McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures
and Parallel Systems

Minimizing Register Requirements under Resource-Constrained Rate-Optimal Software Pipelining

R. Govindarajan
Erik R. Altman
Guang R. Gao

ACAPS Technical Memo 80 (Revised)
June 1, 1995

govind@cs.mcgill.ca, erik@cs.mcgill.ca, gao@cs.mcgill.ca

Abstract

The rapid advances in high-performance computer architecture and compilation techniques provide both challenges and opportunities to exploit the rich solution space of software pipelined loop schedules. In this paper, we develop a framework to construct a software pipelined loop schedule which runs on the given architecture (with a fixed number of processor resources) at the maximum possible iteration rate (à la *rate-optimal*) while minimizing the number of buffers — a close approximation to minimizing the number of registers.

The main contributions of this paper are:

- First, we demonstrate that such problem can be described by a simple mathematical formulation with precise optimization objectives under a periodic linear scheduling framework. The mathematical formulation provides a clear picture which permits one to visualize the overall solution space (for rate-optimal schedules) under different sets of constraints.
- Secondly, we show that a precise mathematical formulation and its solution *does* make a significant performance difference! We evaluated the performance of our method against three other leading contemporary heuristic methods: Huff's Slack Scheduling [19], Wang, Eisenbeis, Jourdan and Su's FRLC [36], and Gasperoni and Schwiegelshohn's modified list scheduling [15]. Experimental results show that the method described in this paper performed significantly better than these methods.

The techniques proposed in this paper are useful in two different ways:

(i) As a compiler option which can be used in generating faster schedules for performance-critical loops (if the interested users are willing to trade the cost of longer compile time with faster runtime).

(ii) Provide a framework for compiler writers to evaluate and improve other heuristics-based approaches by providing quantitative information as where and how much their heuristic methods could be further improved.

Contents

List of Figures

List of Tables

1 Introduction

Software pipelining has been proposed as an efficient method for loop scheduling. It derives a static parallel schedule — a periodic pattern — that overlaps instructions from different iterations of a loop body. Software pipelining has been successfully applied to high-performance architectures [1, 2, 3, 7, 11, 10, 19, 21, 22, 23, 24, 29, 30, 34]. Today, rapid advances in computer architecture — hardware and software technology — provide a rich solution space involving a large number of schedules for software pipelining. In exploiting the space of good compile-time schedules, it is important to find a fast, software-pipelined schedule which makes the best use of the machine resources — both function units and registers — available in the underlying architecture.

In this paper, we are interested in addressing problems facing software pipelining such as:

> **Problem 1:** [*OPT*] Given a loop $\mathcal{L}$ and a machine architecture $\mathcal{M}$, construct a schedule that achieves the highest performance of $\mathcal{L}$ within the resource constraints of $\mathcal{M}$ while using the minimum number of registers.

The performance of a software-pipelined schedule can be measured by the *initiation rate* of successive iterations. Thus "highest performance" refers to the "fastest schedule" or to the schedule with the maximum initiation rate. A schedule with the maximum initiation rate is called a *rate-optimal* schedule.

The following two important questions are related to Problem 1, the *OPT* problem.

Question 1: Can a simple mathematical formulation be developed for the *OPT* problem?

Question 2: Does the optimality formulation pay off in real terms ? We need to answer the question "So what, after all ?"

In order to answer **Question 1**, we consider an instance of **Problem 1**. That is,

> **Problem 2:** [*OPT-T*] Given a loop $\mathcal{L}$ a machine architecture $\mathcal{M}$, and an iteration period T, construct a schedule, if one exists, with period T satisfying the resource constraints of $\mathcal{M}$ and using the minimum number of registers.

In this paper we consider target architectures involving both pipelined and non-pipelined execution units. Our approach to solving the *OPT-T* problem is based on a periodic scheduling framework for software pipelining framework [35, 24]. Based on the periodic scheduling framework, we present a simple Integer Linear Programming (ILP) formulation for *OPT-T*. We are able to express the resource constraints as linear constraints — combining such resource constraints with the work by Ning and Gao, where a tight upper bound for register requirement is specified using linear constraints [24], a unified formulation for the *OPT-T* problem is obtained. As in [24], we use FIFO buffers to model register requirement in this paper.

To efficiently solve the *OPT-T* problem, tight bounds for the integer variables are obtained, using methods similar to *As Soon As Possible (ASAP)* and the *As Late As Possible (ALAP)* schedules. Further, a lower bound on the number of buffers is obtained from Ning/Gao's linear program formulation [24]. (The relationship between the Ning/Gao formulation and ours can be better understood by examining Fig. 2 (page 10) in which the tradeoff between buffer and function unit optimality is depicted.) These bounds drastically reduce the search space of the integer program which facilitates obtaining the optimal solution in a reasonable computation time.

Readers who are familiar with related work in this field will find the optimality objective in the above problem formulation to be very ambitious. Of course, the general complexity of the optimal solution is NP-Hard, and heuristics are needed to solve the problem efficiently. However, we feel that a clearly stated optimality objective in the problem formulation is quite important for several reasons:

1. The solution space of "good" schedules [1] has increased considerably with the rapid advances in high-performance architecture. Current and future generation processors are likely to contain multiple function units. Likewise, in compilers, advances made in dependence analysis (such as array dataflow analysis [12] and alias analysis [16]) will expose more instruction-level parallelism in the code, while loop unrolling, loop fusion and other techniques will increase the size of the loop body [8]. So a given loop is likely to have many good schedules to choose from, and optimality criteria are essential to guide the selection of the best ones.

2. There are always a good number of users who have performance-critical applications. For them, the runtime performance of these applications is of utmost concern. For these applications, the user may be willing to trade a longer compilation time for an improvement in the runtime speed. Compilers for future generation high-performance architectures should not deny such opportunities to these users. The techniques developed in this paper can be provided to such users via a compiler option.

3. The techniques proposed in this paper can also be used in a scheduling framework to ascertain the optimal solution so as to evaluate and improve existing/newly proposed heuristic scheduling methods.

4. Lastly, the use of an integer program solver to generate schedules inside a production compiler might be debatable. 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 1008 loops, only 415 are unique. Further a particular 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.

[1] A discussion on the solution space of software-pipelined schedules is presented in Section 3.

Thus the usefulness of the techniques proposed in this paper should be viewed in the light of items (1) to (4) above.

We have implemented the solution method and tested it on 1008 loops extracted from various benchmark programs such as the `SPEC92`, the `NAS kernels`, `linpack`, and the `livermore loops`. The loops were scheduled for different architectural configurations involving pipelined or non-pipelined execution units. In our experiments, we were able to obtain the optimal schedule for more than 80% of the test cases considered. These experiments, run on a Sparc 20, required an execution with median ranging from 0.6 to 2.7 seconds for the different architecture configurations. The geometric mean of execution time ranged from 0.9 to 7.4 seconds.

Question 2, the "So what?" question, has been addressed by comparing our method with 3 other approaches, Huff's Slack Scheduling [19], Wang, Eisenbeis, Jourdan and Su's FRLC variant of DESP–Decomposed Software Pipelining [36], and Gasperoni and Schwiegelshohn's modified list scheduling approach [15]. We have implemented our solution method to the *OPT-T* and the *OPT* problems as well as the above three heuristic methods in an experimental scheduling testbed. We have measured the performance of various scheduling methods on the 1008 kernel loops. The ILP approach yielded schedules that are faster in 6% of the test cases compared to Slack Scheduling, in 21% of the test cases compared to the FRLC method, in 27% of the test cases compared to the modified list scheduling. In terms of buffer requirement, the ILP approach did significantly better than the three heuristic methods in, respectively, 61%, 87%, and 83% of the test cases.[2]

In this paper we have concentrated only loop bodies without conditional statements. Huff [19] has observed that over 92% of loops from a variety of benchmarks, including the SPEC89 Fortran benchmarks and the Perfect Club codes, have bodies without conditional statements. However, for loops involving conditional statements, our approach can be handled by techniques discussed in[38]. Code size reduction can be achieved when using special hardware support such as rotating register files and predicated execution [30].

