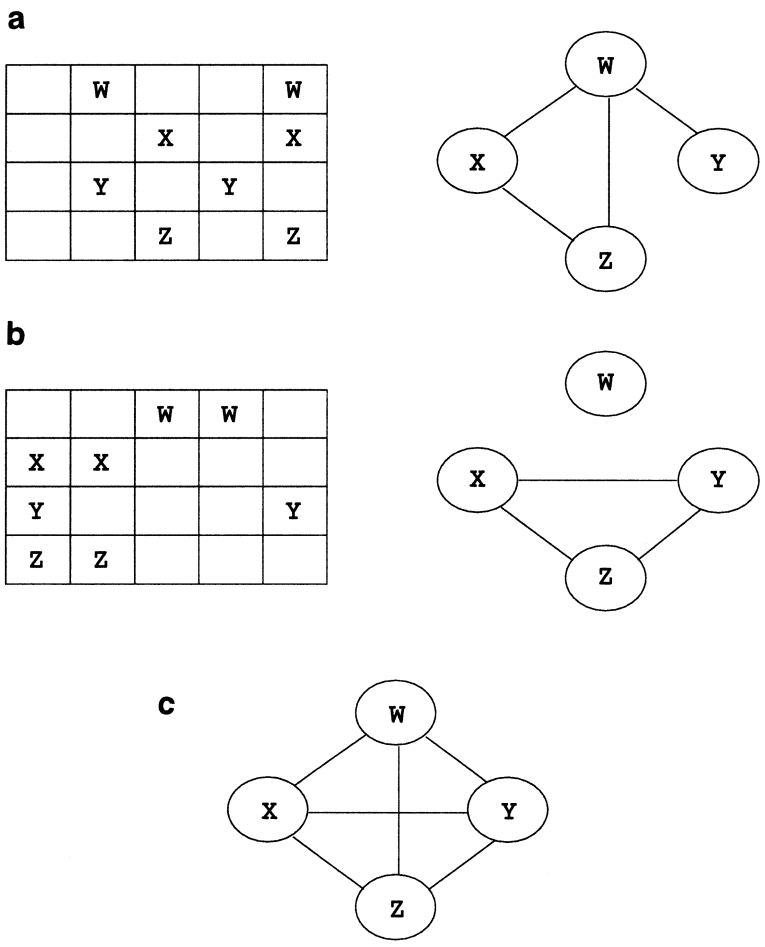

FIG. 2. Derivation of function unit requirements from resource usage tables. (a) **Shifter** usage and the corresponding interference graph, (b) **adder** usage and the corresponding interference graph, and (c) combined interference graph.

of the number of **FP add units** needed can be obtained by adding the columns of the *resource usage table* of each stage. The **shifter** is used by at most three operations at any given time (at offset 4), while the **adder** is also used by at most three operations (at offset 0). The question now is, can a schedule with resource usage as depicted in Fig. 2 be supported with three **FP add units**? In other words, can instructions `W`, `X`, `Y`, and `Z` share resources in such way that three **FP add units** suffice?

Interference graphs can be used to identify which instructions cannot share the same resource. Figure 2 depicts interference graphs for both the **shifter** and **adder** stages. As is clear from the *resource usage table*, instruction `X` interferes with instruction `Z` at offset 2 and with instruction `W` at offset 4. Thus the interference graph for the **shifter** contains an edge `XZ` and an edge `XW`. Although `X` and `Y` never use the **shifter** at the same offset, they both use the **adder** at offset 0 resulting in edge `XY` in the **adder** interference graph. Since the interferences from both stages must be observed, a new interference graph must be created containing all the edges from interference graphs of the individual stages. This new interference graph, the combined interference graph, is also depicted in Fig. 2. Clearly, every instruction interferes with every other instruction, and as a consequence, four **FP add units** would be needed for this schedule.

The two problems just outlined illustrate that in order to obey fixed function unit assignment—that a single instruction uses stages from the same function unit as well as uses the same function unit in all iterations—a scheduler must simultaneously solve *both* the *scheduling* problem (when instructions are initiated) and the *mapping* problem (to which function unit instructions are assigned). *This paper introduces two approaches, both of which perform the required scheduling and mapping. Furthermore, both do so optimally.* Formally,

Problem 1. Given a DDG and an architecture with structural hazards, construct a schedule that has the shortest initiation interval, using only the available resources and mapping each instruction x to the same function unit $\mathrm{FU}(x)$ in every iteration.

3.2. Our Approach

One important observation we have made is to identify the function unit assignment problem as a circular-arc coloring problem [15]. Thus we formulate the mapping problem as a graph coloring problem using integer linear constraints. The resource constraints in the scheduling problem can be expressed in two different ways. The first approach extends our earlier work on expressing resource constraints for simple resource usage (for pipelined function units) [13] and models resource requirements using *resource usage tables*.

The second approach uses a more elegant formulation for resource constraints, borrowing ideas from classical pipeline theory. To be precise, it makes use of the notion of *forbidden* and *permissible* latencies in modeling resource usage. Using them models different stages of a pipeline unit as a single resource. This is in contrast to most of the existing heuristic-based software pipelining methods [2, 11, 16, 18, 20, 21, 25, 26, 28, 35, 36] which model different stages of a pipeline as different resources. It should be noted here that it is the formulation of the mapping problem that has facilitated the use of *forbidden* latencies for formulating the rate-optimal scheduling problem.

Next in Sections 4 and 5 we develop the two formulations.

4. FORMULATION 1: DIRECT SCHEDULING

In this section we develop our first formulation. This formulation builds on our earlier work for clean pipelines [13]. For clarity of exposition, details of the earlier formulation are briefly reviewed in Section 4.1. Section 4.2 then describes the scheduling portion of our first approach. Section 4.3 discusses how this approach is modified to guarantee a fixed mapping of instructions to function units. Combining the constraints from Sections 4.2 and 4.3 yields a unified formulation for scheduling and mapping.

4.1. ILP Formulation for Clean Execution Units

We now briefly review our earlier work on ILP scheduling formulations for clean pipelines [13].

As outlined in Section 3, our ILP scheduling is a form of modulo scheduling. Beginning with $\mathbf{II} = \mathbf{MII}$, we attempt to find a schedule for a fixed initiation interval **II**. If the ILP solver finds no such schedule at **II**, there is no such schedule and **II** is incremented by 1. This process continues until a schedule is found—in practice at or very close to **MII**. Since **II** is constant at each attempt, the time at which each operation executes has the linear form $j \cdot \mathbf{II} + t_x$, where j is the iteration number, 0, 1, 2, ..., and t_x is the time at which instruction x executes in iteration 0 of the loop. The values t_x are chosen such that the data dependences from the DDG are satisfied. In particular for each dependence edge, xy in the DDG, Reiter showed that the following linear constraint on t_x values must hold,

$$t_y - t_x \geq d_x - \mathbf{II} \cdot m_{xy}, \tag{1}$$

where d_x is the latency of instruction x and m_{xy} is the number of iterations separating x and its consumer instruction y. ($m_{xy} = 0$ for most edges in most DDGs.)

In order to represent resource constraints in a linear form, our earlier work defined a pair of values k_x and o_x for each t_x, such that

$$t_x = \mathbf{II} \cdot k_x + o_x, \tag{2}$$

where o_x is an integer *offset* $0 \leq o_x < \mathbf{II}$. All instructions with the same offset value o_x begin execution at the same time. For clean pipelines, the goal was to make sure that no more instructions began at once than existed function units on which to execute. To this end, our previous work expressed each o_x in terms of a set of 0–1 integer variables $a_{\tau,x}$:

$$o_x = [a_{0,x},\ a_{1,x},\ \ldots,\ a_{(\mathbf{II}-1),x}] \times [0,\ 1,\ \ldots,\ (\mathbf{II}-1)]^{Transpose}. \tag{3}$$

The variables $a_{\tau,x}$ were 1 if and only if instruction x was scheduled to begin at time step τ in the steady-state repetitive pattern of the loop. Otherwise $a_{t,x} = 0$. Since each instruction x must be scheduled for execution at exactly one time τ in the repetitive pattern, we obtained the condition for all instructions x in the loop:

$$\sum_{\tau=0}^{\mathbf{II}-1} a_{\tau,x} = 1. \tag{4}$$

By replacing o_x in Eq. (2) with the right side of Eq. (3), the times t_x can be expressed in terms of k_x and the $a_{\tau,x}$ variables. When all instructions x in the loop are considered, this yields the matrix equation:

$$\mathcal{T} = \mathbf{II} \cdot \mathcal{K} + \mathcal{A}^{\text{Transpose}} \times [0,\ 1,\ \ldots,\ \mathbf{II}-1]^{\text{Transpose}}, \tag{5}$$

where $\mathcal{K}$ and $\mathcal{T}$ are N-element vectors, $\mathcal{A}$ is a $\mathbf{II} \times N$ matrix, and N is the number of instructions in the loop. That is,

$$\mathcal{T} = \begin{bmatrix} t_0 \\ t_1 \\ \vdots \\ t_{N-1} \end{bmatrix},\ \mathcal{K} = \begin{bmatrix} k_0 \\ k_1 \\ \vdots \\ k_{N-1} \end{bmatrix}$$

and

$$\mathcal{A} = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,(N-1)} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,(N-1)} \\ \vdots & \vdots & & \vdots \\ a_{(\mathbf{II}-1),0} & a_{(\mathbf{II}-1),1} & \cdots & a_{(\mathbf{II}-1),(N-1)} \end{bmatrix}. \tag{6}$$

Each row in the $\mathcal{A}$ matrix represents an offset, $0,\ 1,\ \ldots,\ \mathbf{II}-1$, in the steady-state loop execution. Each column represents one of the N instructions from the loop. Thus summing the elements of row τ yields the number of instructions from the loop which begin at offset τ. The row with the largest sum gives the number of **instruction issue units** needed (and with a straightforward extension, the number of each type of function unit as well.) If a given architecture has R **issue units**, then the sum in all rows must be less than R. Formally, this gives the constraint

$$R \geq \sum a_{\tau,x}, \qquad \forall\, \tau \in [0,\ \mathbf{II}-1].$$

With this we have sufficient linear constraints to guarantee that instruction dependences are obeyed and that function unit usage does not exceed availability—*assuming that none of the function units have any structural hazards*:

[ILP Formulation for Clean Pipelines]

$$\sum a_{\tau,x} \leq R,\ \forall\, r \in [0,\ h-1] \text{ and } \forall\, \tau \in [0,\ \mathbf{II}-1]$$

$$\mathcal{T} = \mathbf{II} \cdot \mathcal{K} + \mathcal{A}^{\text{Transpose}} \times [0,\ 1,\ \ldots,\ \mathbf{II}-1]^{\text{Transpose}} \tag{7}$$

$$\sum_{\tau=0}^{\mathbf{II}-1} a_{\tau,x} = 1 \qquad \forall\, x \in [0,\ N-1]$$

$$t_y - t_x \geq d_x - \mathbf{II} \cdot m_{xy} \qquad \forall\, \text{Edges } (x,\ y) \text{ in DDG} \tag{8}$$

$$t_x \geq 0,\ k_x \geq 0,\ \text{and } a_{\tau,x} \geq 0 \text{ are integers} \qquad \forall\, x \in [0,\ N-1],\ \forall\, \tau \in [0,\ \mathbf{II}-1] \tag{9}$$

If desired, some weighted sum of function unit usage could be minimized. Alternatively, the objective function could be to minimize register usage [4, 7, 22] while obeying function unit constraints.

