In the Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation, La Jolla, California, June 18-21, 1995, pages 139-150. Copyright © 1995 ACM (see notice below).

Scheduling and Mapping: Software Pipelining in the Presence of Structural Hazards *

Erik R. Altman
Dept. of Electrical Engineering
McGill University
Montreal, H3A 2A7, CANADA
erik@macs.ee.mcgill.ca

R. Govindarajan
Dept. of Computer Science
Memorial Univ. of Newfoundland
St. John's, A1B 3X5, CANADA
govind@cs.mun.ca

Guang R. Gao
School of Computer Science
McGill University
Montreal, H3A 2A7, CANADA
gao@cs.mcgill.ca

Abstract

Recently, software pipelining methods based on an ILP (Integer Linear Programming) framework have been successfully applied to derive rate-optimal schedules for architectures involving clean pipelines — *pipelines without structural hazards. The problem for architectures beyond such clean pipelines remains open. One challenge is how, under a unified ILP framework, to simultaneously represent resource constraints for* unclean *pipelines, and the assignment or* mapping *of operations from a loop to those pipelines. In this paper we provide a framework which does exactly this, and in addition constructs* rate-optimal *software pipelined schedules.*

The proposed formulation and a solution method have been implemented and tested on a set of 1066 loops taken from various scientific and integer benchmark suites. The formulation found a rate-optimal schedule for 75% of the loops, and required a median time of only 2 seconds per loop on a Sparc 10/30.

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 software pipelining algorithms [1, 5, 7, 11, 13, 16, 17, 18, 20, 24, 27, 26] have been proposed which operate under resource constraints. An excellent survey of these algorithms can be found in [19].

*This work was supported by research grants from NSERC (Canada) and MICRONET –Network Centers of Excellence (Canada).

Integer Linear Programming (ILP)-based and other approaches have been proposed for finding an optimal periodic schedule on machines with simple resource usage [6, 9, 25]. However, this work has assumed that function units are either completely pipelined or have no pipelining at all. Furthermore, it did not address the issue of *mapping*, that is the assignment of instructions to specific function units. For example if a loop has 3 **divide** instructions, `a`, `b`, and `c` and the architecture, 2 **Divide** units, `X` and `Y`, the *mapping* determines whether `a` is executed on `X` or `Y`, and likewise for `b` and `c`. Earlier work based on the ILP approach [6, 9] implicitly assumes that the mapping is done at runtime, based on which of the function units are available. However, as we shall show, for non-pipelined function units and function units with structural hazards, there are significant restrictions on mapping — restrictions which preclude some schedules which would otherwise be legal. As a result, we require that the FU assignment (mapping) be fixed at compile-time.

In this paper, we aim to eliminate these earlier restrictions. The approach outlined here is a unified framework which: (1) finds the fastest possible schedule, (2) handles pipelines with arbitrary structural hazards, and (3) correctly maps instructions to these pipelines at compile time. Specifically,

- We extend our ILP method to architectures which are either non-pipelined or pipelined but containing arbitrary structural hazards, namely *unclean pipelines*. We show that the mapping required can be identified as an instance of the *circular-arc graph coloring* problem [10] and can be represented by integer linear constraints and smoothly integrated into our overall ILP framework.

- We have implemented this extended framework and have performed tests on 1066 loops found in typical benchmark programs such as the `SPEC92`, the `NAS kernels`, `linpack`, and the `livermore loops`. In these experiments, we obtained an optimal schedule for greater than 75% of the loops.

We ran the experiments on a Sparc 10/30, which required a median of 2 seconds and a geometric mean of only 5 seconds per loop to compute these schedules. To our knowledge, this is the first report of experimental results on optimal ILP-based schedules for pipelines with structural hazards on a large number of real loops. We hope that these results will help the compiler community in the assessment of ILP approaches.

In spite of these experimental results, more experimental work in the future is required to study the practicality of an ILP approach in a real compiler. Nonetheless, we believe that it is very useful to compiler designers to establish an optimal bound so as to compare and improve their heuristic methods of software pipelining.

The rest of this paper is organized as follows. In the following section we present the necessary background and motivation for using an ILP approach for software pipelining. In Section 3, we motivate our approach to the scheduling problem in the presence of unclean pipelines with an example. Section 4 deals with the formulation of the scheduling and mapping problems for non-pipelined execution units. In Section 5 we extend our formulation to function units with arbitrary structural hazards. Experimental results involving 1066 loops are reported in Section 6. We compare our work with other related work in Section 7. Concluding remarks are presented in Section 8.

2 Background and Motivation

Software pipelining has been proposed as an efficient method for loop scheduling for high-performance VLIW and super-scalar 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 [7, 13, 22, 26] 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.

A software pipelined schedule can be represented in a linear periodic form [23], under which an instruction i in iteration j is initiated at time $j * T + t_i$. T is the period (or *initiation interval* — often abbreviated as **II** in the literature) of the schedule. The value t_i is a constant and indicates the time at which i first executes. As a result, the mathematical formulation of software pipelining using an ILP framework appears to be quite natural. The basic formulation requires the casting of precedence constraints into linear form which is straightforward [24]. In [18], it was demonstrated that register storage requirements can also be represented in a simple linear form and integrated easily into the basic ILP framework. In [9], we extended this framework further to include function unit constraints (also in linear form) for hardware pipelines without structural hazards. In the next section, we present an example which demonstrates this framework.

Of course, finding optimal solutions for an ILP problem is NP-Hard. 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. There is experimental evidence that ILP approaches may produce better schedules than many heuristic methods for software pipelining [9].

One interesting question is: will cleverly designed exhaustive search methods be superior to an ILP solver in terms of efficiency? Although we have lately been working on exploiting such alternatives [2], it is still too early to make a conclusion. In the meantime, 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 largely beyond the scope of this paper.

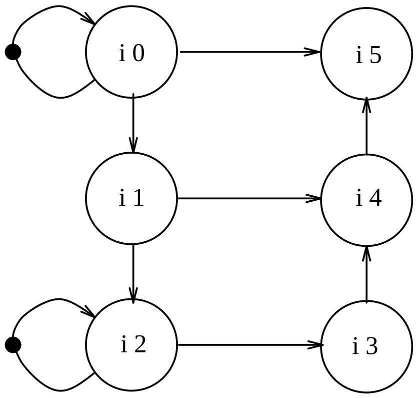
In the rest of this paper, we discuss how to extend the ILP framework to handle hardware pipelines with structural hazards.