Finally, as it will become evident, the proposed framework can easily handle other optimization problems in software pipelining. For example, given the number of available registers, it can minimize either the number of required FUs or a weighted sum of the FUs in different FU types. Other possible problem formulations can be observed from Figure 2 (refer to page 10).

This paper is organized as follows. In the following section, we motivate our approach with the help of an example. The solution space of software pipelined schedules is discussed in Section 3. In Section 4, the formulation of the *OPT-T* problem for pipelined execution units is developed. The *OPT-T* formulation for non-pipelined function units is presented in Section 5. Section 6 deals with an iterative solution to the *OPT* problem. In Section 7, the results of scheduling the 1008 benchmark loops are reported. Our ILP schedules are compared with the schedules generated by other leading heuristic methods in Section 8. In Section 9, we discuss other related work. Concluding remarks are presented in Section 10.

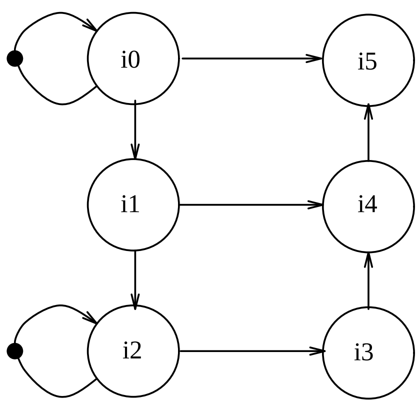
[2] For a small number of test cases, less than 4%, the ILP schedule were worse in terms of either initiation rate or buffer requirement. But this is due to fact that we limit our ILP search to a maximum 3 minutes. More details on the results are presented in Section 7.

2 Background and Motivation

In this section, we motivate the *OPT* problem and the solution method to be presented in the rest of this paper with the help of a program example.

2.1 Motivating Example

We introduce the notion of rate-optimal schedules under resource constraints, and illustrate how to search among them the ones which optimize the register usage. A more rigorous introduction to these concepts will be given in the next section. We adopt as our motivating example the loop $\mathcal{L}$ in Figure 1 given by Rau et al in [30].



(a) Dependence Graph

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

i_0:	vr33 = vr33 + vr32	% vr33 is address of a[i] %
i_1:	vr34 = load m(vr33)	% vr34 = a[i] %
i_2:	vr35 = vr35 + vr34	% vr35 = s %
i_3:	vr36 = vr35 * vr35	
i_4:	vr37 = vr36 * vr34	% vr37 = new a[i] %
i_5:	store(vr37, m(vr33))	
	branch to i_0 if $i \leq n$	
	enddo	

(b) Program Representation

Figure 1: An Example Loop

Both C language and instruction level representations of the loop are given in Fig. 1(b) while the dependence graph is depicted in Figure 1(a). Assume that instruction i_0 is executed in an **Integer FU** with an execution time of 1 time unit. Instructions i_2 (Floating Point (FP) ADD) and instructions i_3 and i_4 (FP MULTIPLY) are executed in an **FP** Unit with an execution time of 2 time units. Lastly, the FP LOAD (i_1) and FP STORE (i_5) are executed by a **Load/Store** Unit with execution times of 2 and 1 time units respectively. We will assume an architecture with 3 **Integer FU**s, 2 **FP** Units and 1 **Load/Store** unit. Further, in this subsection, we will assume that all pipelined function units are free of structural hazards and an operation can be initiated in each function unit at each time step. Scheduling non-pipelined function units are discussed in Section 2.3.

The performance of a software-pipelined schedule for $\mathcal{L}$ can be measured by the *initiation rate* of successive iterations. In the following discussion, we often use the reciprocal of the initiation rate, the *initiation interval* T. Let us first establish a lower bound for T — i.e. the shortest initiation interval for loop $\mathcal{L}$ under various constraints. It is well known that, the initiation interval is governed by both loop-carried dependencies in the graph and the resource constraints presented by the architecture. Under the loop-carried dependency constraint, the shortest initiation interval, T_{dep}, is given by:

$$T_{dep} = \max_{\forall cycles\ C} \frac{d(C)}{m(C)}$$

where $d(C)$ is the sum of the delays (or latencies) of the instructions (or *nodes*) in cycle C of the dependence graph, and $m(C)$ is the sum of the dependence distances around cycle C [33]. Those cycles C_{crit} with the maximum value of $\frac{d(C_{crit})}{m(C_{crit})}$ are termed *critical cycles* of the graph. In our example graph, (refer to Fig. 1(a)), the self loop on instructions i_2 is the critical cycle. Thus, T_{dep} for the given dependency graph is 2.

Resource constraints (of the architecture) also impose a lower bound on the initiation interval. Each resource type (function unit), e.g. **Integer** FU, impose such a lower bound. The resource constraint bound is: is:

$$T_{res}(r) = \frac{\text{number of instructions that execute in FU type } r}{\text{number of FUs of type } r}.$$

In our example,

$$T_{res}(\textbf{Integer FU}) = \frac{1}{3}; \quad T_{res}(\textbf{FP Unit}) = \frac{3}{2}; \quad T_{res}(\textbf{Load/Store Unit}) = \frac{2}{1} = 2$$

The overall resource constraint bound on T, denoted by T_{res}, is

$$T_{res} = \max_r(T_{res}(r)) \quad \text{for all FU types} \quad r$$

Thus

$$T_{res} = \max(\frac{1}{3},\ \frac{3}{2},\ 2) = 2$$

Considering both dependence and resource constraints, the lower bound on minimum initiation interval (T_{lb}) for our example with pipelined FUs is

$$T_{lb} = \max\{\lceil T_{dep}\rceil, \lceil T_{res}\rceil\} = \max\{2, 2\} = 2$$

That is, any schedule of loop $\mathcal{L}$ that obeys the resource constraint will have a period greater than or equal to $T_{lb} = 2$. The smallest iteration period $T_{min} \geq T_{lb}$, for which a resource-constrained schedule exists, is called the rate-optimal period (with the given resource constraints) for the given loop. It can be observed that the initiation rate $\frac{1}{T_{lb}}$ for a given DDG may be improved by unrolling the graph a number of times. The unrolling factor can be decided based on either the T_{dep} or the T_{res} value, or on both. However, for the purpose of this paper, we do not consider any unrolling of the graph. Though the techniques developed in this paper can be used in those cases as well.

2.2 An Illustration of the *OPT* Problem

In this paper we investigate periodic *linear schedules*, under which the time the various operations begin their execution are governed by a simple linear relationship. That is, under the

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

Table 1: Schedule A for the Motivating Example

linear schedule considered in this paper, the j-th instance of an instruction i begins execution at time $T \cdot j + t_i$, where $t_i \geq 0$ is an integer offset and T is the initiation interval or the *iteration period* of the given schedule. ($\frac{1}{T}$ is the *initiation rate* of the schedule.)

Table 1 gives a possible schedule (Schedule A) with period 2 for our example loop. This schedule is obtained from the linear schedule form $T \cdot j + t_i$, with T = 2, t_{i_0} = 0, t_{i_1} = 2, t_{i_2} = 4, t_{i_3} = 7, t_{i_4} = 9, and t_{i_5} = 11. Schedule A has a prologue (from time step 0 to time step 9) and a repetitive pattern (at time steps 10 and 11). During the first time step in the repetitive pattern (time step 10), 1 FP instruction (i_2), 1 Integer instruction (i_0), and 1 store instructions are executed, requiring 1 **FP Unit**, 1 **Integer** FU and 1 **Load/Store** Unit. Instructions i_3, i_4 and i_5 are executed during the second time step (time step 11), requiring 2 **FP** Units and 1 **Load/Store** Unit. Since this resource requirement of the repetitive pattern is less than what is available in the architecture, it is a resource-constrained schedule. Further, Schedule A is one of those resource-constrained schedules which achieves the fastest initiation interval (T_{min} = 2).