4.2. ILP Formulation for Function Units with Structural Hazards

To determine resource requirements when function units have structural hazards, we need to know not just when each instruction is *initiated*, but also at what time steps each stage is required. In our motivating example, stage 3 of the **FP unit** was required for two time steps. Though this is not directly reflected in the $\mathcal{A}$ matrix, it is possible to derive it from the $\mathcal{A}$ matrix. For this purpose, we define the *usage* matrix using the $\mathcal{A}$ matrix and the reservation table [19] of the function unit.

As illustrated in Section 3.1, the reservation table of a function unit with s stages and latency d has s rows and d columns. We change our nomenclature slightly here so that the reservation table marks times at which a stage is used with a `1` instead of an `X`. All other entries in the reservation table are `0`.

For simplicity's sake we assume that a function unit of a particular type r has a single reservation table describing its use of various stages. This assumption is only to simplify discussion and is not a limitation of the formulation, as will be discussed in Section 4.4. As described in Section 3.1, the maximum number of time steps over any stage of a pipeline is used (not necessarily continuously) is $d_{\max}$. Clearly, for fixed function unit assignment, $\mathbf{II} \geq d_{\max}$. Further we assume that no stage in a pipeline is used by an instruction at two time steps separated by multiples of **II**. This requirement prevents instances of the same instruction from different iterations using the same resource. Previous articles made a similar assumption known as the *modulo scheduling constraint* [7, 16, 25]. In Section 4.4 we discuss how to relax this modulo scheduling constraint and the associated cost.

From the reservation table and the initiation interval **II**, we obtain a *modified reservation table*. The process for this is straightforward:

- For each function unit whose execution time $d = \mathbf{II}$, the modified reservation table is the same as the reservation table.
- For each function unit whose execution time $d < \mathbf{II}$, $(\mathbf{II} - d)$ zero columns are added to the reservation table.
- For each function unit whose execution time $d > \mathbf{II}$, entries from the reservation table are placed in the modified reservation table at their offset modulo **II**.

The modulo scheduling constraint guarantees that multiple uses of a stage are not mapped onto a single time step in the modified reservation table. To illustrate the modified reservation table, consider the **FP unit** whose reservation table is in Section 3.1, and which has execution time $d = 3$. Figure 3 shows its modified reservation table when $\mathbf{II} = 4$ and $\mathbf{II} = 2$.

The sth row in the modified reservation table of function unit type r specifies the usage of stage s. Let us denote this row by RT_r^s, a vector of length **II**. An entry $RT_r^s[l]$ is 1 if stage s is required l time steps after the initiation of an instruction (`mod` **II**). Using this vector, and the $\mathcal{A}$ matrix we can define the resource usage matrix $\mathcal{U}_r^s$ for each stage s of a pipeline r. The xth column of $\mathcal{U}_r^s$ represents the use of stage s by instruction x in various time steps. An instruction x uses stage s at time step τ if (i) the instruction was initiated at time step $(\tau - l)$ `mod` **II** and (ii) stage s is required l time steps after the

a

	Time Steps			
	0	*1*	*2*	*3*
Stage 1	1	0	0	0
Stage 2	0	1	0	0
Stage 3	0	1	1	0

b

	Time Steps	
	0	*1*
Stage 1	1	0
Stage 2	0	1
Stage 3	1	1

FIG. 3. Modified reservation tables. (a) $\mathbf{II} = 4$ and (b) $\mathbf{II} = 2$.

initiations of an operation. These two conditions correspond to having $\mathcal{A}[((\tau - l) \bmod \mathbf{II}), x] = 1$ and $RT_r^s[l] = 1$. Thus each element of $\mathcal{U}$ can be defined mathematically as

$$\mathcal{U}_r^s[\tau, x] = \sum_{l=0}^{(\mathbf{II}-1)} a_{((\tau-l) \bmod \mathbf{II}), x} \cdot RT_r^s[l], \qquad \forall \tau \in [0, \mathbf{II} - 1],$$
$$\forall s \in [0, (s_r - 1)], \text{ and } i \in \mathcal{I}(r), \tag{10}$$

where $\mathcal{I}(r)$ represents the set of instructions that execute in function unit type r. Note that the mod operator is used only in setting up the set of equations and need not be dealt with in finding a solution to the equations. Thus the equations specified by (10) are linear.

To understand the resource usage matrix more clearly, let us again consider our motivating example. In Schedule A, the period $\mathbf{II} = 3$. The $\mathcal{A}$ matrix for this schedule is

$$\mathcal{A} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Since $\mathbf{II} = 3$, the modified reservation table for the **FP unit** is same as the original reservation table. The $\mathcal{U}$ matrices for the three stages of the **FP unit** are as follows:

$$\mathcal{U}_{\mathbf{FP}}^{s1} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \qquad \mathcal{U}_{\mathbf{FP}}^{s2} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$
$$\mathcal{U}_{\mathbf{FP}}^{s3} = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}.$$

Note that the matrix $\mathcal{U}_{\mathbf{FP}}$ is defined only for instructions i_2, i_3, and i_4 which execute in the **FP unit**. Similarly $\mathcal{U}_{\mathbf{load/store}}$ is defined only for instructions i_1 and i_5.

The resource requirements of a schedule can be calculated by adding the elements of each row of the usage matrix. From the resource usage matrix, it can be seen that the stage $s3$ of the **FP unit** is required by two instructions in all time steps. Thus at least two **FP units** are required to support Schedule A. Thus the resource requirement of a schedule can be formally specified using the $\mathcal{U}$ matrix as

$$R_r^s(\tau) = \sum_{i \in \mathcal{I}(r)} \mathcal{U}^s[\tau, x] = \sum_{i \in \mathcal{I}(r)} \sum_{l=0}^{(\mathbf{II}-1)} a_{((\tau-l) \bmod \mathbf{II}), x} \cdot RT_r^s[l], \tag{11}$$

where $R_r^s[\tau]$ represents the resource requirement for stage s of function unit type r at time τ in the repetitive pattern. Thus the number of r-type function units required for the schedule is

$$R_r \geq R_r^s[\tau] \qquad \forall s \in [0,\ (s_r - 1)] \text{ and } \forall \tau \in [0,\ \mathbf{II} - 1].$$

This can be rewritten as

$$R_r \geq \sum_{i \in \mathcal{I}(r)} \sum_{l=0}^{(\mathbf{II}-1)} a_{((\tau - l) \bmod \mathbf{II}),\, x} \cdot RT_r^s[l],$$
$$\forall \tau \in [0,\ \mathbf{II} - 1],\ \text{and } s \in [0,\ (s_r - 1)]. \tag{12}$$

Thus we have a linear constraint on the number of function units with hazards which are available for software pipelining a loop. As noted in Section 4.1, we can add to this any desired objective function, such as one minimizing registers subject to function unit constraints. Alternatively, we may wish to minimize power consumption by minimizing the number of function units used. We can achieve this goal by minimizing the weighted sum of R_r for each function unit type r. If the weight associated with function unit type r is C_r. then the objective function is

$$\text{minimize } \sum_r C_r \cdot R_r.$$

As described in the Section 3.1, Eqs. (11) and (12) are insufficient for architectures which require a fixed mapping of instructions to function units that is constant from iteration to iteration. In the following section we remedy this shortcoming.

4.3. Coloring Formulation for Fixed Function Unit Assignment

Consider Schedule *A* that was depicted in Table 1. The use of stage 3 of the **FP unit** in the steady-state loop kernel is depicted in Fig. 4a, mapping time steps 15, 16, and 17 to 0, 1, and 2, respectively. Figure 4b shows the overlap of resource usage by instructions i_2, i_3, and i_4. Note that the use of stage 3 of the function unit by i_2 wraps around from time 2 to 0. This is a problem. At time 2, i_2 begins executing on the function unit that was used by i_3 at times 0 and 1. Since each instruction is supposed to use the same function unit on every iteration, this causes a problem at time 0, when i_2 is still executing on the function unit needed by i_3. The problem is that Eq. (11) notes only the number of function units in use at one time, i.e., the number of solid horizontal lines present at each of the three time steps in Fig. 4b.

This problem bears a striking similarity to the problem of assigning variables with overlapping lifetimes to different registers. In particular, it is a *circular-arc coloring* problem [15]. We must ensure that the two fragments corresponding to i_2 get the same color, a fact represented by the dotted arc in Fig. 4b. In addition the arcs of i_2 overlap with both i_3 and i_4, meaning i_2 must have a color different than both. Similarly i_3 and i_4 must have different colors from each other.

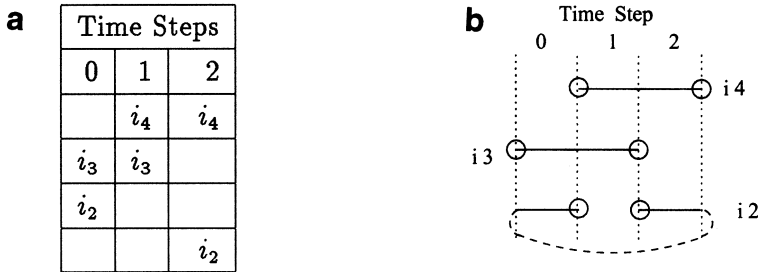

FIG. 4. Resource usage of stage 3 of **FP unit**.