3 Scheduling Unclean Pipelines

In this section, we use a motivating example to illustrate the scheduling problems in the presense of unclean pipelines. In section 3.1, we address the issues when the execution units are non-pipelined — a special, but very important case of unclean pipelines. In section 3.2, we examine execution units which are pipelined but have structural hazards.

3.1 Non-pipelined Execution Units

We use the simple example loop in Figure 1 as a running example throughout. Both C language and instruction-level representations are given in Figure 1(b) while the Data Dependence Graph (DDG) is



(a) Dependence Graph

```
for (i = 0; i < n; i++) {
      s = s * a[i];
      a[i] = s * s * a[i];
}
```

$i0$:	vr33 = vr33 + vr32	% vr33 is address of a[i] %
$i1$:	vr34 = load m(vr33)	% vr34 = a[i] %
$i2$:	vr35 = vr35 * vr34	% vr35 = s %
$i3$:	vr36 = vr35 * vr35	% vr36 stores s $*$ s %
$i4$:	vr37 = vr36 * vr34	% vr37 = new a[i] %
$i5$:	store(vr37, m(vr33))	
	branch to $i0$ if $i \leq n$	
	enddo	

(b) Program Representation

Figure 1: An Example Loop

depicted in Figure 1(a).

For purpose of discussion in this section, we consider an architecture with 3 **Integer** units, 2 **Floating-Point** (**FP**) units and 1 **Load/Store** unit. We assume that the **Integer** units perform all integer operations and likewise the **FP** units perform all floating point operations. Further let the execution time of an **Integer add** ($i0$) be 1 cycle, while that of an **FP multiply** ($i2$, $i3$, and $i4$) be 2 cycles. Assume also that **load** ($i1$) takes 2 cycles and **store** ($i5$) takes 1 cycle.[1] Lastly we assume that all function units are non-pipelined, i.e. the next operation cannot be initiated on an FU until the previous execution is completed. It is well known that the minimum initiation interval T_{min} of successive iterations of a loop is constrained by both *loop-carried* dependences and available resources [4, 13, 16, 19, 20]. The loop-carried dependences put a lower bound, T_{dep}, on T. The value of T_{dep} is determined by the critical cycle(s) in the loop [23]. Specifically

$$T_{dep} = \frac{\text{sum of instruction execution times}}{\text{sum of dependence distances}}$$

along the critical cycle(s). For the DDG in Figure1(a), T_{dep} is 2 corresponding to the self loop at instruction $i2$.

Another lower bound T_{res} on T is enforced by resource constraints. Consider (1) all the operations in the loop that use FU type r (e.g. **FP** unit), (2) there are F_r of type r in the architecture, and (3) the total execution time of all instructions in FU type r is T_r cycles, then

$$T_{res} = \max_r \left\lceil \frac{T_r}{F_r} \right\rceil$$

where the maximum is taken over all FU types r. In our example the T_{res} bound is given by both the **FP** unit and the **Load/Store** unit. That is,

$$\begin{aligned} T_{res} &= \max(T_{\mathbf{Int}}, T_{\mathbf{FP}}, T_{\mathbf{Load/Store}}) \\ &= \max\left(\left\lceil \frac{1}{3} \right\rceil, \left\lceil \frac{2+2+2}{2} \right\rceil, \left\lceil \frac{2+1}{1} \right\rceil\right) \\ &= \max(1, 3, 3) = 3 \end{aligned}$$

The shortest initiation interval T_{lb} is the maximum of T_{res} and T_{dep}. However there may or may not exist a schedule with a period T_{lb} satisfying the given resource constraints. T_{min} is the minimum period for which a schedule exists [2].

There are several reasons that we require a compile-time fixed mapping of instructions to function units. It is often impractical for an instruction to be switched from one function unit to another during the course of its execution. For instance, since a specific operation (or part) of the instruction word is linked to a function unit in VLIW architectures, it becomes necessary to have a *fixed* assignment from an instruction to FU. This fixed instruction (task) to FU mapping may also be essential in architectures with multiple register sets (such as the Multiflow Trace machine [3]), with each dedicated to one (or a group of) execution units.

To illustrate the significance of this fixed assignment constraint, examine Schedule A in Table 1. This schedule is for the DDG in Figure 1(a) and is obtained from the linear schedule form $T \cdot i + t_s$, with $T = 3$, $t_{i0} = 0$, $t_{i1} = 1$, $t_{i2} = 3$, $t_{i3} = 5$, $t_{i4} = 7$, and $t_{i5} = 9$. We use the notation $\_i1$ to indicate that the execution of instruction $i1$ continues from the previous time step. Since the FUs are assumed to be non-pipelined, instructions continuing from previous time step, continue to engage the FU. Schedule A has a prolog (from

[1] These latency values are assumed to make the illustrations in this section compact. More realistic values for the execution times based on PowerPC-604 [14] are assumed in the experimental section.

[2] In some cases an initiation rate better than $\frac{1}{T_{min}}$ can be obtained by unrolling the loop. We do not consider unrolling in this paper, although the techniques presented here can be used with unrolled loops as well.

	Time Steps													
	0	1	2	3	4	5	6	*7*	*8*	*9*	10	11	12	13
Iteration = 0	*i*0	*i*1	_*i*1	*i*2	_*i*2	*i*3	_*i*3	**i4**	**_i4**	i5				
Iteration = 1				*i*0	*i*1	_*i*1	*i*2	**_i2**	**i3**	**_i3**	*i*4	_*i*4	*i*5	
Iteration = 2							*i*0	i1	_i1	**i2**	_*i*2	*i*3	_*i*3	*i*4
Iteration = 3										i0	*i*1	_*i*1	*i*2	_*i*3
Iteration = 4													*i*0	*i*1

Table 1: Schedule A for the Motivating Example

	Time Steps														
	0	1	2	3	4	5	6	7	*8*	*9*	*10*	*11*	12	13	14
Iteration = 0	*i*0	*i*1	_*i*1	*i*2	_*i*2	*i*3	_*i*3	*i*4	**_i4**			i5			
Iteration = 1					*i*0	*i*1	_*i*1	*i*2	**_i2**	**i3**	**_i3**	**i4**	_*i*4		
Iteration = 2									i0	i1	_i1	**i2**	_*i*2	*i*3	_*i*3

Table 2: Alternative Schedule B for the Motivating Example (with $T = 4$)

$[0, 6]$), a repetitive pattern ($[7, 9]$), and an epilog. Note that the resource constraints are met in the repetitive pattern — no more than 3 **Integer**, 2 **FP**, and 1 **Load/Store** unit are needed at any one time. (`Times 7 and 8`: 2 **FP** and 1 **Load/Store**; `Time 9`: 1 **Integer**, 2 **FP**, 1 **Load/Store**.) In other words, Schedule A satisfies both the dependence constraints and the resource constraints of our architecture.