Next let us compute the register requirement for this schedule. In Schedule A, the instruction i_0 fires six times before the first i_5 fires. Since there is a data dependence between i_0 and i_5, the values produced by i_0 must be buffered and accessed by i_5 in order to insure correct execution of the program. Conceptually, some sort of FIFO buffers need to be placed between producer and consumer nodes. In this paper we will assume that a buffer is reserved at a time step when the instruction is issued, and remain reserved until the last instruction consuming that value completes its execution. The size of each buffer depends on the lifetime of the value. Therefore, a buffer of size 6 needs to be allocated for instruction i_0. As another example, four instances of i_1 are executed before the execution of the first instance of i_4. Consequently, a buffer size of 4 is required for instruction i_1. In a similar way a buffer size of 1 each is required for instructions i_3, and i_4, and a buffer size of 2 is required for i_2. Instruction i_5 is a STORE with no successor instructions. Since STORE has latency 1, i_5 requires 1 buffer. Thus, a total buffer size of 15 is required for this schedule as shown below.

Instruction						Total Buffers
i_0	i_1	i_2	i_3	i_4	i_5	
6	4	2	1	1	1	15

These conceptual FIFO buffers can either be directly implemented using dedicated architecture features such as circular buffers or rotating registers [32], or be mapped to physical registers (with appropriate register moves) on conventional architectures as described in [21, 25]. In [25], it was demonstrated that the minimum buffer requirement provides a very tight upper bound on the total register requirement, and once the buffer assignment is done, a classical graph coloring method can be subsequently performed which generally leads to the minimum register requirement. In this paper, we assume that such a coloring phase will always be performed once the buffer size is determined. Consequently we restrict our attention to these FIFO buffers or logical registers.

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

Table 2: Schedule B for the Motivating Example

A question of interest is: do there exist other rate-optimal schedules of $\mathcal{L}$ with the same resource constraint, but which use fewer registers? This is exactly what we have posed as Problem 1 (the *OPT* problem) in the introduction: The answer is affirmative, and is illustrated by Schedule B in Table 2 which uses only 14 buffers. This schedule is also resource constrained with an iteration period 2. The values of t_s for the instructions are

$$t_{i_0} = 0 \quad t_{i_1} = 1 \quad t_{i_2} = 3 \quad t_{i_3} = 6 \quad t_{i_4} = 8 \; and \; t_{i_5} = 10.$$

The buffer requirements for this schedule are as shown below:

Instruction						Total
i_0	i_1	i_2	i_3	i_4	i_5	Buffers
5	4	2	1	1	1	14

It may be verified that no schedule with period 2, satisfying the resource constraint, uses less than 14 buffers. Thus Schedule B is the solution we sought for the *OPT* problem — a rate-optimal schedule for the given loop $\mathcal{L}$. Note that we generated this schedule using the method outlined in Section 4.3.

2.3 A Schedule for Non-Pipelined FUs

Next let us focus on the issues involved in scheduling non-pipelined FUs. When the FUs are non-pipelined, each instruction initiated on an execution pipe continues to keep the FU busy until it completes its execution. Thus the T_{res} lower bound for non-pipelined FUs is:

$$T_{res}(r) = \frac{\text{sum of the execution time of instructions that execute in FU type } r}{\text{number of FUs of type } r}.$$

For the motivating example of Section 2.1,

$$T_{res}(\textbf{Integer FU}) = \frac{1}{3}; \quad T_{res}(\textbf{FP Unit}) = \frac{2+2+2}{2} = 3: \quad T_{res}(\textbf{Load/Store}) = \frac{2+1}{1} = 3$$

Thus,

$$T_{res} = \max(\frac{1}{3},\ 3,\ 3) = 3$$

The lower bound T_{lb} is

$$T_{lb} = \max(\lceil T_{dep} \rceil, \lceil T_{res} \rceil) = \max(2, 3) = 3$$

	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	$\_\_i_3$	i_4	$\_\_i_4$	i_5				
Iteration = 1				i_0	i_1	$\_\_i_1$	i_2	$\_\_i_2$	i_3	$\_\_i_3$	i_4	$\_\_i_4$	i_5	
Iteration = 2							i_0	i_1	$\_\_i_1$	i_2	$\_\_i_2$	i_3	$\_\_i_3$	i_4
Iteration = 3										i_0	i_1	$\_\_i_1$	i_2	$\_\_i_2$

Table 3: Schedule C for Non-Pipelined Execution Units

A schedule, Schedule C, for non-pipelined FUs is shown in Table 3. In this table we use the notation, e.g. $\_i_2$ to indicate that instruction i_2 continues its execution from the previous time step. The repetitive pattern, starting at time step 9, indicates that during each time step at most 2 **FP**, 1 **Integer**, and 1 **Load/Store** Units are required. Thus, it appears that Schedule C is a resource-constrained rate-optimal schedule for non-pipelined FUs. Unfortunately, this schedule is not legal. This is because, for Schedule C, we cannot find a *fixed* assignment of instructions to FUs. By this we mean that a compile-time mapping of instructions to specific FUs cannot be done for the repetitive pattern. To see this, consider the repetitive pattern starting at time step 9. If we assign the first **FP** unit to instruction $i2$ at time step 9, and the second **FP** unit to $i4$ at time step 10, then we have the first **FP** unit free at time step 11 and the second **FP** unit free at time 12 (or time step 9, taking the time steps with *modulo* 3). But mapping i_3 to the first **FP** unit at time step 11 and to the second **FP** unit at time 9 implies that the instruction i_3 migrates or switches from one FU to another during the course of its execution. Such a switching is impractical. In order to ensure that an instruction do not switch FUs during its execution, we require that there be a *fixed* assignment of instructions to FUs. Unfortunately, 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).

As indicated in the above example, for architectures with non-pipelined FUs, the software pipelining problem involves not only **instruction scheduling** (when each instruction is scheduled for execution) but also **mapping** (how instructions are assigned to FUs). Thus, to obtain rate-optimal resource-constrained software pipelining, we need to formulate the two related problems, namely scheduling and mapping, in a unified framework. Section 5 discusses such a formulation for non-pipelined FUs.

Table 4 shows a correct software pipelined schedule for the motivating example. In this schedule, instructions i_3 and i_4 share the first **FP** unit while i_2 executes on the second **FP** unit. Note that the period of the schedule is $T = 4$.

	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	i_4	--i_4			i_5		
Iteration = 1					i_0	i_1	--i_1	i_2	--i_2	i_3	--i_3	i_4	--i_4	
Iteration = 2									i_0	i_1	--i_1	i_2	--i_2	i_3
Iteration = 3													i_0	i_1

Table 4: Schedule D with Fixed FU Assignment

In order to give a proper perspective of problems addressed in this paper, a discussion on the solution space of linear schedules is presented in the following section.

3 The Solution Space of Linear Schedules

This section presents an overall picture of the solution space for periodic linear schedules P with which we are working. Within this space, the set of periodic linear schedules our interest is only in those periodic schedules which use R function units or less, which is denoted by the region labeled R. Obviously R is a subset of P. It may be noted that the *initiation intervals* of some of the schedules in R can be greater than or equal to T_{min} defined in Section 2.1. Since we are interested in rate-optimal schedules, we denote all schedules with period T_{min} by the region labeled T. There can be periodic schedules in T which use more than R function units.

The intersection of the sets T and R refers to the set of schedules with a period T_{min} and using R or less function units. This is denoted by the region labeled TR. The schedules in TR are rate-optimal under the resource constraint R — that is there is no schedule which uses not more than R resources, and has a faster initiation interval. In our example loop $\mathcal{L}$, Schedule A is an element of TR. By the definition T_{min}, it is guaranteed that there exists at least one schedule with $T = T_{min}$ and uses R or less resources. Hence TR is always nonempty.