The trick is how to express the coloring problem in a format compatible with the rest of the ILP formulation. One natural way of representing this coloring is

$$|c_x - c_y| \geq \frac{\mathcal{U}_r^s[\tau, x] + \mathcal{U}_r^s[\tau, y] - 1}{2} \qquad \forall s \in [0, s_r - 1], \text{ and } \forall x, y \in \mathcal{I}(r) \quad (13)$$

since $\mathcal{U}_r^s[\tau, x]$ and $\mathcal{U}_r^s[\tau, y]$ represent the use of a function unit by instructions x and y, respectively, at time step τ. Here the colors c_x and c_y, which represent the colors for instructions x and y, are positive integers. The colors c_x and c_y must be different if instructions x and y overlap during loop execution. That is, if both $\mathcal{U}_r^s[\tau, x]$ and $\mathcal{U}_r^s[\tau, y]$ are 1 at some time step τ. In such cases the value of the right-hand side at time τ is $1/2$, thus forcing c_x and c_y to be different. Conversely if only one of $\mathcal{U}_r^s[\tau, x]$ and $\mathcal{U}_r^s[\tau, y]$ is 1 at time τ or if both are 0, then the right-hand side is 0 or $-(1/2)$, thus allowing c_x and c_y to be the same (if x and y do not overlap at any other time).

Unfortunately, absolute value is not a linear operation, so this constraint does not fit with the rest of our ILP framework. To overcome this problem, we adopt an approach outlined by Hu [17]. We introduce a set of $w_{x,y}$ integers, 0–1 variables, with one such variable for each pair of nodes using the same type of function unit. Roughly speaking these $w_{x,y}$ variables represent the sign of $c_x - c_y$, as discussed in Appendix A. Using them, Eq. (13) can be represented with the equations

$$c_x - c_y \geq \frac{\mathcal{U}_r^s[\tau, x] + \mathcal{U}_r^s[\tau, y] - 1}{2} - N \cdot w_{x,y} \quad (14)$$

$$c_y - c_x \geq \frac{\mathcal{U}_r^s[\tau, x] + \mathcal{U}_r^s[\tau, y] - 1}{2} - N \cdot (1 - w_{x,y}) \quad (15)$$

$$1 \leq c_k \leq N \qquad \forall k \in [0, N - 1], \quad (16)$$

where N, the number of nodes in the DDG, is an upper bound on the number of colors.

THEOREM 4.1. *Equations* (14)–(16) *require that two nodes x and y be assigned different colors* (*mapped to different function units*) *if and only if they overlap.*

For a proof, please see Appendix A.

Given this coloring formulation one could minimize the number of required colors (function units). That is

$$\min\Big(\min_{i \in \mathcal{I}(r)} c_x\Big)$$

for all nodes x using the function unit of type r. This objective function can be represented in a linear form as

$$\min Q$$

subject to

$$Q \geq c_x \qquad \forall x \in \mathcal{I}(r).$$

However, for our ILP formulation we do not minimize colors directly. Instead we require that there be at most as many function units of each type as colors. Hence,

$$R_r \geq c_x \qquad \forall x \in \mathcal{I}(r),\ \forall r \in [0,\ h-1]. \tag{17}$$

It can be seen that Eqs. (14)–(17) together model the resource constraints. In the presence of Eqs. (14)–(17), Eq. (12) is redundant and hence can be removed. (Subsequently, in our experiments we reintroduce the resource constraints expressed using the usage matrix (Eqs. (12)) and study their impact on the time for solving the ILP formulation.) The complete ILP formulation for function units with structural hazards is shown in Fig. 5.

[ILP Formulation for Pipelines with Hazards]

$$\text{minimize} \sum_{r=0}^{h-1} C_r \cdot Q_r$$

subject to

$$c_x \leq R_r \quad \forall x \in \mathcal{I}(r), \text{ and } r \in [0, h-1] \tag{18}$$

$$c_x - c_y \geq (\mathcal{U}_r^s[\tau, x] + \mathcal{U}_r^s[\tau, y] - 1)/2 - N \cdot w_{xy} \quad \forall x, y \in \mathcal{I}(r),\ \forall \tau \in [0, \mathbf{II}-1],\ \text{and } \forall r \in [0, h-1] \tag{19}$$

$$c_y - c_x \geq (\mathcal{U}_r^s[\tau, x] + \mathcal{U}_r^s[\tau, y] - 1)/2 - N \cdot (1 - w_{xy}) \quad \forall x, y \in \mathcal{I}(r),\ \forall \tau \in [0, \mathbf{II}-1],\ \text{and } \forall r \in [0, h-1] \tag{20}$$

$$\mathcal{U}_r^s[\tau, x] = \sum_{l=0}^{(\mathbf{II}-1)} a_{((\tau-l) \bmod \mathbf{II}),x} \cdot RT_r^s[l] \quad \forall \tau \in [0, \mathbf{II}-1], \forall x \in \mathcal{I}(r), \forall s \in [0, s_r - 1] \text{ and } r \in [0, h-1] \tag{21}$$

$$\mathcal{T} = \mathbf{II} \cdot \mathcal{K} + \mathcal{A}^{\text{Transpose}} \times [0, 1, \cdots, \mathbf{II} - 1]^{\text{Transpose}} \tag{22}$$

$$\sum_{\tau=0}^{\mathbf{II}-1} a_{\tau,x} = 1 \quad \forall x \in [0, N-1] \tag{23}$$

$$t_y - t_x \geq d_x - \mathbf{II} \cdot m_{xy} \quad \forall \text{Edges } (x, y) \text{ in DDG} \tag{24}$$

$$t_x \geq 0, k_x \geq 0, a_{\tau,x} \geq 0, c_x \geq 1, 0 \leq d_{x,y} \leq 1, \text{ and } \mathcal{U}_r^s[\tau, x] \geq 0 \text{ are integers} \quad \forall x \in [0, N-1], \forall \tau \in [0, \mathbf{II}-1], \forall s \in [0, s_r - 1] \tag{25}$$

FIG. 5. ILP formulation for pipelines with hazards.

4.4. Remarks

In this section we describe our approach when a reservation table does not obey the modulo scheduling constraint, i.e., when it uses the same stage at two or more times that are equal modulo **II**. Since $d_{\max} \leq$ **II**, it is possible to introduce delays in the pipelines [19] such that no stage is used at times that are equal modulo **II**. (Additionally, the hardware must be designed to allow such delay insertion.) However, delay insertion often increases the latency of a function unit and hence may affect **RecMII** which in turn affects **II**. The following approach can be used to choose the right **II** value.

(1) Start with the smallest **II** value such that **II** $\geq$ **MII**. (2) By introducing delays, make sure that the modulo scheduling constraint is satisfied by all function unit types r. (3) Once appropriate delays have been introduced in the pipe, recompute **RecMII**. Note that **ResMII** will not be affected by the introduction of delays. (4) If the new **RecMII** value is less than or equal to **MII**, schedule the loop for the current value of **II**. Otherwise, increment **II** by 1 and recompute the delays and repeat Steps 2–4. Following this approach—that of incrementing **II** by 1 rather than replacing **II** with **RecMII**—guarantees that there is no lower value of **II** for which a schedule exists under the given assumptions. Our experiments have shown that although delays were introduced in the pipeline in 11% of cases, in none did the introduction of delays increase **II**.

Finally, we describe how our formulation can handle function units with multiple reservation tables, e.g., a **floating point unit** may support many operations, such as **FP-add**, **FP-multiply**, and **FP-divide**, each having a different resource usage pattern. In this case, the resource usage table RT is associated with the instruction rather than the function unit. That is, for each instruction x, we use the RT_x^s which is determined from the appropriate reservation table.

5. FORMULATION 2: SCHEDULING USING FORBIDDEN LATENCIES

The ILP formulation developed in the previous section, like many heuristic software pipelining methods, models individual stages of a function unit as different resources. An elegant and efficient alternative is to make use of *forbidden latencies*, a concept originally developed to support efficient hardware control of pipelines [19]. The use of forbidden latencies helps in detecting resource conflicts and also allows modeling of each function unit as a single resource instead of a set of stages. In the following subsection we briefly review some definitions from hardware pipeline theory [19] and adapt them for software pipelining. In Section 5.2 we build on these definitions as well as the formulation of the *mapping* problem in Section 4.3 to develop a second framework for the software pipelining problem.

5.1. Forbidden Latency Set for Software Pipelined Schedules

We start with a set of definitions from [19].

DEFINITION 5.1. The time elapsed between two initiations made in a function unit is the **latency** between the two instructions.

DEFINITION 5.2. Two initiations are said to cause a **collision** if these two instructions attempt to use the *same stage* of a function unit at the *same time step*.

Since initiations causing collisions or resource conflicts should be forbidden, the latencies between those operations are called forbidden latencies.

DEFINITION 5.3. A latency f is said to be **forbidden** if there exists a pair of X marks separated by f columns in any row of the reservation table. Initiations made with such a latency f cause a collision.

DEFINITION 5.4. A **permissible latency** is one that does not cause a collision.

We adapt the definition of forbidden latency to account for the fact that instructions in a software pipelined schedule are executed repeatedly. Thus, all latencies are considered with a wrap-around. For example, if t_x and t_y are, respectively, the time steps at which instructions x and y are initiated, then latency between these two instructions is $(t_y - t_x)$ mod **II**. Further, we can also say that the next initiation of x is separated from the current initiation of y by $(t_x - t_y)$ mod **II**. Thus, for the software pipelined schedule, we require that both $(t_y - t_x)$ mod **II** and $(t_x - t_y)$ mod **II** be permissible. Hence we adapt the definition of forbidden latencies as follows. Further we use the modified reservation table (discussed in Section 4.2) rather than the original reservation table in defining forbidden latencies.

DEFINITION 5.5. A latency f is said to be forbidden if there exists a pair of X marks at time steps t and $(t + f)$ mod **II** in any row of the *modified reservation table*.