Unfortunately, Schedule A does not satisfy the fixed FU assignment criteria. To see this, consider the repetitive pattern starting at time step 7. If we assign the first **FP** unit to instruction $i4$ at time step 7, and the second **FP** unit to $i3$ at time step 8, then we have the first **FP** unit free at time step 9 and the second **FP** unit free at time 7 (or time step 10, taking the time steps with *modulo* 3). Thus we cannot make a fixed FU assignment for instruction $i2$. In fact there does not exist any schedule with a period $T = 3$ which satisfies the fixed FU assignment *and* requires only 2 **FP** units (in addition to 1 **Integer** and 1 **Load/Store** unit).

An important challenge here is that the problem of instruction-to-FU-mapping and resource-constrained software pipelining must be formulated within a unified framework. This leads to the first problem to be formulated and solved in this paper.

> **Problem 1:** Given a DDG and an architectural configuration with non-pipelined execution units, construct a schedule that has the shortest initiation interval, using only the available resources, and mapping each instruction i to the same function unit $FU(i)$ in every iteration.

One important observation we have made is to identify the FU assignment problem as a circular-arc coloring problem [10]. In Section 4.2 we introduce this coloring as a linear constraint to the basic ILP formulation for handling non-pipelined FUs which will be presented in Section 4.

Finally, under fixed FU assignment, Schedule B shown in Table 2 is a $T = T_{min} = 4$ schedule (one with the shortest initiation interval) for the example architecture, with 3 **Integer** units, 2 **FP** units and 1 **Load/Store** unit.

3.2 Pipelines with Structural Hazards

In this subsection we illustrate how the software pipelining problem is altered by the presence of structural hazards in function unit pipelines. In such a function unit, the resource usage of different stages of the pipeline is generally represented by a reservation table [15].

Once again we consider the example loop shown in Figure 1 with an architecture consisting of 3 **Integer** units of latency 1, 2 **FP** units of latency 3, and 1 **Load/Store** unit of latency 3. However, we now assume that the **FP** and **Load/Store** units have structural hazards. In order to make the discussion simple, we assume both the **FP** unit and the **Load/Store** unit to be 3-stage pipelines with the simple reservation tables shown in Figure 2(a) and Figure 2(b).

Schedule C in Figure 2(c) is compatible with this architecture. Unlike Schedules A and B, Schedule C depicts only when an operation is initiated. When hazards are present, resource usage is more complicated, and we instead represent it by a set of *resource-usage tables* as shown in Figure 2(d) for the **FP** unit.

	Time Steps		
	0	1	2
Stage 1	x		
Stage 2		x	
Stage 3		x	x

(a) **FP** Unit

	Time Steps		
	0	1	2
Stage 1	x		x
Stage 2		x	
Stage 3			x

(b) **Load/Store** Unit

Reservation Tables

	Time Steps																
	0	1	2	3	4	5	6	7	8	9	10	11	12	*13*	*14*	*15*	*16*
Iteration = 0	*i*0	*i*1			*i*2			*i*3			*i*4						i5
Iteration = 1					*i*0	*i*1			*i*2			*i*3			**i4**		
Iteration = 2									*i*0	*i*1			*i*2			**i3**	
Iteration = 3													*i*0	i1			**i2**
Iteration = 4																	i0

(c) Schedule C for Pipelines with Structural Hazards

	Stage 1				Stage 2				Stage 3			
	Time Steps				Time Steps				Time Steps			
	13	14	15	16	13	14	15	16	13	14	15	16
Iter. = 0												
Iter. = 1		*i*4					*i*4		*i*3		*i*4	*i*4
Iter. = 2			*i*3		*i*2			*i*3	*i*2	*i*2		*i*3
Iter. = 3				*i*2								
Iter. = 4												

(d) Resource Usage Tables for **FP** Unit

Figure 2: Reservation Tables, Schedule C, and corresponding Resource Usage Tables

The way in which the **FP** *resource usage tables* in Figure 2(d) are derived from the **FP** reservation table in Figure 2(a) and Schedule C in Figure 2(c) is straightforward. First consider Stage 1 of the **FP** unit. In the reservation table in Figure 2(a) it is used only at time 0. Thus in the resource usage table for **FP** Stage 1 in Figure 2(d), each **FP** operation (**i2, i3**, and **i4**) occurs at the same time as in Schedule C. (For software pipelining, we look only at the periodic pattern from $[13, 16]$.) Stage 2 however, is used starting one cycle after the start of the operation. Thus each **FP** operation is rotated one to the right from the resource usage table of Stage 1. Since **i2** was at time 16 in Stage 1, it "wraps around" to time 13 in Stage 2. The reservation table indicates that **FP** Stage 3 is used both one *and* two cycles after the start of a floating point operation. This means that the resource usage table for Stage 3 has operations at the same times as Stage 2 and in addition has them rotated one place to the right. Stage 3 is the only one in which multiple instructions use a stage at the same time: **i2** and **i3** both use Stage 3 at time 13, while **i3** and **i4** do so at time 16. This being the case, both **FP** units in the architecture are required. This mapping of reservation tables and schedules to *resource usage tables* allows us in Section 5 to solve a generalization of **Problem 1**:

> **Problem 2:** 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 i to the same function unit $FU(i)$ in every iteration.

Combining our ILP approach with the resource usage table provides the key to an ILP formulation for Problem 2. As can easily be seen, the fixed FU assignment problem in the presence of structural hazards is also related to the coloring problem. In fact, this can be derived by an extension of the formulation of the non-pipelined case discussed earlier.