To optimally use the available registers in the architecture, it is important to pick, in TR, a schedule that uses minimum registers. The set of such schedules is denoted by the region labeled TRB. Therefore, to put our problem statement in proper perspective, the goal in the *OPT* problem (See Introduction **Problem 1**) is to find a linear schedule which lies within region TRB. Note that the existence of such a schedule is guaranteed, from the fact that region TR is nonempty and the definition of set TRB. In our example, Schedule A is not a member of TRB while Schedule B is.

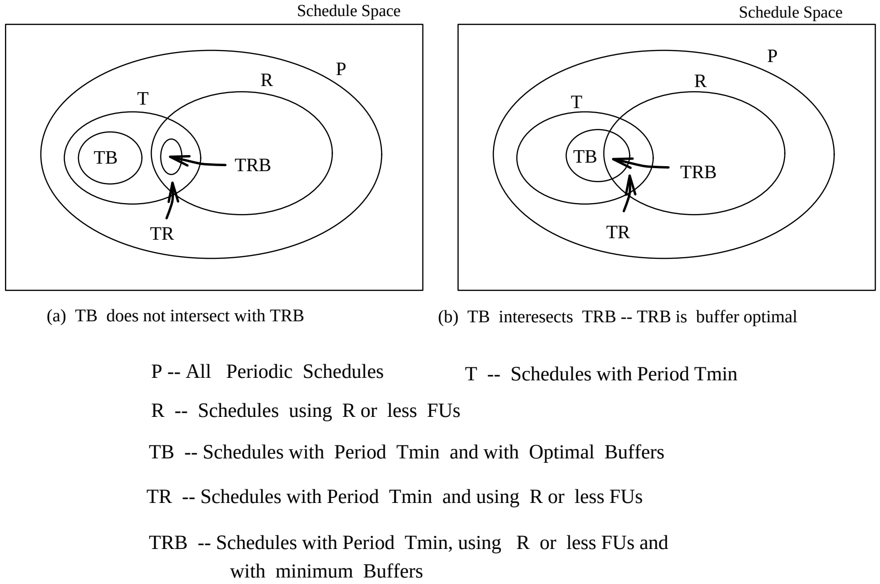


Figure 2: Schedule Space of a Given Loop

In Figure 2 there is a region labeled TB which denotes the set of all schedules with an initiation interval T_{min} that use the minimum number of registers. That is, for the initiation interval T_{min}, there may be schedules which use fewer registers than those in TRB. However, a schedule in TB may or may not satisfy the resource constraint R. In our example loop $\mathcal{L}$, in fact, the intersection of TB and R is empty. Figure 2(a) depicts this situation. Of course, this is not always the case. Fig. 2(b) represents the case when TB intersects R. Notice that in this case, TRB is a subset of TB.

An interesting feature of the TB region is that a schedule belonging to TB can be computed efficiently using a low-degree polynomial time algorithm developed by Ning and Gao [24]. As alluded to in the Introduction, this fact will be used as a key heuristic later in searching for a solution in TRB. More specifically, the register requirement of a TB schedule is used as a lower bound for the number of registers in the *OPT* problem.

4 *OPT-T* Formulation for Pipelined FUs

In this section, we first briefly introduce some background material. In the subsequent subsection, we develop the integer program formulation for the *OPT-T* problem. In Section 4.3, the *OPT-T* formulation for the motivating example of Fig. 1 is shown.

4.1 Definitions

This paper deals only innermost loops. We represent such loops with a Data Dependence Graph (DDG), where nodes represent instructions, and arcs the dependences between instructions.

With loop-carried dependences, the DDG could be cyclic. If node i produces a result in the current iteration and the result is used by node j, dd iterations later, then we say that the arc (i, j) has a *dependence distance* dd, and we use m_{ij} to denote it. In the DDG this is represented by means of dd initial tokens on the arc (i, j).

Definition 4.1 *A data dependence graph is a 4-tuple* (V, E, m, d) *where* V *is the set of nodes,* E *is the set of arcs,* $m = \{m_{ij}, \forall(i, j) \in E\}$ *is the dependence distance vector on arc set* E*, and* $d = \{d_i, \forall i \in V\}$ *is the delay function on node set* V*.*

In this paper we focus on the periodic schedule form $T \cdot j + t_i$ discussed in Section 2. A periodic schedule is said to be *feasible* if it obeys all dependence constraints imposed by the DDG. The following lemma due to Reiter [33] characterizes feasible periodic schedules.

Lemma 4.1 (Reiter [33]) *The initial execution times* t_i *are feasible for a periodic schedule with period* T *if and only if they satisfy the set of inequalities:*

$$t_j - t_i \geq d_i - T \cdot m_{ij} \tag{1}$$

where d_i *is the delay of node* i*,* T *the period, and* m_{ij} *the dependence distance for arc* (i, j)*.*

In this paper, we assume that the rate-optimal period T_{min} is always an integer. If not, the given DDG can be unrolled a suitable number of times, such that the resulting (unrolled) DDG has a integer period. Further, we have concentrated in this paper on straightline code. Huff found that a large majority of `FORTRAN` loops contain no conditionals [19]. For loops involving conditionals, we assume a hardware model that supports predicated execution as in [32]. If-conversion [4] can be performed to support this model. As well, in [30] it was shown that predicated execution simplifies code generation after modulo scheduling.

4.2 ILP Formulation

In order to represent the repetitive pattern, also known as the *modulo reservation table*, of the software pipelined schedule in a succinct form, we introduce the $\mathcal{A}$ matrix. The matrix $\mathcal{A}$ is a $T \times N$ matrix where T is the period of the schedule and N is the number of nodes in the DDG. The element $\mathcal{A}[t, i]$ is either 0 or 1 depending on whether instruction i is scheduled for execution at time step t in the repetitive pattern.

To make things clearer, consider the repetitive pattern of Schedule B. Here $T = 2$ and $N = 6$. The $\mathcal{A}$ matrix is:

$$\mathcal{A} = [a_{t,i}] = \begin{bmatrix} 1 & 0 & \mathbf{0} & \mathbf{1} & \mathbf{1} & 1 \\ 0 & 1 & \mathbf{1} & \mathbf{0} & \mathbf{0} & 0 \end{bmatrix}$$

The requirements of a particular FU type r at a time step t can be computed by adding the elements in row t which correspond to instructions executed by FU type r. For example, the number of **FP Units** required at each time step can be calculated by adding $a_{t,2}$, $a_{t,3}$, and $a_{t,4}$.

Thus, it can be seen that 2 **FP** units are required at time step 0 and 1 **FP Unit** is required at time step 1. Similarly by adding $a_{t,1}$ and $a_{t,5}$ we can observe that the number of **Load/Store** units required at each time step is 1. Thus, in general, the number of FUs of type r required at time t by a schedule is:

$$\sum_{i \in \mathcal{I}(r)} a_{t,i}$$

where $\mathcal{I}(r)$ denotes the set of instructions that execute in FU type r. If there are F_r FUs in type r, then resource constraints of the architecture can be specified as:

$$\sum_{i \in \mathcal{I}(r)} a_{t,i} \leq F_r \quad \text{for all } t \text{ and for all } r \tag{2}$$

Next we concentrate on some constraints on the $\mathcal{A}$ matrix. In order to ensure that each instruction is scheduled exactly once in the repetitive pattern, we require that the sum of each column in the above $\mathcal{A}$ matrix to be 1. This can also be expressed as a linear constraint as:

$$\sum_{t=0}^{T-1} a_{t,i} = 1 \quad \text{for all } i \in [0, N-1] \tag{3}$$

For Schedule B, the values of the t_i variables used in the linear form are:

$$t_0 = 0; \quad t_1 = 1; \quad t_2 = 3; \quad t_3 = 6; \quad t_4 = 8; \quad t_5 = 10;$$