Note that if f is forbidden, then by the above definition, (**II** $- f$) is also forbidden. As an example, for the modified reservation table shown in Fig. 3a latency 1 and 3 are forbidden whereas latency 2 is permissible. Finally, we use the notation $\mathcal{F}$ to denote the set of forbidden latencies.

5.2. Alternative ILP Formulation

Let x and y be instructions that execute in the same function unit type r. As discussed in Section 4.3 we associate colors c_x and c_y with instructions that represent the function units on which they execute. If the latency f between the initiation of x and y is forbidden, then x and y must execute in different function units. That is, if $t_x - t_y$ mod **II** $= f$, then $c_x \neq c_y$. The modulo operation is nonlinear and hence is not directly useful for our integer program formulation. Fortunately, the modulo operation can be removed by transforming the t_x variables to o_x variables as in Eq. (2) in Section 4.1 and hence to the $a_{t,x}$ variables.

The initiation of instructions x and y are separated by a latency f, if both $a_{t,x} = 1$ and $a_{((\tau+f) \bmod \mathbf{II}),y} = 1$, for any $\tau \in [0, \mathbf{II} - 1]$. In order to establish the resource constraint for the software pipelined schedule, we need to establish that all pairs of instructions x, y that execute on the same function unit type execute in different function units (i.e., have different c_x, c_y values if the initiation of x and y are separated by a forbidden latency f. Otherwise c_x may be the same as c_y. Notice that this constraint is similar to the mapping constraint discussed in Section 4.3, which required x and y to be mapped

[ILP Formulation using Forbidden Latencies]

$$\text{minimize} \sum_{r=0}^{h-1} C_r \cdot Q_r$$

subject to

$$c_x \le R_r \quad \forall x \in \mathcal{I}(r), \text{ and } r \in [0, h-1] \tag{30}$$

$$c_x - c_y \ge (a_{\tau,x} + a_{((\tau+f) \bmod \mathbf{II}),y} - 1)/2 - N \cdot w_{xy} \quad \forall x, y \in \mathcal{I}(r), \ \forall \tau \in [0, \mathbf{II}-1], \text{ and } \forall f \in \mathcal{F}_r \tag{31}$$

$$c_y - c_x \ge (a_{\tau,x} + a_{((\tau+f) \bmod \mathbf{II}),y} - 1)/2 - N \cdot (1 - w_{xy}) \quad \forall x, y \in \mathcal{I}(r), \ \forall \tau \in [0, \mathbf{II}-1], \text{ and } \forall f \in \mathcal{F}_r \tag{32}$$

$$\mathcal{T} = \mathbf{II} \cdot \mathcal{K} + \mathcal{A}^{\text{Transpose}} \times [0, 1, \cdots, \mathbf{II}-1]^{\text{Transpose}} \tag{33}$$

$$\sum_{\tau=0}^{\mathbf{II}-1} a_{\tau,x} = 1 \quad \forall x \in [0, N-1] \tag{34}$$

$$t_y - t_x \ge d_x - \mathbf{II} \cdot m_{xy} \quad \forall (x,y) \in E \tag{35}$$

$$t_x \ge 0, k_x \ge 0, a_{\tau,x} \ge 0, c_x \ge 1, \text{ and } 0 \le d_{x,y} \le 1 \text{ are integers}$$
$$\forall x \in [0, N-1], \forall \tau \in [0, \mathbf{II}-1], \forall s \in [0, s_r - 1] \tag{36}$$

FIG. 6. Alternative ILP formulation using forbidden latencies.

to different colors if their resource usage overlaps. Hence the constraint $c_x \neq c_y$ if both $a_{\tau,x} = 1$ and $a_{((\tau+f) \bmod \mathbf{II}),y} = 1$ can be formulated as

$$c_x - c_y \ge \frac{a_{\tau,x} + a_{((\tau+f) \bmod \mathbf{II}),y} - 1}{2} - N \cdot w_{x,y}, \quad \forall \tau \in [0, \mathbf{II}-1] \text{ and } \forall f \in \mathcal{F}_r \tag{26}$$

$$c_y - c_x \ge \frac{a_{\tau,x} + a_{((\tau+f) \bmod \mathbf{II}),y} - 1}{2} - N \cdot (1 - w_{x,y}), \quad \forall \tau \in [0, \mathbf{II}-1] \text{ and } \forall f \in \mathcal{F}_r. \tag{27}$$

As before,

$$1 \le c_k \le N \qquad \forall k \in [0, N-1], \tag{28}$$

where N, the number of nodes in the DDG, is an upper bound on the number of colors. For each pair of instructions that execute in the same function unit type, there is one set of equations (Eqs. (26)–(28)). To enforce the resource constraint, we must ensure

$$c_x \le R_r, \qquad \forall x \in \mathcal{I}(r) \text{ and } \forall \text{ function unit type } r \tag{29}$$

where R_r is the number of function units required for the schedule in type r.

It can be seen that Eqs. (26)–(29) effectively model the resource constraints. Hence they can replace Eqs. (19)–(21) in the ILP formulation shown in Fig. 5. The revised ILP formulation is depicted in Fig. 6.

5.3. Remarks

In Section 6 we study how our two ILP formulations perform on a number of benchmark loops. We refer to the formulation in Section 4 as *ILP-Res* and the one here

that uses forbidden latencies as *ILP-Forb*. The resource constraints expressed by Eq. (12), though considered redundant, can be added to either of these formulations. The resulting formulations are referred to as *ILP-Res-Usage* and *ILP-Forb-Usage*. Though this inclusion increases the number of constraints in the ILP formulation, it may help in obtaining the solution for the ILP problem faster. We want to empirically study the effect of the inclusion of this constraints.

6. EXPERIMENTAL RESULTS

In this section we present results of our experiments involving the different ILP formulations. We considered an architecture with two **integer ALUs** and one each of **load**, **store**, **FP add**, **FP multiply**, and **FP divide units**. To extensively test the formulations, we considered two different configurations. In configuration **C1**, the pipelines have moderate structural hazards whereas in **C2**, there is a high degree of structural hazards. The reservation tables of the execution units for the two configurations are shown in Appendix B.

We ran our experiments on a set of 415 single-basic-block inner loops taken from `SPEC92` (`integer` and `floating point`), `linpack`, `livermore`, and the `NAS kernels`. We started off with DDGs for 1008 inner loops; however, further analysis revealed that out of these 1008 DDGs, only 415 were unique. Hence we concentrated only on the corresponding 415 loops. The DDGs for the loops were obtained by instrumenting a highly optimizing research compiler. Large loops generally contain sufficient parallelism within a single iteration to keep function units busy. Thus, most compilers limit the size of loops which are software pipelined. We followed suit and like [18] used loops with at most 64 instructions.

To solve the ILPs, we used the commercial program, *CPLEX*. In order to deal with the fact that our ILP approach can take a very long time on some loops, we adopted the following approach. *First*, we limited CPLEX to 3 min in trying to solve any single ILP; i.e., a maximum of 3 min was allowed to find a schedule at a given **II**. *Second*, initiation intervals from [**MII**, **MII** + 5] were tried if necessary. Using this approach and these parameters, we were able to obtain optimal solutions in the large majority of cases.

6.1. Performance of ILP Formulations

As mentioned in Section 5.3, we considered four different ILP formulations, *ILP-Res*, *ILP-Res-Usage*, *ILP-Forb*, and *ILP-Forb-Usage*. First we report the performance of the individual formulations. In the subsequent subsection we compare their performance.

First we report in Fig. 7 the quality of the schedules constructed. The quality is reported by the type of ILP formulation as well as by how far its **II** deviates from **MII**. For example, in Fig. 7a the *ILP-Res* method obtains schedules for configuration **C1** at **MII** in 229 of the benchmark loops. In Fig. 8 we report three characteristics of the benchmark loops. These are the geometric mean of (i) the number of nodes in the DDG, (ii) the number of edges in the DDG, and (iii) **II**.

For configuration **C1** with pipelines involving moderate structural hazards, the ILP approach found the optimal schedule at the lower bound, in 229–261 benchmark loops (roughly 55–63%). For configuration **C2**, the approach was successful in finding the optimal schedule in 45–53%. Due to the 3 min time limit imposed on the CPLEX solver,

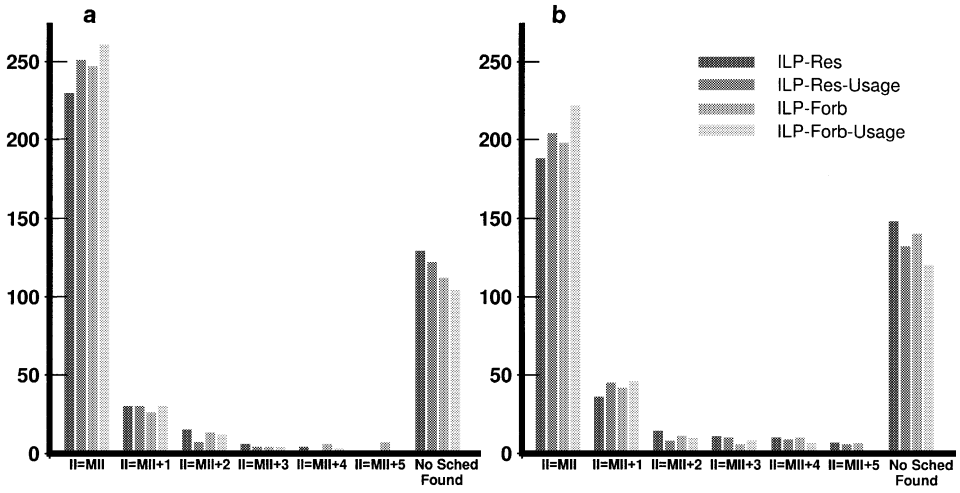

FIG. 7. Number of loops scheduled by **II** and method. (a) Config C1 and (b) Config C2.

in the remaining cases our method found a schedule at a **II** greater than **MII**. We do not know if there actually was a schedule at **MII** in these cases; i.e., we do not know if $\mathbf{II}_{\min} = \mathbf{MII}$—CPLEX's 3 min time limit expired without indicating whether or not a schedule exists at **MII**.