4 Scheduling Non-Pipelined Execution Units

In this section, we develop a formulation for the software pipelining problem for architectures involving non-pipelined execution units. The formulation is based on our earlier work on integer programming formulation for clean pipelines [9]. The basic ILP formulation reported in [9] is extended to non-pipelined architectures in Section 4.1. In Section 4.2, we develop a graph coloring formulation which guarantees a schedule with fixed FU assignment for each instruction. The result of this integration is a unified formulation for scheduling and mapping.

4.1 The Basic ILP Formulation

As discussed in Section 2, we consider linear periodic schedules of the form $j \cdot T + t_i$. The repetitive pattern in the software pipelined schedule is represented succinctly in our ILP formulation by a $T \times N$ matrix $\mathcal{A}$, where N is the number of nodes in the DDG. $\mathcal{A}$ is a 1-0 matrix with $a_{t,i} = 1$ if and only if instruction i is scheduled to begin at time step t in the repetitive pattern. Otherwise $a_{t,i} = 0$. As will be seen shortly, the $\mathcal{A}$ matrix captures the *modulo-reservation table* [16, 20] in a form that is suitable for representing resource constraints in a linear form.

The t_i variables in our linear periodic schedule form can be related to the $\mathcal{A}$ matrix as

$$\mathcal{T} = T \cdot \mathcal{K} + \mathcal{A}^{\text{Transpose}} \times [0, 1, \cdots, T-1]^{\text{Transpose}} \quad (1)$$

where $\mathcal{K}$ and $\mathcal{T}$ are N-element vectors. Intuitively k_i corresponds to the iteration from which the first occurrence of i in the repetitive pattern comes, while $\mathcal{A}^{\text{Transpose}} \times [\cdots]^{\text{Transpose}}$ is the offset of that i from the start of the pattern.

The $\mathcal{T}$, $\mathcal{K}$, and $\mathcal{A}$ matrices for Schedule B are depicted in Figure 3. For example, the 1 in the 4th row, 3rd column of $\mathcal{A}$ indicates that **i2** begins execution at the end of the Schedule B's loop pattern (times $[8, 11]$ in Figure 2).

Notice that the $\mathcal{A}$ matrix specifies only the time step (in the repetitive pattern) at which an instruction is *initiated*. To determine resource requirements with

$$\mathcal{T} = [0,\ 1,\ 3,\ 5,\ 7,\ 11]^{\text{Transpose}},$$

$$\mathcal{K} = [0,\ 0,\ 0,\ 1,\ 1,\ 2]^{\text{Transpose}}, \quad \text{and}$$

$$\mathcal{A} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} : 6 \text{ ops} \times (T = 4)$$

$$U = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}.$$

Figure 3: $\mathcal{T}$, $\mathcal{K}$, $\mathcal{A}$, and U matrices for Schedule B

non-pipelined function units, we need to know not just when each instruction is *initiated*, but also how long each executes.

If an instruction i with latency d_i is initiated at time step t in the repetitive pattern, it will execute until time step $(t + d_i - 1) \bmod T$. Since the FU is non-pipelined, no other instructions may use it for this entire period. Intuitively this converts the modulo reservation table $\mathcal{A}$ which indicates only when an operation starts to the matrix U which indicates the entire time during which an operation uses a non-pipelined FU, as illustrated in Figure 3 for Schedule B.

Formally, U is derived from $\mathcal{A}$ as follows:

$$u_{t,i} = \sum_{l=0}^{(d_i-1)} a_{((t-l) \bmod T),i}, \quad \forall t \in [0, T-1], \text{ and } \forall i \in V \quad (2)$$

Notice that if all instructions have delay $d_i = 1$ cycle, then $u_{t,i} = a_{t,i}$ [3]. Since, $\sum_{t=0}^{T-1} a_{t,i} = 1$ $u_{t,i} \in \{0, 1\}$. To compute the resource requirement at time step t, we need to add the $u_{t,i}$'s for all i. However since different instructions may need different types of function units, we compute the resource requirement for each type of function unit separately. To accomplish this, we consider $\mathcal{I}(r)$: the set of all instructions that require a function unit of type r.

If R_r function units of type r are required for the schedule then, clearly,

$$\sum_{i \in \mathcal{I}(r)} u_{t,i} \le R_r, \quad \forall t \in [0, T-1] \quad (3)$$

The ILP formulation attempts to minimize a weighted sum of the R_r for the various function unit types. The weight (or cost) associated with FU type r is C_r and may reflect for example, the criticality of the FU for the loop. In this way it can act as a heuristic to help guide the ILP. The objective function, then, is

$$\min \sum_{r=0}^{h-1} C_r \cdot R_r$$

Lastly, as shown in [9, 18, 23], the dependence constraints specified by the DDG can be enforced on the schedule by

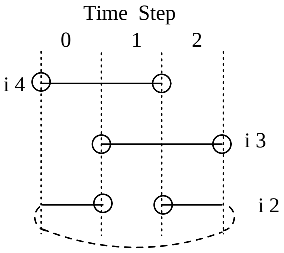
$$t_j - t_i \ge d_i - T \cdot m_{ij} \quad \forall (i,j) \in E \quad (4)$$

where d_i is the latency of instruction i and m_{ij} is the dependence distance from i to j. Further, in order to guarantee that each instruction is scheduled exactly once in the repetitive pattern, we require additional constraints on the $\mathcal{A}$ matrix [9]. These constraints are shown in Equations (9) and (10). The

[3] Since clean pipelines can initiate a new operation in each cycle, the resource usage for an instruction is, conceptually, for only one cycle. Hence in those cases, again, $u_{t,i} = a_{t,i}$.

Time Steps		
0	1	2
i4	**_i4**	*i*5
_i2	**i3**	**_i3**
*i*1	_*i*1	**i2**
		*i*0

(a)



(b)

Figure 4: A Repetitive Pattern and its (partial) Resource Usage

complete integer programming formulation is shown below. The formulation minimizes the objective function subject to the dependency constraints of Equation (8) and the resource constraints of Equation (5).