The main question is how to relate the $\mathcal{A}$ matrix to the t_i variables. For this purpose we can rewrite each t_i as:

$$t_i = k_i * T + o_i \text{ such that } o_i \in [0, T-1] \tag{4}$$

Or in other words, k_i and o_i are defined as:

$$k_i = t_i \text{ div } T \quad \text{and } o_i = t_i \% T$$

where % represents the modulo operation. For Schedule B,

$$k_0 = 0; \quad k_1 = 0; \quad k_2 = 1; \quad k_3 = 3; \quad k_4 = 4; \quad k_5 = 5;$$

$$o_0 = 0; \quad o_1 = 1; \quad o_2 = 1; \quad o_3 = 0; \quad o_4 = 0; \quad o_5 = 0;$$

Observe both o_i and $\mathcal{A}[t, i]$ represent the position of instruction i in the repetitive pattern, perhaps in two different ways. Therefore, we can express o_i in terms of $\mathcal{A}[t, i]$ as:

$$\mathcal{O} = \begin{bmatrix} o_0 \\ o_1 \\ o_2 \\ o_3 \\ o_4 \\ o_5 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \\ 1 & 0 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{5}$$

Notice that the transpose of the $\mathcal{A}$ matrix is used in the above equation. That is,

$$\mathcal{O} = \mathcal{A}^{\text{Transpose}} \times [0, 1]^{\text{Transpose}}$$

Using this in Equation 4 and rewriting it in the matrix form, we obtain

$$\mathcal{T} = \begin{bmatrix} t_0 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \end{bmatrix} = \begin{bmatrix} k_0 \\ k_1 \\ k_2 \\ k_3 \\ k_4 \\ k_5 \end{bmatrix} \times T + \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \\ 1 & 0 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{6}$$

Or, in general,

$$\mathcal{T} = \mathcal{K} * T + \mathcal{A}^{\text{Transpose}} \times [0, 1, \cdots T-1]^{\text{Transpose}} \tag{7}$$

Lastly, we need to represent the register requirements of the schedule in a linear form. As mentioned earlier, in this paper, we model register requirements by FIFO buffers placed between producer and consumer nodes. Such an approach was followed in [24]. Further, we assume that buffer space is reserved as soon as the producer instruction commences its executions and remains reserved until the (last) consumer instruction begins its execution.

Consider an instruction i and its successor j. The result value produced by i is consumed by j after m_{ij} iterations. This duration, called the *lifetime* of the result, is equal to $(t_j + T \cdot m_{ij} - t_i)$ in the periodic schedule. During this time, i would have fired $(t_j + T \cdot m_{ij} - t_i)/T$ times, and therefore this many buffers are needed to store the output of i. If instruction i has more than one successor j, then the register requirement for i is the maximum of $(t_j + T \cdot m_{ij} - t_i)/T$, for all j. In other words, the number of buffers b_i associated with an instruction i is given by

$$b_i \geq \frac{t_j + T \cdot m_{ij} - t_i}{T}\ , \quad \forall j \text{ such th at } (i,j) \in E \tag{8}$$

Rewriting Equation (8), we get

$$T \cdot b_i + t_i - t_j \geq T \cdot m_{ij} \tag{9}$$

In [25], it was demonstrated that minimum buffer requirement provides a very tight upper bound on the total register requirement, and once the buffer assignment is done, a classical graph coloring method can subsequently be performed which generally leads to the minimum register requirement. In this paper, we assume that such a coloring phase will always be performed once the schedule is determined.

Now integrating the buffer requirements with our ILP formulation, we can obtain the formulation which minimizes the buffer requirements in constructing rate-optimal resource constrained schedules. For this purpose, the objective function is minimizing the total number of buffers used by the schedule. That is

$$\text{minimize} \sum_{i=0}^{N-1} b_i$$

The complete ILP formulation is shown in Figure 3.

[ILP Formulation for Pipelined FUs]

$$\text{minimize} \sum_{i=0}^{N-1} b_i$$

subject to

$$\sum_{i \in \mathcal{I}(r)} a_{t,i} \leq F_r, \quad \text{for all } t \in [0, T-1] \text{ and for all } r \tag{10}$$

$$T \cdot \mathcal{K} + \mathcal{A}^{\text{Transpose}} \times [0, 1, \cdots, T-1]^{\text{Transpose}} = \mathcal{T} \tag{11}$$

$$T \cdot b_i + t_i - t_j \geq T \cdot m_{ij} \text{ forall } i \in [0, N-1] \text{ and } (i,j) \in E \tag{12}$$

$$\sum_{t=0}^{T-1} a_{t,i} = 1 \quad \text{for all } i \in [0, N-1] \tag{13}$$

$$t_j - t_i \geq d_i - T \cdot m_{ij} \quad \forall (i,j) \in E \tag{14}$$

$$b_i \geq 0, t_i \geq 0, k_i \geq 0, \text{ and } a_{t,i} \geq 0 \text{ are integers } \quad \forall i \in [0, N-1], \forall t \in [0, T-1] \tag{15}$$

Figure 3: ILP formulation for Pipelined FUs

4.3 *OPT-T* Formulation for the Motivating Example

To illustrate the operation of the *OPT-T* formulation, we again examine the motivating example presented in Section 2.

The minimum iteration period for the DDG in Figure 1 is $T = 2$. Further there are $N = 6$ nodes. Equation (14) gives the dependence constraints for a feasible schedule:

$$\begin{array}{llll} t_1 - t_0 \geq 1 & t_5 - t_0 \geq 1 & t_2 - t_1 \geq 2 & t_4 - t_1 \geq 2 \\ t_3 - t_2 \geq 2 & t_4 - t_3 \geq 2 & t_5 - t_4 \geq 2 & \end{array} \tag{16}$$

Equation (13) requires that a node be scheduled exactly once:

$$\begin{array}{lll} a_{0,0} + a_{1,0} = 1 & a_{0,1} + a_{1,1} = 1 & a_{0,2} + a_{1,2} = 1 \\ a_{0,3} + a_{1,3} = 1 & a_{0,4} + a_{1,4} = 1 & a_{0,5} + a_{1,5} = 1 \end{array} \tag{17}$$

Equation (11) relates the elements of the A matrix to $\mathcal{K}$ and $\mathcal{T}$:

$$\begin{array}{ll} 2 \cdot k_0 + 0 \cdot a_{0,0} + 1 \cdot a_{1,0} = t_0 & 2 \cdot k_1 + 0 \cdot a_{0,1} + 1 \cdot a_{1,1} = t_1 \\ 2 \cdot k_2 + 0 \cdot a_{0,2} + 1 \cdot a_{1,2} = t_2 & 2 \cdot k_3 + 0 \cdot a_{0,3} + 1 \cdot a_{1,3} = t_3 \\ 2 \cdot k_4 + 0 \cdot a_{0,4} + 1 \cdot a_{1,4} = t_4 & 2 \cdot k_5 + 0 \cdot a_{0,5} + 1 \cdot a_{1,5} = t_5 \end{array} \tag{18}$$

The following three equations respectively represent the resource constraints for **Integer**, **Load/Store**, and **FP** units.

$$a_{0,0} \leq 3 \qquad a_{1,0} \leq 3 \tag{19}$$

$$a_{0,1} + a_{0,5} \leq 1 \qquad a_{1,1} + a_{1,5} \leq 1 \tag{20}$$

$$a_{0,2} + a_{0,3} + a_{0,4} \leq 2 \qquad a_{1,2} + a_{1,3} + a_{1,4} \leq 2 \tag{21}$$

The register requirement for each instruction is given by Equation (9). For the given DDG, these constraints are:

$$\begin{array}{ccccc} 2 \cdot b_0 + t_0 - t_1 \geq 0 & 2 \cdot b_0 + t_0 - t_5 \geq 0 & 2 \cdot b_1 + t_1 - t_2 \geq 0 & 2 \cdot b_1 + t_1 - t_4 \geq 0 & \\ 2 \cdot b_2 + t_2 - t_3 \geq 0 & 2 \cdot b_3 + t_3 - t_4 \geq 0 & 2 \cdot b_4 + t_4 - t_5 \geq 0 & 2 \cdot b_0 \geq 2 & 2 \cdot b_2 \geq 2 \end{array} \tag{22}$$

Finally, the objective is to minimize the total number of buffers $\sum_{i=0}^{N-1} b_i$ subject to the constraints in Equations 16 – 22, and that $a_{t,i}$, k_i, t_i, and b_i are non-negative integers. Solving this integer program formulation yields Schedule B.