Next we investigated the degradation in **II** resulting from the additional structural hazards in the **C2** pipelines that are not in the **C1** pipelines. As can be seen in Table 3, the loss of performance is small—approximately 10% for all four scheduling approaches corresponding to an increase of approximately 1 in **II**. The performance of **C2** is all the stronger because the added reuse of pipeline stages should result in the use of substantially less area than is required by **C1**. Although we compared machines with the same number of function units for both **C1** and **C2**, the smaller size of **C2** function units may allow more of them on the chip, with a resultant increase in performance (reduction in **II**).

We measured the execution time of our scheduling method, henceforth referred to as *scheduling time*, on a *Sun/Sparc20* workstation. A histogram of the scheduling time as well as the geometric mean and median of the scheduling time are tabulated in Table 4. *ILP-Forb* and *ILP-Forb-Usage* generally performed best.

Another important consideration in evaluating these schedules is the number of registers they require. In all four formulations the schedule for more than 80% of the loops used fewer than 32 variable lifetimes, while the schedule for all the loops had fewer than 64 variable lifetimes. These numbers do not include any register sharing; i.e., they ignore the fact that some of the variable lifetimes might be able to share the same physical register. In addition, they lump together both integer and floating point values. As discussed in Section 4.1, we can easily include register requirements in the ILP framework. However, for simplicity of discussion, we chose not to.

6.2. Comparison of the Four ILP Formulations

In order to make a fair comparison of the four ILP formulations, in this section, we consider only the benchmark loops for which all four formulations obtained a schedule. There are totally 279 (out of 415) such benchmarks for **C1** and 251 for **C2**.

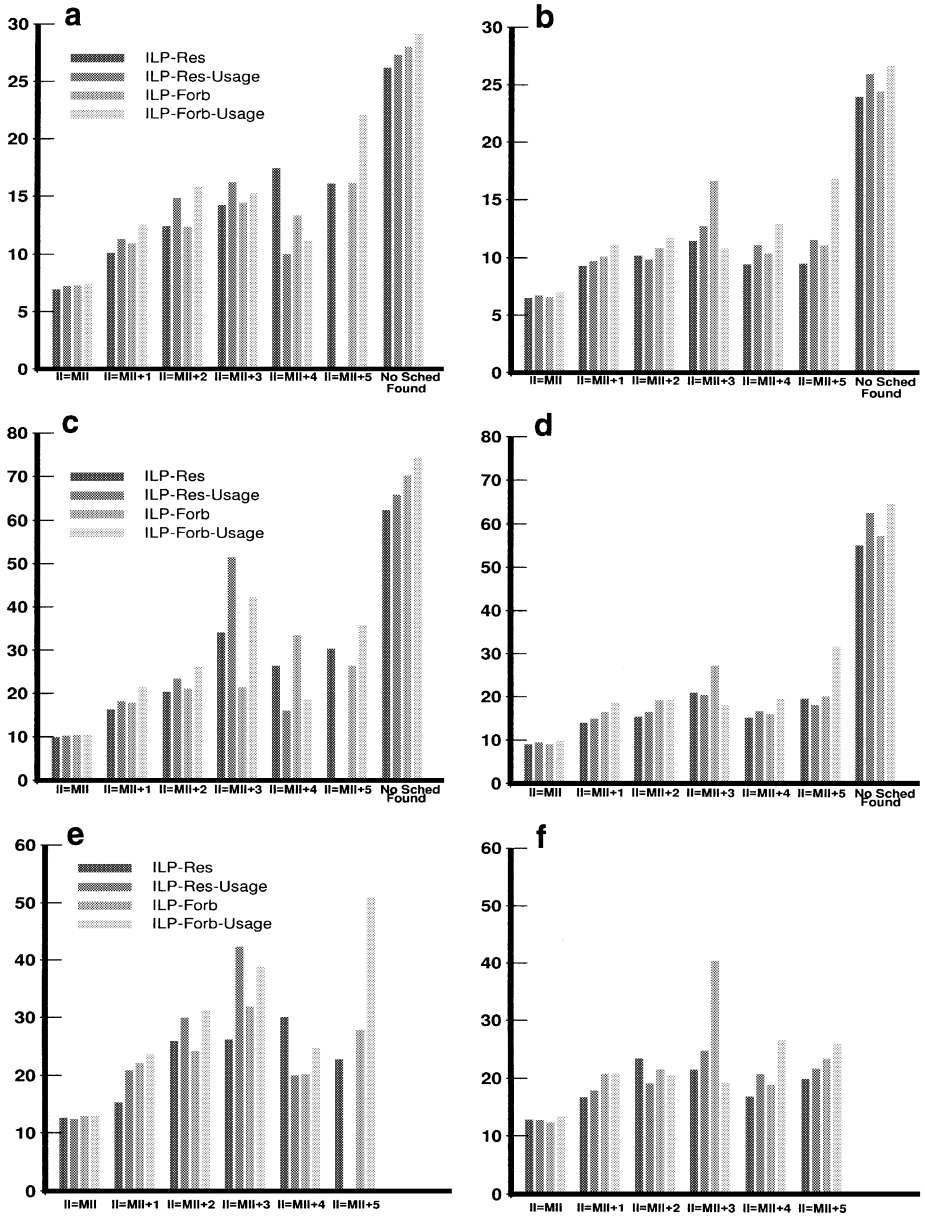

FIG. 8. Characteristics of loops scheduled by **II** and method. Number of nodes, (a) Config C1 and (b) Config C2. Number of edges, (c) Config C1 and (d) Config C2. Mean **II**, (e) Config C1 and (f) Config C2.

Table 3
Difference in II for Loops in Which a Schedule Is Found for Both C1 and C2

	ILP-Res	*ILP-Res-Usage*	*ILP-Forb*	*ILP-Forb-Usage*
$\mathbf{II_{C2}} - \mathbf{II_{C1}}$ (*arith. mean*)	0.907	1.011	0.891	0.781
$\mathbf{II_{C2}}/\mathbf{II_{C1}}$(*geom. mean*)	1.095	1.102	1.089	1.084

Table 4
Histogram of Scheduling Time (in Seconds)

Formulation	Number of test cases with scheduling time (in seconds) in the range												Sched. time	
	< 1	1–2	2–5	5–10	10–20	20–30	30–60	60–120	120–240	240–300	300–600	> 600	Geom. mean	Median
(a) Histogram for configuration C1														
ILP-Res	28	65	35	33	21	9	18	12	3	6	29	27	12.06	6.11
ILP-Res-Usage	23	75	35	43	16	13	12	14	4	16	30	12	10.50	6.13
ILP-Forb	47	76	48	17	25	5	14	4	4	10	24	29	8.94	3.15
ILP-Forb-Usage	44	73	64	19	19	9	12	11	7	5	28	20	8.13	3.42
(b) Histogram for configuration C2														
ILP-Res	15	64	32	19	19	6	12	9	6	6	36	42	20.14	10.38
ILP-Res-Usage	15	60	36	24	19	10	6	12	7	15	45	33	21.92	12.68
ILP-Forb	40	63	39	15	15	5	10	3	7	2	41	34	13.38	3.81
ILP-Forb-Usage	26	63	40	24	23	6	12	15	9	7	43	26	15.91	8.76

Table 5
Average Scheduling Time for Different ILP Formulations

	Average scheduling time (in seconds)			
	ILP-Res	*ILP-Res-Usage*	*ILP-Forb*	*ILP-Forb-Usage*
	(a) For configuration C1			
Geometric mean	10.80	8.86	6.59	5.55
Median	5.98	5.55	2.58	2.57
Arith. mean	114.87	75.43	101.77	59.68
Geom. mean of normalized sched. time	1.97	1.60	1.19	1.00
	(b) For configuration C2			
Geometric mean	20.14	21.92	13.38	15.91
Median	10.38	12.68	3.82	8.73
Arith. mean	225.92	210.96	196.89	159.98
Geom. mean of normalized sched. time	1.31	1.41	0.83	1.00

Table 5 tabulates the mean scheduling time for the four ILP formulations for the 279 loops in which all the scheduling methods found schedules (unlike Table 4 which reports times for all loops scheduled by each approach). It reports the arithmetic mean, geometric mean, median, and the geometric mean of normalized scheduling time. (The small difference in geometric mean and median of scheduling time when compared to Table 4 is due to the fact that we consider the 279 benchmarks for which all formulations obtained the schedule [10, 31].) The geometric mean of normalized scheduling time is computed by normalizing the scheduling time for individual benchmarks (with the scheduling time of the respective benchmarks under *ILP-Forb-Usage* formulation) and taking a geometric mean of the normalized values.

We observe from Table 5 that *ILP-Forb-Usage* performs better, in terms of scheduling time, than *ILP-Res-Usage* and *ILP-Res* for both configurations **C1** and **C2**. In fact, the geometric mean, median, arithmetic mean, and the normalized scheduling time of *ILP-Forb-Usage* is better than those of *ILP-Res* and *ILP-Res-Usage*. Further, based on the geometric mean of normalized scheduling time, we can say that *ILP-Forb-Usage* is faster than *ILP-Res-Usage* by roughly 60% and 40% for configurations **C1** and **C2**. The comparison between *ILP-Forb-Usage* and *ILP-Forb* does not lead to any clear conclusion. The scheduling time for *ILP-Forb-Usage* for configuration **C1** is 19% better than that for *ILP-Forb*. However, in **C2**, *ILP-Forb* has 17% lower scheduling time than *ILP-Forb-Usage*. This may be due to the fact when the structural hazards in the architecture are higher, as in configuration **C2**, the introduction of extra constraints increases the scheduling time; however this is quite the opposite for architectures with fewer structural hazards. It should, however, be noted from Fig. 7 and Table 6, that despite the increase

Table 6
Performance Comparison of ILP Formulations Based on II

	Performance improvement, in terms of **II**			
	ILP-Res	*ILP-Res-Usage*	*ILP-Forb*	*ILP-Forb-Usage*
	(a) For configuration C1			
ILP-Res	—/—	9/0.05	12/0.11	5/0.04
ILP-Res-Usage	29/0.13	—/—	23/0.16	8/0.06
ILP-Forb	22/0.11	15/0.11	—/—	12/0.11
ILP-Forb-Usage	23/0.15	8/0.05	20/0.17	—/—
	(b) For configuration C2			
ILP-Res	—/—	15/0.12	17/0.12	6/0.06
ILP-Res-Usage	35/0.13	—/—	29/0.12	7/0.05
ILP-Forb	36/0.18	25/0.22	—/—	16/0.26
ILP-Forb-Usage	36/0.13	21/0.10	34/0.11	—/—

in scheduling time *ILP-Forb-Usage* produces better schedules in terms of **II**. Hence, we order the formulations in terms of increasing performance as

[*ILP-Res, ILP-Res-Usage, ILP-Forb, ILP-Forb-Usage*].