[ILP Formulation for Non-Pipelined Architectures]

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

subject to

$$\sum_{i \in \mathcal{I}(r)} u_{t,i} \leq R_r, \quad \forall r \in [0, h-1],\ \forall t \in [0, T-1] \quad (5)$$

$$u_{t,i} = \sum_{l=0}^{(d_i-1)} a_{((t-l) \bmod T),i} \quad \forall t \in [0, T-1], \forall i \in E \quad (6)$$

$$\mathcal{T} = T \cdot \mathcal{K} + \mathcal{A}^{\text{Transpose}} \times [0, 1, \cdots, T-1]^{\text{Transpose}} \quad (7)$$

$$t_j - t_i \geq d_i - T \cdot m_{ij} \quad \forall (i,j) \in E \quad (8)$$

$$\sum_{t=0}^{T-1} a_{t,i} = 1 \quad \forall i \in [0, N-1] \quad (9)$$

$$t_i \geq 0, k_i \geq 0, a_{t,i} \geq 0 \text{ and } u_{t,i} \geq 0 \text{ are integers} \quad (10)$$
$$\forall i \in [0, N-1], \forall t \in [0, T-1]$$

4.2 Coloring Formulation for Fixed FU Assignment

Unfortunately, the ILP formulation presented in the previous section sometimes underestimates the number of function units required to execute a loop under fixed FU assignment. For instance, the schedule in Table 1 depicts **FP** operations in the loop kernel (times 7, 8, and 9) in boldface. Clearly at each of these times only two boldface **FP** instructions are executing. Since this loop kernel is repeatedly executed, we map times 7, 8, and 9 to 0, 1, 2 as shown in Figure 4(a).

The usage of **FP** units is shown in Figure 4(b). Note that the function unit used by **i2** wraps around from time 2 to 0. This is a problem. At time 2, **i2** begins executing on the function unit that was used by **i4** at times 0 and 1. Since each instruction is supposed to use the same FU on every iteration, this causes a problem at time 0, when **i2** is still executing on the FU needed by **i4**. The problem is that Equation 5 notes only the number of FU's in use at one time, i.e. the number of solid horizontal lines present at each of the 3 time steps in Figure 4(b).

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 [10]. We must ensure that the two fragments corresponding to **i2** get the same color, a fact represented by the dotted arc in Figure 4(b). In addition the arcs of **i2** overlap with both **i3** and **i4**, meaning **i2** must have a different color than either. Similarly **i3** and **i4** must have different colors than each other.

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

$$|c_i - c_j| \geq \frac{u_{t,i} + u_{t,j} - 1}{2} \quad (11)$$

The u variables are merely the U matrix from Section 4.1, which represents the resource usage pattern of Figure 4(b). The value of $u_{t,i}$ is 1 if instruction i is executing at time t and is 0 if it is not. The $u_{t,j}$ is analogous for instruction j. Here the colors c_i and c_j are the colors representing the FUs for instructions i and j. Both c_i and c_j are positive integers, and *they must be different if nodes i and j overlap during loop execution.* If instructions i and j overlap, then both $u_{t,i}$ and $u_{t,j}$ will be 1 at those times where the overlap occurs. At such times the value of the right hand side is $\frac{1+1-1}{2} = \frac{1}{2}$, thus forcing c_i and c_j to be different, as desired. On the other hand, if instructions i and j never overlap, then $u_{t,i}$ or $u_{t,j}$ or both will be 0 at all times t. This gives the right hand side a value of 0 or $-\frac{1}{2}$, and no matter the choice of colors (including $c_i = c_j$), Inequality (11) holds.

Alas, absolute value is not a linear operation, so Inequality 11 does not fit with the rest of our ILP framework. To surmount this problem, we adopt an approach outlined by Hu [12]. We introduce a set of $w_{i,j}$ integer, 0-1 variables, with one such variable for each pair of nodes using the same type of function unit. Roughly speaking these $w_{i,j}$ variables represent the sign of $c_i - c_j$. Using them, Equation (11) can be represented with the following equations:

$$c_i - c_j \geq \frac{u_{t,i} + u_{t,j} - 1}{2} - N \cdot w_{i,j} \quad (12)$$

$$c_j - c_i \geq \frac{u_{t,i} + u_{t,j} - 1}{2} - N \cdot (1 - w_{i,j}) \quad (13)$$

$$1 \leq c_k \leq N \quad \forall k \in [0, N-1] \tag{14}$$

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

Theorem 4.1 *Equations 12 – 14 require that two nodes i and j be assigned different colors (mapped to different function units) if and only if they overlap.*

For a proof, please see [2]

For our ILP formulation we require that there be at least as many function units of each type as colors. Hence we replace Equation 5 with:

$$R_r \geq c_i \quad \forall i \in \mathcal{I}(r),\ \forall r \in [0, h-1] \tag{15}$$

The complete ILP formulation with fixed FU assignment (i.e. for **Problem 1**) is obtained by adding Equations (12), (13), and (15) to the formulation in Section 4.1 and eliminating Equation (5) as redundant.

5 Scheduling Pipelines with Structural Hazards

Having provided a formulation to solve **Problem 1**'s twin requirements of a rate optimal software pipelined schedule and fixed FU assignment for non-pipelined execution units, we now extend that formulation to solve **Problem 2**'s requirement that this be done for function units with structural hazards.

5.1 ILP Formulation

In order to represent the usage of the various stages of a pipeline by each instruction, we use reservation tables [15]. A reservation table has s stages and d columns, where d is the execution time of an instruction in the pipeline. For simplicity sake we assume that an FU 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 at the end of this section. The number of time steps over which a particular stage s in a pipeline is used (not necessarily continuously) during the execution of an instruction is denoted by d_s. Clearly, for fixed FU assignment, $T \geq d_s$. Further, it is necessary that no stage in a pipeline is used by an instruction at two time steps that are equal mod T. Such a constraint, known as the modulo scheduling constraint [5, 11, 19], is necessary for fixed FU assignment. In the subsequent discussion we assume that the modulo scheduling constraint is always satisfied for the given T. Cases where the modulo scheduling constraint is not satisfied is beyond the scope of this paper. In the roughly 10% of cases where the modulo scheduling constraint is violated, we apply the technique described in [8]. Due to space restrictions, we omit details of that technique here.