In solving the above integer programming problem, we need to obtain values for all $a_{t,i}$ variables and k_i variables, and thus obtain the values for the t_i variables which determine the schedule. Each t_i variable can take values only within a specific range (determined by the dependences and the iteration period of the DDG), which in turn will restrict the range of t for which $a_{t,i}$ can take the value 1.

5 *OPT-T* Formulation for Non-Pipelined FUs

In this section we develop the formulation for the *OPT-T* problem for non-pipelined FUs. As illustrated in Section 2.3, this problem requires both scheduling and mapping to be performed simultaneously. In the following section we show how the resource usage for non-pipelined FUs can be modeled. The formulation of the mapping problem is discussed in Section 5.2.

5.1 Resource Usage in Non-Pipelined FUs

In order to estimate the resource requirements with non-pipelined FUs, we need to know not just when each instruction is *initiated* (given by the $\mathcal{A}$ matrix), but also how long each executes. For example, instruction i_4 in Schedule C is initiated at time step 10 (or time 1 in the repetitive pattern) and executes until time step 11. Equivalently, since the execution time of **FP Multiply** is 2 time units, i_4 executes until $(10 + 2 - 1) \ \% \ 3 = 2$ in the repetitive pattern. Or in other words, instruction i_4 requires an FU at time steps 1 and 2 in the repetitive pattern. Likewise, instruction i_3 requires an FU at time steps 2 and 0 in the repetitive pattern. Thus we need to define a usage matrix $\mathcal{U}$ from the $\mathcal{A}$ matrix to represent the usage of non-pipelined FUs.

First we illustrate the $\mathcal{A}$ matrix and the usage matrix $\mathcal{U}$ for Schedule C.

$$\mathcal{A} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \quad \text{and} \quad \mathcal{U} = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Notice that the **FP** instructions and the **Load** instructions which take 2 time units to execute, require the FU for more than one time step in the usage matrix. As before, adding the appropriate elements of each row gives the FU requirement for type r.

How do we obtain the $\mathcal{U}$ matrix from $\mathcal{A}$? An instruction i initiated at time $t \ \% \ T$ requires the FU until time step $(t + d_i - 1) \ \% \ T$ in the repetitive pattern. Alternatively, we can say that instruction i requires a function unit at time step t if i began execution less than d_i time steps prior to t. Thus we can define $\mathcal{U}[t, i]$ as:

$$\mathcal{U} = u_{t,i} = \sum_{l=0}^{(d_i-1)} a_{((t-l)\%T),i}, \quad \forall t \in [0, T-1], \text{ and } \forall i \in V \tag{23}$$

Notice that if the execution time $d_i = 1$ cycle, then $u_{t,i} = a_{t,i}$. 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, $u_{t,i} = a_{t,i}$.

In our example loop, instructions i_0 and i_5 take one time unit to execute. Hence

$$u_{t,i_0} = a_{t,i_0} \quad \text{and} \quad u_{t,i_5} = a_{t,i_5}$$

That is,

$$u_{0,i_0} = a_{0,i_0} \ ; \quad u_{1,i_0} = a_{1,i_0} \ ; \quad u_{2,i_0} = a_{2,i_0}$$
$$u_{0,i_5} = a_{0,i_5} \ ; \quad u_{1,i_5} = a_{1,i_5} \ ; \quad u_{2,i_5} = a_{2,i_5}$$

For instruction i_2, i_3 and i_4, $u_{t,i}$ is defined as:

$$u_{0,i_1} = a_{0,i_1} + a_{2,i_1} \ ; \quad u_{1,i_1} = a_{1,i_1} + a_{0,i_1} \ ; \quad u_{2,i_1} = a_{2,i_1} + a_{1,i_1} \ ;$$
$$u_{0,i_2} = a_{0,i_2} + a_{2,i_2} \ ; \quad u_{1,i_2} = a_{1,i_2} + a_{0,i_2} \ ; \quad u_{2,i_2} = a_{2,i_2} + a_{1,i_2} \ ;$$
$$u_{0,i_3} = a_{0,i_3} + a_{2,i_3} \ ; \quad u_{1,i_3} = a_{1,i_3} + a_{0,i_3} \ ; \quad u_{2,i_3} = a_{2,i_3} + a_{1,i_3} \ ;$$
$$u_{0,i_4} = a_{0,i_4} + a_{2,i_4} \ ; \quad u_{1,i_4} = a_{1,i_4} + a_{0,i_4} \ ; \quad u_{2,i_4} = a_{2,i_4} + a_{1,i_4} \ ;$$

The requirement for type r FUs at time step t is

$$\sum_{i \in \mathcal{I}(r)} u_{t,i}.$$

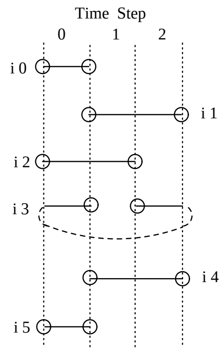
Since this should be less than the number of available FUs,

$$\sum_{i \in \mathcal{I}(r)} u_{t,i} \leq F_r \ \text{ for all } \ t \in [0, T-1] \text{ and for all } r \tag{24}$$

Replacing the resource constraint (Equation 10) in the ILP formulation (refer to Figure 3) by Equations 23 and 24, we obtain the scheduling part of the ILP formulation for non-pipelined FUs. However, as explained in Section 2.3, the complete formulation must include the mapping part (fixed FU assignment) as well. Otherwise the schedules produced by the formulation may require the switching instructions between FUs during the course of execution. In the following subsection we show how the mapping problem can also be formulated under the same framework.

Time Steps		
0	1	2
i_5		
--i_3	i_4	--i_4
i_2	--i_2	i_3
i_0	i_1	--i_1

(a)



(b)

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

5.2 Fixed FU Assignment

Consider Schedule C shown in Table 3. Since the loop kernel is repeatedly executed, we map times 9, 10, and 11 to 0, 1, and 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 **i3** wraps around from time 2 to 0. This is a problem. At time 2, $i3$ begins executing on the function unit that was used by $i2$ 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 $i3$ is still executing on the FU needed by **i2**. The problem is that Equation 24 only notes 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). However, we need to ensure that the two segments (corresponding to instruction i_3) get assigned to the same FU.

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 [17]. We must ensure that the two fragments corresponding to i_3 get the same color, a fact represented by the dotted arc in Figure 4(b). In addition the arcs of i_3 overlap with both i_2 and i_4, meaning i_3 must have a different color than either. Similarly i_2 and i_4 must have different colors than each other.

Now using the usage matrix, we can formulate the coloring problem using integer constraints. If two instructions i and j are executing at time t then clearly each must get a different FU assigned to it. That is, if c_i and c_j represent the colors (or function unit to which they are mapped to) of instructions i and j respectively, then $c_i \neq c_j$ if both $u_{t,i}$ and $u_{t,j}$ are 1. Such a constraint can be represented in integer programming by adopting the approach given by Hu [18]. 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 $c_i - c_j$.

$$c_i - c_j \quad \geq \quad \frac{u_{t,i} + u_{t,j} - 1}{2} - N \cdot w_{i,j} \tag{25}$$

$$c_j - c_i \quad \geq \quad \frac{u_{t,i} + u_{t,j} - 1}{2} - N \cdot (1 - w_{i,j}) \tag{26}$$

$$1 \leq c_k \leq N \quad \forall k \in [0, N-1] \tag{27}$$

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

In [5] we prove that the above constraints (Equations 25, 26 and 27) together guarantee that two nodes i and j are assigned different colors (mapped to different function units) if and only if they overlap. For our ILP formulation we require that there be at least as many function units as colors. Hence we replace Equation 24 with Equations 25 – 27 and

$$c_i \leq F_r \quad \text{for all } i \in [0, N-1] \text{ and } i \in \mathcal{I}(r)$$

The complete ILP formulation is shown in Figure 5.