Finally, in our comparison we give less emphasis for arithmetic mean of scheduling time, due to the large variations observed in the exeuction time for various benchmark loops. It is also observed that the standard deviation of scheduling time is very large.

Even though *ILP-Forb-Usage* performed better overall than the other three formulations, it is indeed the case that each formulation obtained the schedule faster (compared to others) for some benchmark loops. In other words, it was not the case that (any) one of the formulations performed uniformly better than all others in *all* benchmarks. In Tables 6 and 7 we compare the formulations pairwise in terms of initiation interval (**II**) and the time to construct the schedule. We report the number of benchmark programs where each formulation performed better and the average improvement achieved. The improvement, e.g., in **II**, achieved by *ILP-Forb-Usage* over *ILP-Res-Usage* is computed as

$$\frac{(\textbf{II} \text{ achieved by } \textit{ILP-Res-Usage} - \textbf{II} \text{ achieved by } \textit{ILP-Forb-Usage})}{\textbf{II} \text{ achieved by } \textit{ILP-Forb-Usage}}.$$

For example, in **C1**, the **II** achieved by *ILP-Res-Usage* is better than that obtained by *ILP-Res* in 29 benchmarks and the average improvement achieved is 0.13 (or 13%). As can be seen from Tables 6 and 7, *ILP-Forb-Usage* achieves better **II** and faster scheduling time for more benchmarks than any other formulation. Further, the average improvement achieved is also larger, in general, compared to other formulations.

Table 7
Performance Comparison of ILP Formulations Based on Scheduling Time

	Performance improvement (in terms of sched. time)			
	ILP-Res	*ILP-Res-Usage*	*ILP-Forb*	*ILP-Forb-Usage*
	(a) For configuration C1			
ILP-Res	—/—	141/1.31	41/10.21	43/2.54
ILP-Res-Usage	128/10.53	—/—	59/20.81	43/1.54
ILP-Forb	231/5.21	214/3.26	—/—	153/2.88
ILP-Forb-Usage	227/14.13	230/2.55	116/24.71	—/—
	(b) For configuration C2			
ILP-Res	—/—	122/7.35	41/58.99	72/2.45
ILP-Res-Usage	122/13.92	—/—	62/29.81	58/5.01
ILP-Forb	208/74.99	187/84.79	—/—	147/89.99
ILP-Forb-Usage	178/11.49	189/6.99	100/18.05	—/—

6.3. Performance Comparison with Slack Scheduling Method

To answer the question of whether the optimality objective and the long computation time of our methods pay off, we compared our schedules with an implementation of Huff's *Slack Scheduling* [18]. These results are tabulated in Table 8.

Table 8
Performance Comparison with *Slack Scheduling*

	Comparison of initiation interval						
	$\mathbf{II}_{ILP} = \mathbf{II}_{Slack}$	$\mathbf{II}_{ILP} < \mathbf{II}_{Slack}$			$\mathbf{II}_{ILP} > \mathbf{II}_{Slack}$		
Configuration	No. of cases	No. of cases	% of cases	% improvement	No. of cases	% of cases	% improvement
	(a) configuration C1						
ILP-Res	120	149	52.6	27.8	14	4.9	30.0
ILP-Res-Usage	117	159	54.8	29.8	14	4.8	21.9
ILP-Forb	123	163	54.3	28.0	14	4.6	31.1
ILP-Forb-Usage	121	174	56.7	28.1	12	3.9	26.4
	(b) configuration C2						
ILP-Res	100	120	45.3	26.1	45	16.9	47.3
ILP-Res-Usage	97	131	46.8	27.6	53	18.8	43.4
ILP-Forb	99	129	47.4	26.9	44	16.2	52.6
ILP-Forb-Usage	104	146	49.8	27.3	43	14.7	43.6

The comparison reveals that in more than 50% of the cases, the ILP schedule is faster than *Slack Scheduling*. The average improvement achieved by the ILP methods, in terms of **II** value, is more than 28% for all ILP formulations. Though earlier studies report near optimal performance of heuristic methods for architectures with clean pipelines [18, 24], we observe a large deviation from near optimal performance when hazards are introduced. Table 8 also reveals that in approximately 5–18% of cases, *Slack Scheduling* outperformed the ILP approaches. This arises from the time limit imposed on the *CPLEX* solver, i.e., from increasing **II** before *CPLEX* has a chance to finish with the smaller **II** value. We observe that, consistent with the other results, *ILP-Forb-Usage* achieves the largest improvement (in more than 174 cases) over *Slack Scheduling* for Configuration **C1**.

We note however, that *Slack Scheduling* produces schedules much faster than our methods, with a scheduling time less than 1 s in a large majority of cases. *Slack Scheduling* was uniform in this regard—for almost all loops, it obtained a schedule faster than our methods.

In summary, compared to *Slack Scheduling*, our ILP method yielded considerable improvement: (28%) for more than 50% of the loops examined. Thus it may be reasonable to make our approach available for performance-critical applications via a compiler switch. Further, compiler designers can use our method to obtain optimal schedules for test loops so as to compare and improve existing/newly proposed heuristic methods [30].

7. RELATED WORK

Software pipelining has been extensively studied [1, 5–7, 11, 18, 20–22, 26–28, 32, 33, 35, 37]. Rau and Fisher provided a comprehensive survey of these works in [25]. As stated in [25], software pipelining methods vary in several aspects: (1) whether or not they consider finite resources, (2) whether they model simple or complex resource usage, (3) whether the algorithm is one-pass, iterative, incremental, or enumerative search-based. In particular, in terms of resource constraints, the work reported in [1, 22, 24, 33] does not consider any resource constraint while the methods reported in [2, 4, 11, 16, 21, 26, 34–36] deal with function unit constraints but with simple resource usage. Both function unit and register resource constraints are considered in [7, 18]. Software pipelining methods for complex usage patterns with limited function units was dealt with by [5, 20, 23]. The methods proposed in these works that aim to be implementable in a practical compiler all use heuristic approaches.

By contrast, in this paper, we developed two clear mathematical formulations of the software pipelining problem. Even though the use of an integer programming method may be unacceptable as an automatic option in a practical optimizing compiler, it can be used for performance-critical kernel loops. The proposed methods are also useful in the context of a scheduling testbed where our optimal scheduling method can be used to judge how well other existing/newly proposed heuristics perform and hence to improve them [30]. In [12] we compared three heuristic algorithms with an ILP-based scheduling method for clean pipelines. In [8] Feautrier independently gave an ILP formulation similar to our earlier work. In this work we have extended our ILP formulation to pipelines with structural hazards. The proposed formulations are able to schedule and map simultaneously and obtain a fixed function unit assignment where each instruction x is executed on the same function unit $\mathrm{FU}(x)$ throughout the loop execution. Our methods

are unique in that they combine scheduling and mapping in a unified framework and attempt to achieve an optimal solution. An advantage of our methods is that they can be extended to handle multifunction pipelines as well. They can incorporate (1) minimizing buffers (logical registers) [22], (2) minimizing the maximum number of live values at any time step in the repetitive pattern, as in the method proposed by Eichenberger *et al.* [7], or (3) minimizing the number of registers needed for a loop [4].

8. CONCLUSIONS

In this paper we have proposed two methods to find software pipelined schedules for realistic pipelines with structural hazards. Both methods yield optimal schedules in terms of both initiation interval and one other desired user metric, such as register usage or function unit requirements. Additionally, both of our methods do all of this within a single unified framework. Furthermore, both methods yield schedules which can be used by dynamic out-of-order superscalar machines or by VLIW machines in which each instruction is tied to a particular function unit. The first method we propose models hardware pipeline stages directly, while the other elegantly extends hardware pipeline theory so as to model function units in their entirety.

We have implemented our scheduling methods and run experiments on a set of 415 unique loops taken from `SPEC92`, the `NAS kernels`, `linpack`, and `livermore` for two different configurations with varying degrees of structural hazards. Extensive experimental results, comparing the performance of four different ILP formulations in terms of **II**, and scheduling time, for the three configurations have been reported. In 50% of the loops, our approaches were able to find a schedule with a better initiation interval than *Slack Scheduling*, with a mean improvement of 28%. Since earlier work reported *Slack Scheduling* and other heuristics had close to optimal performance in terms of **II** on machines with clean or mostly clean pipelines, this suggests that scheduling for machines with hazards introduces significant new complexity. Furthermore this scheduling complexity pays dividends in significantly reducing the size of function units. This reduced size may allow more function units to be placed on a chip than otherwise, thus boosting performance. Even if no additional function units are added, we have found performance to be within 10% of pipelines with significantly fewer hazards.

While the automatic use of integer programming methods in practical optimizing compilers may not be acceptable, the proposed formulations are still interesting and useful. They can be used to evaluate heuristic methods and to derive optimal schedules for performance-critical applications where the user is willing to pay a high compilation cost for savings in runtime. They can also be used in hardware synthesis, where high one time overhead of finding an optimal schedule is acceptable.

APPENDIX A: PROOF OF COLORING THEOREM

In this appendix we show that our coloring formulation is correct; that is we prove the Theorem 4.1.

THEOREM 3.1. *Equations* (14)–(16) *require that two nodes x and y be assigned different colors* (*mapped to different function units*) *if and only if they overlap at some time.*

For clarity Eqs. (14)–(16) are repeated below:

$$c_x - c_y \geq \frac{u_{t,x} + u_{t,y} - 1}{2} - N \cdot w_{x,y} \tag{37}$$

$$c_y - c_x \geq \frac{u_{t,x} + u_{t,y} - 1}{2} - N \cdot (1 - w_{x,y}) \tag{38}$$

$$1 \leq c_k \leq N \qquad \forall k \in [0,\ N-1]. \tag{39}$$

Next we establish a bound for $c_x - c_y$.

LEMMA A.1. *For all nodes x and y, $-N + 1 \leq c_x - c_y \leq N - 1$.*

Proof. This lemma follows from Eq. (39). ■

Now to prove the theorem, we break the problem into two parts: (1) when two nodes x and y are assigned the same colors $c_x = c_y$ and (2) when two nodes x and y are assigned different colors, $c_x \neq c_y$. First, the same color case.