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

(a) ($T = 4$)

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

(b) ($T = 2$)

Figure 5: Modified Reservation Tables

For each function unit whose execution time $d < T$, we extend the reservation table of the function unit to T columns by adding $(T-d)$ zero (column) vectors. Since $d < T$, each entry in the *modified reservation table* is at most 1. When $d > T$, we fold the reservation table for the function unit to T columns such that the t-th column in the original reservation table is 'merged' with the (t mod T)-th column in the modified reservation table. The 'merge' operation results in adding the respective entries in the two columns. Use of the modulo scheduling constraint guarantees that each entry in the modified reservation table is at most 1. As an example, the **FP** unit whose reservation table is in Figure 2(a) has execution time $d = 3$. Figure 5 shows its modified reservation table when $T = 2$ and $T = 4$.

The s-th 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 T. An entry $RT_r^s[t]$ is 1 if stage s is required t time steps after the initiation of an instruction (mod T). Using this vector, and the $\mathcal{A}$ matrix we can define the resource usage matrix U for pipelines with hazards. Unlike in the non-pipelined case, here the resource usage matrix U^s is defined for each stage s of the pipeline. However, as before, U^s is a 2-dimensional matrix defined over all time steps in the repetitive pattern and for all instructions that execute on this FU type.

$$U^s[t, i] = \sum_{l=0}^{(T-1)} a_{((t-l) \texttt{ mod } T), i} \cdot RT_r^s[l], \tag{16}$$

There is a similarity between Equations (2) and (16). The introduction of RT_r^s facilitates selectively choosing appropriate $a_{t,i}$'s. A randomly generated $\mathcal{A}$ matrix with two instructions and its corresponding U^3 matrix for stage 3 of the **FP** unit (see Figure 5(a)) are shown in Figure 6. The RT vector is also provided for clarity.

Using the resource usage matrix defined by Equation 16, the resource requirement of stage s in function unit type r at time step t in the repetitive pattern is

$$R_r^s(t) = \sum_{i \in \mathcal{I}(r)} U^s[t, i]$$

Thus the number of FUs of type r required for the schedule is computed by taking the maximum of $R_r^s(t)$

$$\mathcal{A}^{\text{Transpose}} = \begin{array}{|c|c|c|c|} \hline 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 \\ \hline \end{array}$$

$$| \leftarrow \quad T = 4 \quad \rightarrow |$$

$$RT_r^{s\,\text{Transpose}} = \begin{array}{|c|c|c|c|} \hline 0 & 1 & 1 & 0 \\ \hline \end{array}$$

$$U^s[t,i]^{\text{Transpose}} = \begin{array}{|c|c|c|c|} \hline 0 & 0 & 1 & 1 \\ \hline 1 & 1 & 0 & 0 \\ \hline \end{array}$$

Figure 6: Example $\mathcal{A}$, RT_r^s, and corresponding U^s.

over all time steps t in the repetitive pattern and over all stages s. That is, if R_r is the number of FU's of type r required for this schedule, then

$$R_r \geq R_r^s(t) \quad \forall s \in [0, (s_r - 1)] \text{ and } \forall t \in [0, T-1] \quad (17)$$

For the example shown in Figure 6, it can be seen that stage 3 of the **FP** unit can be shared by the instructions corresponding to the two rows of U.

As before, we minimize a weighted sum of R_r for each function unit of type r. Lastly, we adopt the coloring formulation developed in Section 4.2 to ensure fixed FU assignment. The complete formulation for the software pipelining problem with structural hazards is shown in Figure 7.

We briefly mention how our formulation can handle FUs with multiple reservation tables. This is important, because, in practice, an **FP** 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 i, we use the RT_i^s which is determined from the appropriate reservation table.

[ILP Formulation for Pipelines with Hazards]

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

subject to

$$\sum_{i \in \mathcal{I}(r)} U^s[t,i], \leq R_r \quad (18)$$

$$\forall s \in [0, s_r - 1], \forall t \in [0, T-1], \text{ and } \forall r \in [0, h-1]$$

$$c_i \leq R_r \quad \forall i \in \mathcal{I}(r), \text{ and } r \in [0, h-1] \quad (19)$$

$$c_i - c_j \geq \frac{U^s[t,i] + U^s[t,j] - 1}{2} - N w_{ij} \quad (20)$$

$$c_j - c_i \geq \frac{U^s[t,i] + U^s[t,j] - 1}{2} - N(1 - w_{ij}) \quad (21)$$

$$\forall i,j \in \mathcal{I}(r), \quad \forall t \in [0, T-1], \text{ and } \forall r \in [0, h-1]$$

$$\mathcal{T} = T \cdot \mathcal{K} + \mathcal{A}^{\text{Transpose}} \times [0, 1, \cdots, T-1]^{\text{Transpose}} \quad (22)$$

$$\sum_{t=0}^{T-1} a_{t,i} = 1 \quad \forall i \in [0, N-1] \quad (23)$$

$$t_j - t_i \geq d_i - T \cdot m_{ij} \quad \forall (i,j) \in E \quad (24)$$

$$U^s[t,i] = \sum_{l=0}^{(T-1)} a_{((t-l) \bmod T),i} \cdot RT_r^s[l] \quad (25)$$

$$\forall t \in [0, T-1], \forall i \in \mathcal{I}(r), \forall s \in [0, s_r - 1], r \in [0, h-1]$$

$$\{t_i \geq 0, k_i \geq 0, a_{t,i} \geq 0, c_i \geq 1, 0 \leq d_{i,j} \leq 1, U^s[t,i] \geq 0\} : \text{integers}$$

$$\forall i \in [0, N-1], \forall t \in [0, T-1], \forall s \in [0, s_r - 1] \quad (26)$$

Figure 7: ILP formulation for Pipelines with Hazards

6 Experimental Results

In this section we provide results of experiments using our framework. Due to the lack of commercial processors containing pipelined function units with hazards, we were forced to generate our own. However, we have tried to make them as plausible as possible by matching the execution latency and throughput to the PowerPC-604 architecture. In particular, we started with fully pipelined function units as in the real 604, but added an extra "hazard" use at time step 2 in Stage 1 of the **Load** unit and at time step 2 of Stage 4 of the **FP Mult** unit as depicted in Table 3. We re-emphasize that since our approach handles hazards, architects of future systems may opt to save chip space by using pipelines with hazards. As in the real PowerPC-604, the **FP divide** unit is non-pipelined with latency 17. Other units remain fully pipelined. The **Integer** ALU has a latency of 1, the **Floating Point Add** and the **Store** units are assumed to be fully pipelined with latency 3. Finally, there are assumed to be 2 **Integer** ALU's and one each of the other types.