[ILP Formulation for Non-Pipelined FUs]

$$\text{minimize } \sum_{i=0}^{N-1} b_i$$

subject to

$$c_i \leq F_r \quad \text{for all } i \in [0, N-1] \text{ and } i \in \mathcal{I}(r) \tag{28}$$

$$u_{t,i} = \sum_{l=0}^{(d_i-1)} a_{((t-l)\%T),i}, \quad \text{for all } t \in [0, T-1], \text{ and } \text{ and } i \in V \tag{29}$$

$$T \cdot \mathcal{K} + \mathcal{A}^{\text{Transpose}} \times [0, 1, \cdots, T-1]^{\text{Transpose}} = \mathcal{T} \tag{30}$$

$$T \cdot b_i + t_i - t_j \geq T \cdot m_{ij} \quad \text{for all } i \in [0, N-1] \text{ and } (i,j) \in E \tag{31}$$

$$\sum_{t=0}^{T-1} a_{t,i} = 1 \quad \text{for all } \ i \in [0, N-1] \tag{32}$$

$$c_i - c_j \quad \geq \quad \frac{u_{t,i} + u_{t,j} - 1}{2} - N \cdot w_{i,j} \text{ for all } i, j \in \mathcal{I}(r) \text{ and } r \tag{33}$$

$$c_j - c_i \quad \geq \quad \frac{u_{t,i} + u_{t,j} - 1}{2} - N \cdot (1 - w_{i,j}) \text{ for all } i, j \in \mathcal{I}(r) \text{ and } r \tag{34}$$

$$t_j - t_i \geq d_i - T \cdot m_{ij} \quad \text{for all } (i,j) \in E \tag{35}$$

$$1 \leq c_k \leq N \quad \text{for all } k \in [0, N-1] \tag{36}$$

$$b_i \geq 0, t_i \geq 0, k_i \geq 0, \text{ and } a_{t,i} \geq 0 \text{ are integers} \quad \text{for all } i \in [0, N-1], \text{ and } t \in [0, T-1] \tag{37}$$

Figure 5: ILP formulation for Non-Pipelined FUs

6 A Solution Method for *OPT* Problem

The successful formulation of the *OPT-T* problem provides the basis of our solution to the *OPT* problem. To solve the *OPT* problem, we need to iteratively solve the *OPT-T* formulation for increasing values of T starting from T_{lb} until we find a schedule satisfying the function unit constraint. In other words, T_{min} is the smallest value greater than or equal to T_{lb} for which a schedule obeying the resource constraint exists. We want to solve the *OPT-T* formulation with iteration period T_{min}. It has been observed that in most cases, T_{min} is at or near T_{lb} [21, 19]. Thus using an iterative search, starting at T_{lb} we can quickly converge to T_{min}.

In solving the ILP formulation of the *OPT-T* problem, we can guide our search by giving a lower bound on the number of buffers required. We illustrate this idea as follows. Let T be the smallest iteration period for which a schedule obeying the function unit constraint exists. For this value of T, by solving the minimum register optimal schedule formulation proposed by Ning and Gao [24], we can obtain a lower bound on the number of buffers. Ning and Gao's formulation is a linear program formulation and can be solved efficiently. However since this formulation [24] does not include resource constraints, the obtained schedule may or may not satisfy resource constraints.

7 Performance of ILP Schedules

In this section we present the performance results of the ILP scheduler. Section 8 is devoted to a comparison with heuristic methods.

We have implemented our ILP based software pipelining method on a UNIX workbench. We have experimented with 1008 single-basic-block inner loops extracted from various scientific benchmark programs such as `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. We have considered loops with up to 64 nodes in the DDG as in [19]. The DDG's varied widely in size, with a median of 7 nodes, a geometric mean of 8, and an arithmetic mean of 12.

To solve the ILP's, 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 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*, initiation intervals from $[T_{min}, T_{min} + 5]$ were tried if necessary. As soon as a schedule was found before $T_{min} + 5$, we did not try any greater values of T.

We have assumed the following execution latencies for the various instructions.

Instructions:	**Integer**	**FP Add**	**Load**	**Store**	**Multiply**	**Divide**
Clock cycle(s):	1	3	3	1	4	17

We applied our scheduling for different architectural configurations. We considered architectures with pipelined or non-pipelined execution units. We also considered architectures where the FUs are generic, i.e. each FU can execute any instruction. Such FUs are referred to as *homogeneous FUs*. A heterogeneous FU type, like **Load/Store** Unit, on the other hand, can only execute instructions of a specific type (or a class of types). The six different architectural configurations considered in our experiments are:

A1 – 6 pipelined homogeneous FUs

A2 – 4 pipelined homogeneous FUs

A3 – 6 non-pipelined homogeneous FUs

A4 – 4 non-pipelined homogeneous FUs

A5 – pipelined heterogeneous FUs (2 **Integer** FUs and one of **Load/Store**, **FP Add**, **Multiply** and **Divide** units.)

A6 – Same as **A5**, but function units are non-pipelined.

The 1008 loops were scheduled for each of these architectures.

In a large majority of cases, the ILP approach found an optimal schedule close to T_{min} as shown in Table 5. To be specific, for architectures with homogeneous pipelined FUs (**A1** and **A2**), the ILP approach found an optimal schedule in more than 88% of cases. For non-pipelined homogeneous FUs, an optimal schedule was found in 71% of the cases. Lastly, for architectures with heterogeneous FUs (**A5** and **A6**) it varies from 80% to 85%. For all architectural configurations, in a small fraction of the test cases, the ILP method found a schedule at a T greater than a *possible* T_{min}. That is, in these cases, the obtained schedule is a *possible* optimal schedule. We say a *possible* T_{min} and *possible* optimal schedule here since there is no evidence — CPLEX' 3 minute time limit expired without indicating whether or not a schedule exists for a lower value of T_{min}. Table 5 indicates how far the schedule found was from a *possible* optimal schedule.

Next we proceed to compare how close the ILP schedules were to the optimal buffer requirement. In deriving minimal buffer, rate-optimal schedules, CPLEX's 3 minutes time limit was sometimes exceeded before finding a buffer optimal schedule. In those cases we took the best schedule obtained so far. In other words, this could be one of the schedule from the set TR in Fig. 2. Once again, this schedule could *possibly* lie in TRB, but there is no evidence — for or against — as the 3 minute time limit of CPLEX was exceeded. We compare the buffer requirement of this schedule with that of a TB scheduled obtained from the Ning-Gao formulation [24]. We note again that the Ning-Gao formulation obtains minimal buffer, rate optimal schedules using linear programming techniques and does not include resource constraints. Thus the bound obtained from Ning-Gao's formulation is a loose lower bound, and there may or may not exist a resource-constrained schedule with this buffer requirement. Let us denote the buffer requirement of TB, TR, and TRB schedules by B_{TB}, B_{TR}, and B_{TRB} respectively. Then $B_{\mathsf{TR}} \geq B_{\mathsf{TRB}} \geq B_{\mathsf{TB}}$. To compare the quality of schedules, we take the

Initiation Interval	Number of Loops					
	A1	**A2**	**A3**	**A4**	**A5**	**A6**
$T = T_{min}$	946	882	714	699	854	792
$T = T_{min} + 1$	1	4	37	39	60	9
$T = T_{min} + 2$	0	24	9	10	18	9
$T = T_{min} + 3$	0	4	7	5	13	9
$T = T_{min} + 4$	6	1	1	17	9	9
$T = T_{min} + 5$	0	6	8	5	1	9
No Schedule found	55	87	232	233	53	166

Table 5: Schedule Quality in Terms of Iteration Period

minimum buffer requirement B_{min} as B_{TRB} if a TRB schedule is found and B_{TB} otherwise. Thus, when a TRB schedule is not found, B_{min} is an overly optimistic lower bound.