Same color: There are 3 subcases.

1. Nodes x and y overlap at some time t. This being the case, $u_{t,x}$ and $u_{t,y}$ are both 1 reducing Eqs. (37) and (38) to

$$c_x - c_y \geq \tfrac{1}{2} - N \cdot w_{x,y} \tag{40}$$

$$c_y - c_x \geq \tfrac{1}{2} - N \cdot (1 - w_{x,y}). \tag{41}$$

Noting that $c_x - c_y = 0$, rearranging terms yields

$$w_{x,y} \geq \frac{1}{2 \cdot N} \tag{42}$$

$$w_{x,y} \leq 1 - \frac{1}{2 \cdot N}. \tag{43}$$

Recall that $w_{x,y}$ is an integer 0–1 variable. However, if $w_{x,y} = 0$, then Eq. (42) is not satisfied: $0 \geq 1/(2 \cdot N)$ is false since $N \geq 1$. Similarly if $w_{x,y} = 1$, then Eq. (43) is not satisfied: $1 \leq 1 - 1/(2 \cdot N)$ is false. Thus, as desired, c_x cannot be equal to c_y when x and y overlap.

2. Nodes x and y do not ever overlap, and at time t, only one is active ($u_{t,x} = 1$ or $u_{t,y} = 1$, but not both). In either case, Eqs. (37) and (38) reduce to

$$c_x - c_y \geq -N \cdot w_{x,y} \tag{44}$$

$$c_y - c_x \geq -N \cdot (1 - w_{x,y}). \tag{45}$$

Again noting that $c_x - c_y = 0$, rearranging terms yields

$$w_{x,y} \geq 0 \tag{46}$$

$$w_{x,y} \leq 1. \tag{47}$$

Both Eqs. (46) and (47) are clearly true, whether $w_{x,y} = 0$ or $w_{x,y} = 1$.

3. Nodes x and y do not ever overlap, and at time t, neither is active ($u_{t,x} = 0$ and $u_{t,y} = 0$). In this case, Eqs. (37) and (38) reduce to

$$c_x - c_y \geq -\tfrac{1}{2} - N \cdot w_{x,y} \tag{48}$$

$$c_y - c_x \geq -\tfrac{1}{2} - N \cdot (1 - w_{x,y}). \tag{49}$$

The usual rearrangement of terms yields

$$w_{x,y} \geq -\frac{1}{2 \cdot N} \tag{50}$$

$$w_{x,y} \leq 1 + \frac{1}{2 \cdot N}. \tag{51}$$

Again, both Eqs. (50) and (51) can be satisfied by choosing either $w_{x,y} = 0$ or $w_{x,y} = 1$.

It remains to show that Eqs. (37) and (38) are both satisfied whenever two nodes are assigned *different* colors. We do this by breaking the problem into four parts depending on whether $c_x - c_y > 0$ or $c_x - c_y < 0$ and whether $w_{x,y} = 0$ or $w_{x,y} = 1$. The possibilities are illustrated by the table below:

Condition	I	II	III	IV
$c_x - c_y > 0$	True	False	True	False
$w_{x,y}$	0	1	1	0
$-N \cdot w_{x,y}$	0	–N	–N	0
$-N(1 - w_{x,y})$	–N	0	0	–N
Eq. (37) Min LHS	1	$-\mathbf{N} + \mathbf{1}$	1	$-\mathbf{N} + \mathbf{1}$
Eq. (37) Max RHS	0.5	$-N + 0.5$	$-N + 0.5$	0.5
LHS $\geq$ RHS Always?	Yes	Yes	Yes	No
Eq. (38) Min LHS	$-\mathbf{N} + \mathbf{1}$	1	$-\mathbf{N} + \mathbf{1}$	1
Eq. (38) Max RHS	$-N + 0.5$	0.5	0.5	$-N + 0.5$
LHS $\geq$ RHS Always?	Yes	Yes	No	Yes
Eq. (37) Max LHS				–1
Eq. (37) Min RHS				–0.5
LHS $\geq$ RHS Never?				Yes
Eq. (38) Max LHS			–1	
Eq. (38) Min RHS			–0.5	
LHS $\geq$ RHS Never?			Yes	

The four boldface $-\mathbf{N} + \mathbf{1}$ lower bounds of the *left-hand sides* are all due to Lemma A.1 which gives the minimum possible value of $c_x - c_y$. As is shown below, to be in the "proper operating region," Eqs. (37) and (38) require this lower bound (implied by Eq. (39)).

Now to see that both Eqs. (37) and (38) are always satisfied whenever two nodes are assigned different colors, first note that if $(c_x - c_y)$ is positive, then $w_{x,y} = 0$, and if $(c_x - c_y)$ is negative (the only other possibility if $c_x \neq c_y$), then $w_{x,y} = 1$. This is because when $(c_x - c_y)$ is positive and $w_{x,y} = 1$, (i.e., in column III), Eq. (38) is never satisfied. As the last three rows of the table indicate, the *maximum* value of the left-hand

side of Eq. (38) in this case is –1 while the *minimum* value of the right-hand side is –0.5. In other words the left-hand side is never greater than or equal to the right-hand side as required by Eq. (38). Equation (37) is likewise never satisfied when $(c_x - c_y)$ is negative and $w_{x,y} = 0$ (as illustrated in the second to last three rows of column IV in the table).

Conversely in columns I and II of the table, both Eqs. (37) and (38) are always satisfied. Intuitively, this is what is wanted. For example if $c_x > c_y$ and $w_{x,y} = 0$ as in column I, then Eqs. (37) and (38) become

$$c_x - c_y \geq 0.5 \tag{52}$$

$$c_y - c_x \geq 0.5 - N. \tag{53}$$

Equation (52) is exactly the equation we would want if we knew (a) that x and y would overlap and (b) the sign of $c_x - c_y$ ahead of time, as Eq. (52) requires c_x and c_y to be different. In this case, ($c_x > c_y$ and $w_{x,y} = 0$) Eq. (53) imposes no additional constraint.

Conversely if $c_x < c_y$ and $w_{x,y} = 1$ as in column II, then Eqs. (37) and (38) become

$$c_x - c_y \geq 0.5 - N \tag{54}$$

$$c_y - c_x \geq 0.5. \tag{55}$$

In this case, it is the second equation, Eq. (55) that gives the equation we would want if we knew that x and y would overlap and the sign of $c_x - c_y$ ahead of time—with Eq. (54) imposing no additional constraint.

Thus Eqs. (37) and (38) behave as desired in all cases, requiring different colors when nodes overlap and permitting nodes to have the same color when they do not.

APPENDIX B: RESERVATION TABLES USED IN EXPERIMENTS

Table 9
Reservation Tables for C1

Integer ALU's

	Time steps				
	0	1	2	3	4
Stage 1	x				
Stage 2		x			
Stage 3				x	
Stage 4					x

Load units

	Time steps				
	0	1	2	3	4
Stage 1	x				
Stage 2		x			
Stage 3			x		x
Stage 4				x	

Store units

	Time steps			
	0	1	2	3
Stage 1	x			
Stage 2		x		
Stage 3			x	
Stage 4				x

FP add units

	Time steps				
	0	1	2	3	4
Stage 1	x				x
Stage 2		x			
Stage 3			x		
Stage 4		x		x	

FP multiply units

	Time steps						
	0	1	2	3	4	5	6
Stage 1	x				x		
Stage 2		x				x	
Stage 3		x	x	x			x

Table 9—*Continued*

Floating point div units

	Time steps																					
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Stage 1	x																					
Stage 2		x																				
Stage 3		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Table 10
Reservation Tables for C2

Integer ALU's

	Time steps				
	0	1	2	3	4
Stage 1	x		x		
Stage 2		x			x
Stage 3		x		x	
Stage 4		x			x

Load units

	Time steps				
	0	1	2	3	4
Stage 1	x		x		
Stage 2		x			
Stage 3			x		x
Stage 4		x		x	

Store units

	Time steps			
	0	1	2	3
Stage 1	x			
Stage 2		x		x
Stage 3			x	
Stage 4				x

FP add unit

	Time steps				
	0	1	2	3	4
Stage 1	x		x		x
Stage 2		x		x	
Stage 3			x		
Stage 4		x		x	

FP multiply units

	Time steps						
	0	1	2	3	4	5	6
Stage 1	x				x		
Stage 2		x				x	
Stage 3			x				x

Floating point div units

	Time steps																					
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Stage 1	x																					
Stage 2		x																				
Stage 3		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

ACKNOWLEDGMENTS

Kemal Ebcioglu and Mayan Moudgill were instrumental in completing this paper. We also thank Qi Ning, Vincent Van Dongen, and Philip Wong for the fruitful discussions we had with them. The authors are also thankful to IBM for its technical support of this work.