We ran our experiments on a set of 1066 single-basic-block inner loops taken from `SPEC92` (`integer` and `floating point`), `linpack`, `livermore`, and the `NAS kernels`. The DDG's for the loops were obtained by instrumenting a highly-optimizing research compiler. The DDG's varied widely in size— from 1 node to 1301 nodes with a median of 10 nodes, a geometric mean of 12, and an arithmetic mean of 19. To solve the ILP's, we used the commercial program, *CPLEX*.

In order to deal with the fact that our ILP approach

	Time Steps		
	0	1	2
Stage 1	x		x
Stage 2		x	
Stage 3			x

Load Units

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

FP Mult Units

Table 3: Reservation Tables used in the experiments

can take a very long time on some loops, we adopted the following approach. *First*, we limited CPLEX to 3 minutes in trying to solve any single ILP, i.e. a maximum of 3 minutes was allowed to find a schedule at a given T. *Second*, for each loop, at most 6 different T values, from $[T_{lb}, T_{lb}+5]$, were tried until a schedule was found. Using this approach and these parameters, we were able to obtain optimal solutions in the large majority of cases. However, in the future we plan to more systematically investigate how the values of these parameters effect the optimality of the results.

To be specific, our approach found an optimal schedule $T = T_{min}$ for at least 75% of the loops. In fact, the optimal schedule is at the lower bound ($T = T_{lb}$) of what is possible in 69% of the loops as summarized in Table 4. As the gap indicates, 74 or 6% of the schedules were T_{min} but not T_{lb}. That is there was no feasible solution to the ILP at T_{lb} meaning that the lower bound was not tight in those cases. This fraction is similar to what others have found [13, 16] for function units without structural hazards. The third column of Table 4 is the arithmetic mean of the number of nodes in the DDG. As indicated by the third column our ILP approach *does* produce good schedules when the mean number of nodes in the DDG is less than 20.

Table 5 indicates how far the schedule found was from a possible T_{min} schedule. We say a *possible* T_{min} here since there is no evidence. CPLEX' 3 minute time limit expired without indicating whether or not a schedule exists for this T_{min} period. As can be seen from Table 5, for 51 loops our approach found schedules, but cannot tell whether these are *optimal*.

Another important consideration in evaluating these schedules is the number of registers they require. More than 85% have fewer than 64 variable lifetimes while 81% have fewer than 32 variable lifetimes. These numbers 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. Note that we can easily include register requirements in the ILP framework as demonstrated in [9], but for simplicity we choose not to do so in this discussion. [4]

Number of Loops	Initiation Interval	Mean # Nodes in DDG
735	$T = T_{lb}$	6
85	$T = T_{lb} + 1$	10
20	$T = T_{lb} + 2$	16
4	$T = T_{lb} + 3$	18
11	$T = T_{lb} + 4$	17
5	$T = T_{lb} + 5$	21

Table 4: Scheduling Performance for Schedules Found within Time Limit

$T - T_{min}$	0	1	2	3	4	5
# of Schedules	785	30	5	8	5	3

Table 5: Difference between T and T_{min} for Schedules Found within Time Limit

How long did it take to get these schedules? The median time was 2 seconds, while the geometric mean was 5 seconds. Full results are summarized in Table 6. The times to find a schedule in Table 6 may not be considered fast enough for some real compilers. However, we noticed in the course of our experiments that many loop bodies occur repeatedly in different programs. To be precise we found that out of our set of 1066 loops, only 455 are unique. One loop body was common to 73 different loops! This being the case, the compiler could use our ILP approach to precompute optimal schedules for the most commonly occurring loops. This scheme could also be tailored to individual users by adding new loops to the database as the compiler encounters them. In fact, the ILP computation could be run in the background, so that the user may get non-optimal code the first time his/her code is compiled, but on later compilations the desired schedule would be in the database.

7 Related Work

Software pipelining has been extensively studied [1, 4, 5, 7, 13, 16, 17, 18, 20, 21, 24, 26, 27]. Rau and Fisher provide a comprehensive survey of these works in [19]. As stated in [19], software pipelining methods vary in several aspects: (1) whether or not they consider finite resources, (2) whether they model simple

[4] Because of the way our compiler performed memory disambiguation, some of our DDG's contain edges with very large dependence distances—more than the number of iterations in the loop. Such edges result in very large, but fictitious register requirements.

Time (in Sec.)	< 1	1-2	2-5	5-10	10-20	20-30	30-60	60-120	120-240	240-300	300-600	> 600
Number of Loops	345	79	137	28	39	13	57	50	11	31	26	44

Table 6: Time (in seconds) required for scheduling

or complex resource usage, (3) whether the algorithm is one-pass, iterative, incremental, or exhaustive search based. In particular, in terms of resource constraints, the work reported in [1, 24, 18], does not consider any resource constraint while the methods reported in [7, 11, 17, 20, 26, 27] deal with function unit constraints but with simple resource usage. Both function unit and register resource constraints are considered in [13, 5]. Software pipelining methods for complex usage patterns with limited function units was dealt with by [4, 16]. The methods proposed in these works that aim to be implementable in a practical compiler all use heuristic approaches.

In this paper, we develop a clear mathematical formulation of the software pipelining problem. The use of an integer programming method may or may not be acceptable in a practical compiler. The proposed method is important for performance-critical kernel loops and is 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. In [9] we compared three heuristic algorithms with an ILP scheduling method for clean pipelines. In [6] Feautrier independently gave an ILP formulation similar to our earlier work. Our method is unique in that it combines scheduling and mapping in a unified framework and attempts to achieve an optimal solution. An advantage of our method is that it can be extended to handle multi-function pipelines as well. It can incorporate minimizing buffers (logical registers) as in [18] or minimizing the maximum number of live values at any time step in the repetitive pattern, as in the method proposed by Eichenberger, Davidson and Abraham [5].

8 Conclusions

In this paper we have proposed a method to formulate scheduling and mapping problems in a unified framework. Our method can handle non-pipelined execution units and pipelines with structural hazards. The schedules obtained are optimal (in terms of initiation interval) and run with a minimum number of function units (for a fixed FU assignment of each instruction in the loop). The proposed method can also handle multi-function pipelines. We have implemented our scheduling method and run experiments on a set of 1066 loops taken from various benchmark suites such as `SPEC92`, the `NAS kernels`, `linpack` and `livermore`. Our initial experiments with scheduling these loops on an extended PowerPC-604 architecture were encouraging, yielding rate-optimal schedules for more than 75% of the loops.

The scope of periodic schedules and the ILP-based solution framework can be extended to a large variety of loops with conditionals. For example, conditionals can be handled by techniques discussed in [27], while code size reduction can be achieved when using special hardware support such as rotating register files and predicated execution [21]. Furthermore, register constraints can be integrated in the framework as demonstrated in [9].

While the use of integer programming methods in practical compilers may not always be acceptable, we discovered that many loop bodies are common to a large number of loops, making it possible to pre-compute optimal schedules for these loops. This database could also be automatically updated with an individual user's loops. In addition the ILP approach can be used to evaluate other heuristic methods, and also to derive optimal schedules for performance-critical applications where the user is willing to pay a high compilation cost for savings in runtime.

An alternative to ILP is to enumerate all possible schedules. For some loops this works well—our initial results suggest that 75% of loops have a "relatively finite" number of schedules. However, efficient enumeration of schedules is a significantly more demanding task in terms of programming effort, both because of the algorithms required and the lack of off-the-shelf software to assist the effort. Many loops have a large number of optimal schedules — but for those that do not, enumeration can take considerably more time than the ILP approach to find an optimal schedule. Note that as soon as an optimal schedule is found, both approaches normally stop, even if all schedules have not been enumerated. In the future we plan to investigate the use of constraint programming languages for performing software pipelining.

Acknowledgements

Kemal Ebcioğlu, Mayan Moudgill, and Gabriel M. Silberman were instrumental in completing this paper. We wish to thank Qi Ning and the anonymous referees for their helpful suggestions. We are thankful to IBM for its technical support, and acknowledge the Natural Science and Engineering Research Council (NSERC) and MICRONET, Network Centres of Excellence, 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*, pages 308–317, Atlanta, Georgia, Jun. 22–24, 1988.

[2] E.R. Altman. *Two Approaches for Optimal Software Pipelining with Resource Constraints (In Preparation)*. PhD thesis, McGill U., Montréal, Qué., 1995.

[3] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. *IEEE Trans. on Computers*, pages 967-979, Aug. 1988.

[4] J. C. Dehnert and R. A. Towle. Compiling for Cydra 5. *Journal of Supercomputing*, 7:181–227, May 1993.

[5] 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*, pages 75–84, San Jose, Calif., Nov. 30–Dec.2, 1994.

[6] P. Feautrier. Fine-grain Scheduling under Resource Constraints. In *Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, Ithaca, USA, August 1994.

[7] F. Gasperoni and U. Schwiegelshohn. Efficient algorithms for cyclic scheduling. Res. Rep. RC 17068, IBM T. J. Watson Res. Center, Yorktown Heights, 1991.

[8] R. Govindarajan, E. R. Altman, and G. R. Gao. Co-scheduling hardware and software pipelines. ACAPS Technical Memo 92, School of Computer Science, McGill University, Montreal, Que., 1995.

[9] 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*, pages 85–94, San Jose, Calif., Nov. 30–Dec.2, 1994.

[10] L. J. Hendren, G. R. Gao, E. R. Altman, and C. Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. In U. Kastens and P. Pfahler, editors, *Proc. of the Intl. Conf. on Compiler Construction*, number 641 in Lec. Notes in Comp. Sci., pages 176–191. Springer-Verlag, Oct. 1992.

[11] P.Y.T. Hsu. Highly concurrent scalar processing. Technical report, University of Illinois at Urbana-Champagne, Urbana, IL, 1986. Ph.D. Thesis.

[12] T. C. Hu. *Integer Programming and Network Flows*, page 270. Addison-Wesley Pub. Co., 1969.

[13] R. A. Huff. Lifetime-sensitive modulo scheduling. In *Proc. of the SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pages 258–267, Albuquerque, N. Mex., Jun. 23–25, 1993.

[14] IBM/Motorola. *PowerPC 604 RISC Microprocessor Technical Summary*, 1994.

[15] P. M. Kogge. *The Architecture of Pipelined Computers*. McGraw-Hill Book Company, New York, N. Y., 1981.

[16] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proc. of the SIGPLAN '88 Conf. on Programming Language Design and Implementation*, pages 318–328, Atlanta, Georgia, Jun. 22–24, 1988.

[17] 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*, pages 55–71, Portland, Ore., Dec. 1–4, 1992.

[18] 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*, pages 29–42, Charleston, South Carolina, Jan. 10–13, 1993.

[19] B. R. Rau and J. A. Fisher. Instruction-level parallel processing: History, overview and perspective. *J. of Supercomputing*, 7:9–50, May 1993.

[20] 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.*, pages 183–198, Chatham, Mass., Oct. 12–15, 1981.

[21] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. In *Proc. of the SIGPLAN '92 Conf. on Programming Language Design and Implementation*, pages 283–299, San Francisco, Calif., Jun. 17–19, 1992.

[22] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of the 27th Ann. Intl. Symp. on Microarchitecture*, pages 63–74, San Jose, Calif., Nov. 30–Dec.2, 1994.

[23] R. Reiter. Scheduling parallel computations. *J. of the ACM*, 15(4):590–599, Oct. 1968.

[24] 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*, number 634 in Lec. Notes in Comp. Sci., pages 613–624, Lyon, France, Sep. 1–4, 1992.

[25] S. R. Vegdahl. A Dynamic Programming Technique for Compacting Loops. In *Proc. of the 25th Ann. Intl. Symp. on Microarchitecture*, pages 180–188, Portland, Ore., Dec. 1–4, 1992.

[26] J. Wang and E. Eisenbeis. A new approach to software pipelining of complicated loops with branches. Res. rep., INRIA, Rocquencourt, France, Jan. 1993.

[27] N. J. Warter, J. W. Bockhaus, G. E. Haab, and K. Subramanian. Enhanced modulo scheduling for loops with conditional branches. In *Proc. of the 25th Ann. Intl. Symp. on Microarchitecture*, pages 170–179, Portland, Ore., Dec. 1–4, 1992.