REFERENCES

1. A. Aiken and A. Nicolau, Optimal loop parallelization, *in* "Proc. of the SIGPLAN '88 Conf. on Programming Language Design and Implementation," pp. 308–317, Atlanta, June 22–24, 1988.
2. A. Aiken and A. Nicolau, A realistic resource-constrained software pipelining algorithm, *in* "Advances in Languages and Compilers for Parallel Processing" (A. Nicolau, D. Gelernter, T. Gross, and D. Padua, Eds.), Res. Monographs in Parallel and Distrib. Computing, Chap. 14, pp. 274–290, Pitman and the MIT Press, London and Cambridge, MA, 1991.
3. J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, Conversion of control dependence to data dependence, *in* "Conf. Rec. of the Tenth Ann. ACM Symp. on Principles of Programming Languages," pp. 177–189, Austin, Jan. 24–26, 1983.
4. E. R. Altman, "Optimal Software Pipelining with Function Unit and Register Constraints," Ph.D. thesis, McGill U., Montréal, Oct. 1995.
5. J. C. Dehnert and R. A. Towle, Compiling for Cydra 5, *J. Supercomput.* **7** (May 1993), 181–227.
6. K. Ebcioğlu, A compilation technique for software pipelining of loops with conditional jumps, *in* "Proc. of the 20th Ann. Work. on Microprogramming," pp. 69–79, Colorado Springs, Dec. 1–4, 1987.
7. A. E. Eichenberger, E. S. Davidson, and S. G. Abraham, Minimum register requirements for a modulo schedule, *in* "Proc. of the 27th Ann. Intl. Symp. on Microarchitecture," pp. 75–84, San Jose, Nov. 30–Dec. 2, 1994.
8. P. Feautrier, Fine-grain scheduling under resource constraints, *in* "Proc. of the 7th Intl. Work. on Languages and Compilers for Parallel Computing" (K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds.), Ithaca, Aug. 8–10, 1994, Lecture Notes in Computer Science, Vol. 892, pp. 1–15, Springer–Verlag, Berlin/New York, 1995.
9. J. A. Fisher, Trace scheduling: A technique for global microcode compaction, *IEEE Trans. Comput.* **30,** 7 (July 1981), 478–490.
10. P. J. Fleming and J. J. Wallace, How not to lie with statistics: The correct way to summarize benchmark results, *Comm. Assoc. Comput. Mach.* **29,** 3 (Mar. 1986), 218–221.
11. F. Gasperoni and U. Schwiegelshohn, "Efficient Algorithms for Cyclic Scheduling," Research Report RC 17068, IBM T. J. Watson Res. Center, Yorktown Heights, NY, 1991.
12. R. Govindarajan, E. R. Altman, and G. R. Gao, A framework for resource constrained rate-optimal software pipelining, *in* "Proc. of the Third Joint Intl. Conf. on Vector and Parallel Processing (CONPAR 94-VAPP VI)" (B. Buchberger and J. Volkert, Eds.), Linz, Austria, Sep. 6–8, 1994, Lecture Notes in Computer Science, Vol. 854, pp. 640–651, Springer–Verlag, Berlin, 1994.
13. R. Govindarajan, E. R. Altman, and G. R. Gao, Minimizing register requirements under resource-constrained rate-optimal software pipelining, *in* "Proc. of the 27th Ann. Intl. Symp. on Microarchitecture," pp. 85–94, San Jose, Nov. 30–Dec. 2, 1994.
14. R. Govindarajan, E. R. Altman, and G. R. Gao, A framework for resource-constrained rate-optimal software pipelining, *IEEE Trans. Parallel Distrib. Systems* **7,** 11 (Nov. 1996), 1133–1149.
15. L. J. Hendren, G. R. Gao, E. R. Altman, and C. Mukerji, A register allocation framework based on hierarchical cyclic interval graphs, *in* "Proc. of the 4th Intl. Conf. on Compiler Construction, CC '92" (U. Kastens and P. Pfahler, Eds.), Paderborn, Germany, Oct. 5–7, 1992, Lecture Notes in Computer Science, Vol. 641, pp. 176–191, Springer–Verlag, Berlin, 1992.
16. P. Y. T. Hsu, "Highly Concurrent Scalar Processing," Ph.D. thesis, University of Illinois at Urbana-Champagne, Urbana, IL, 1986.
17. T. C. Hu, "Integer Programming and Network Flows," Addison-Wesley, Reading, MA, 1969.
18. R. A. Huff, Lifetime-sensitive modulo scheduling, *in* "Proc. of the ACM SIGPLAN '93 Conf. on Programming Language Design and Implementation," pp. 258–267, Albuquerque, June 23–25, 1993.
19. P. M. Kogge, "The Architecture of Pipelined Computers," McGraw-Hill, New York, 1981.
20. M. Lam, Software pipelining: An effective scheduling technique for VLIW machines, *in* "Proc. of the SIGPLAN '88 Conf. on Programming Language Design and Implementation," pp. 318–328, Atlanta, GA, June 22–24, 1988.

21. S.-M. Moon and K. Ebcioğlu, An efficient resource-constrained global scheduling technique for superscalar and VLIW processors, *in* "Proc. of the 25th Ann. Intl. Symp. on Microarchitecture," pp. 55–71, Portland, OR, Dec. 1–4, 1992.
22. Q. Ning and G. R. Gao, A novel framework of register allocation for software pipelining, *in* "Conf. Rec. of the Twentieth Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages," pp. 29–42, Charleston, SC, Jan. 10–13, 1993.
23. S. Ramakrishnan, Software pipelining in PA-RISC compilers, *Hewlett-Packard J.* **43,** 3 (June 1992), 39–45.
24. J. Ramanujam, Optimal software pipelining of nested loops, *in* "Proc. of the 8th Intl. Parallel Processing Symp.," pp. 335–342, Cancún, Mexico, Apr. 26–29, 1994.
25. B. R. Rau and J. A. Fisher, Instruction-level parallel processing: History, overview and perspective, *J. Supercomput.* (May 1993), 9–50.
26. B. R. Rau and C. D. Glaeser, Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing, *in* "Proc. of the 14th Ann. Microprogramming Work," pp. 183–198, Chatham, MA, Oct. 12–15, 1981.
27. B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker, Register allocation for software pipelined loops, *in* "Proc. of the ACM SIGPLAN '92 Conf. on Programming Language Design and Implementation," pp. 283–299, San Francisco, June 17–19, 1992.
28. B. R. Rau, Iterative modulo scheduling: An algorithm for software pipelining loops, *in* "Proc. of the 27th Ann. Intl. Symp. on Microarchitecture," pp. 63–74, San Jose, Nov. 30–Dec. 2, 1994.
29. R. Reiter, Scheduling parallel computations, *J. Assoc. Comput. Mach.* **15,** 4 (Oct. 1968), 590–599.
30. J. Ruttenberg, G. R. Gao, A. Stouchinin, and W. Lichtenstein, Software pipelining showdown: Optimal vs. heuristic methods in a production compiler, *in* "Proc. of the ACM SIGPLAN '96 Conf. on Programming Language Design and Implementation," pp. 1–11, Philadelphia, May 22–24, 1996.
31. J. E. Smith, Characterizing computer performance with a single number, *Comm. Assoc. Comput. Mach.* **31,** 10 (Oct. 1988), 1202–1206.
32. R. F. Touzeau, A Fortran compiler for the FPS-164 scientific computer, *in* "Proc. of the SIGPLAN '84 Symp. on Compiler Construction," pp. 48–57, Montréal, June 17–22, 1984.
33. V. Van Dongen, G. R. Gao, and Q. Ning, A polynomial time method for optimal software pipelining, *in* "Proc. of the Conf. on Vector and Parallel Processing, CONPAR-92," Lyon, France, Sept. 1–4, 1992, Lecture Notes in Computer Science, Vol. 634, pp. 613–624, Springer–Verlag, Berlin, 1992.
34. S. R. Vegdahl, A dynamic-programming technique for compacting loops, *in* "Proc. of the 25th Ann. Intl. Symp. on Microarchitecture," pp. 180–188, Portland, OR, Dec. 1–4, 1992.
35. J. Wang and E. Eisenbeis, "A New Approach to Software Pipelining of Complicated Loops with Branches," Research Report, Institut Nat. de Recherche en Informatique et en Automatique (INRIA), Rocquencourt, France, Jan. 1993.
36. N. J. Warter, G. E. Haab, J. W. Bockhaus, and K. Subramanian, Enhanced modulo scheduling for loops with conditional branches, *in* "Proc. of the 25th Ann. Intl. Symp. on Microarchitecture," pp. 170–179, Portland, OR, Dec. 1–4, 1992.
37. N. J. Warter, S. A. Mahlke, W. W. Hwu, and B. R. Rau, Reverse if-conversion, *in* "Proc. of the ACM SIGPLAN '93 Conf. on Programming Language Design and Implementation," pp. 290–299, Albuquerque, June 23–25, 1993.

ERIK ALTMAN is a research staff member at the IBM Thomas J. Watson Research Center in Yorktown Heights, New York, where he has worked since receiving his Ph.D. in 1995. He received his Ph.D. and M.Eng. from McGill University, Montreal, Canada, and his S.B. from the Massachusetts Institute of Technology. His research interests include binary translation, VLIW architectures, simulation, and instruction scheduling. Dr. Altman is a member of the IEEE, IEEE Computer Society, and ACM SIGARCH and SIGPLAN.

R. GOVINDARAJAN is working as an assistant professor in the Supercomputer Education and Research Center and in the Department of Computer Science and Automation, Indian Institute of Science, Bangalore,

India. His research interests are in the areas of instruction scheduling, dataflow and multithreaded architectures, distributed shared memory systems, programming models, and scheduling of DSP applications. Govindarajan received his Ph.D. and Bachelors in engineering from the Indian Institute of Science, Bangalore, India. Between 1989 and 1992 he worked as a post-doctoral fellow at the University of Western Ontario, London, Canada, and at McGill University, Montreal, Canada. He worked as an assistant professor in the Department of Electrical Engineering, McGill University, Montreal, Canada from 1992 to 1994 and in the Department of Computer Science, Memorial University of Newfoundland, St. John's, Canada, from 1994 to 1995. R. Govindarajan is a member of the IEEE, IEEE Computer Society, and ACM SIGARCH.

GUANG R. GAO received his S.M. and Ph.D. in electrical engineering and computer science from the Massachusetts Institute of Technology, in 1982 and 1986, respectively. Currently he is an associate professor in the Department of Electrical and Computer Engineering at the University of Delaware, where he has been the founder and leader of the Computer Architecture and Parallel Systems Lab. Prior to that he was an associate professor in the School of Computer Science, and the founder and a leader of the Advanced Compilers, Architectures and Parallel Systems (ACAPS) Group at McGill University, Montreal, Canada.

Professor Gao's main research interests include: (1) programming language design and implementation: program flow analysis, code scheduling, and optimization; (2) parallel programming and parallelizing compilers; and (3) high-performance computing systems and architectures, in particular multithreaded architecture and systems. He has many research publications, over 90 in refereed conference/workshop proceedings and journals. He has edited or co-edited several research monographs. He is the Co-Editor of the *Journal of Programming Languages* and a member of the editorial board of the *IEEE Concurrency Journal*. He has served as a guest editor on special issues for the *Journal of Parallel and Distributed Computing* and *IEEE Transactions on Computers*. He has served on the organizing committees, program committees, and steering committees of many international conferences and workshops in his field. Dr. Gao is a senior member of IEEE and a member of ACM and IFIP WG 10.3.

Received January 17, 1996; revised May 20, 1997; accepted January 28, 1998