Table 6 shows the quality of ILP schedules in terms of their buffer requirements. Here we consider only those cases where the ILP approach found a schedule, optimal or otherwise. As can be seen from this table, the ILP approach produces schedules that require minimal buffers in 85% to 90% of the cases for architectures involving heterogeneous FUs (pipelined or non-pipelined) or homogeneous pipelined FUs (6 or 4 FUs). For architectures with homogeneous non-pipelined FUs (**A3** and **A4**) the quality of schedule, in terms of both computation rate ($1/T$) and buffer requirement is poor compared to all other architectural configurations. This is due to the increased complexity of mapping rather than scheduling. The complexity of mapping instructions to FUs is significantly higher for homogeneous FU than for heterogeneous FUs. This is because, each instruction can potentially be mapped to any of the FUs, and hence the overlap (in execution) of **all pairs of instructions** needs to be considered. On the other hand, in the heterogeneous model, we only need to consider **all pairs of instructions that are executed in the same FU type**.

Initiation Interval	Number of Loops					
	A1	**A2**	**A3**	**A4**	**A5**	**A6**
$B = B_{min}$	916	846	710	696	804	766
$B = B_{min} + 1$	0	1	32	28	33	17
$B = B_{min} + 2$	4	4	21	25	22	8
$B = B_{min} + 3$	1	5	7	8	20	33
$B = B_{min} + 4$	2	21	5	15	20	6
$B = B_{min} + 5$	2	17	1	1	6	6
$B > B_{min} + 5$	28	27	0	2	50	6

Table 6: Schedule Quality in Terms of Buffer Requirement

Finally, how long did it take to get these schedules? We measured the execution time (henceforth referred to as the compilation time) of our scheduling method on a Sun/Sparc20

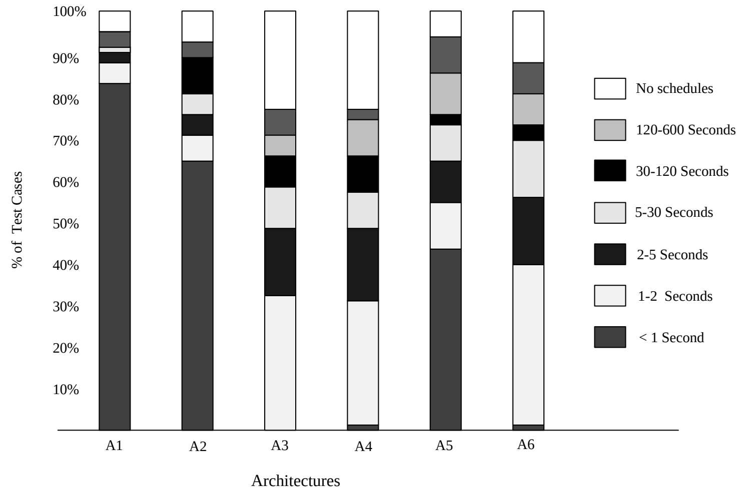


Figure 6: Histogram of Execution Time

workstation. The geometric mean, arithmetic mean, and median of the execution time for the 6 architectural configurations are shown in Table 7. A histogram of the execution time for various architectural configurations is shown in Figure 6. From Table 7 we observe that the geometric mean of execution time is less than is less than 2 seconds for architectures with homogeneous pipelined FUs and less than 5 seconds for architectures with heterogeneous FUs. The median of the execution time is less than 3 seconds for all cases. Architectural configurations **A3** and **A4** (with homogeneous non-pipelined FUs) required a larger execution time compared to other configurations due to increased complexity in mapping instructions.

Architecture	Execution Time		
	Geo. Mean	Median	Arith. Mean
A1	0.90	0.63	13.0
A2	1.80	0.65	35.9
A3	6.60	2.65	63.1
A4	7.40	2.73	74.5
A5	4.00	1.70	73.6
A6	4.70	2.35	55.7

Table 7: Average Execution Time to obtain ILP Schedules

8 Comparison with Heuristic Methods

Our extensive experimental evaluation indicates that the ILP approach can obtain the schedule for a large majority of the test cases reasonably quickly. But does the optimality objective and the associated computation cost pay off in terms computation rate or buffer requirement of the derived schedules? It is often argued that existing heuristic methods (without any mathematical optimality formulation) do very well and consequently there is no need to find optimal schedules. Our results indicate otherwise.

We consider 3 leading heuristic methods for comparative study. They are Huff's Slack Scheduling [19], Wang, Eisenbeis, Jourdan and Su's FRLC [37], and Gasperoni and Schwiegelshohn's Modified List Scheduling [15]. In particular, we compare our ILP approach with all 3 scheduling methods for architecture configurations with pipelined FUs. As the Modified List Scheduling and FRLC methods do not handle non-pipelined FUs, comparison of the ILP approach is restricted to Huff's slack scheduling method for non-pipelined architectures (**A3**, **A4**, and **A6**).

Table 8 compares the computation rate and buffer requirements of ILP schedules with those of the heuristic methods for various architectural configurations. Due to the approach followed in obtaining the ILP schedules — restricting the time to solve an ILP problem to 3 minutes and trying a schedule for the next (higher) T value (sub-optimal schedules) — the computation rate and/or the buffer requirements of ILP schedules are greater than the heuristic methods in a small fraction of the test cases. Note that the buffer requirements are compared only when the corresponding schedules had the same iteration period.

Architecture	Heuristic Method	T_{min}			Buffer Requirements		
		ILP Better	Heuristic Better	Same	ILP Better	Heuristic Better	Same
Pipelined Architectures							
A1	Huff	0	7	946	618	35	293
	FRLC	211	6	736	640	5	91
	Gasperoni	223	4	726	604	13	109
A2	Huff	0	39	882	557	64	261
	FRLC	187	13	721	613	20	88
	Gasperoni	194	14	699	551	39	109
A5	Huff	137	35	766	210	55	210
	FRLC	250	35	665	586	5	74
	Gasperoni	394	28	533	463	13	57
Non-Pipelined Architectures							
A3	Huff	0	63	713	491	34	188
A4	Huff	1	85	689	478	39	172
A6	Huff	190	9	638	478	14	146

Table 8: Comparison with Heuristic Methods

As can be seen from Table 8, Huff's slack scheduling method performed equally well (or better) in terms of iteration period for homogeneous FUs. Huff's method found faster schedules in 6% to 8% of the test cases, especially when the homogeneous FUs are non-pipelined. However, with heterogeneous FUs, ILP schedules are faster in 13% to 20% of the test cases for architectures **A5** and **A6**. Nevertheless, the high computation costs of ILP schedules pay significant dividends in terms of buffer requirements. In more than 60% of the test cases (when the corresponding schedules have the same iteration period), the buffer requirements of ILP schedules are less than those of Huff's Slack Scheduling method.

Compared to Gasperoni's modified list scheduling and Wang, et al's FRLC method, ILP produced faster schedules in 18% to 40% (or 187 to 394) of the test cases for the various architectural configurations considered. The heuristic methods score well in a small fraction (3%) of the test cases. Once again the buffer requirements of ILP schedules are better than its heuristic counterparts in 460 to 640, or roughly 80%, of the test cases.

The most attractive feature of the heuristic methods is their execution time. The execution time for any of the heuristic methods was less than 1 second for more than 90% of the loops. The mean execution time was less than 0.25 second for all the architectural configurations. Of the three heuristic methods, Huff's slack scheduling method required slightly more computation time.

Our experiments reveal that the ILP-based optimal scheduling method does produce good schedules at the expense of a longer compilation time. With the advent of more efficient ILP solvers, the compilation time is likely to decrease in future. Irrespective of the high compilation costs, our experiments suggest the possible use of the ILP approach for performance critical applications. In the following subsection we present a case for the ILP approach even though the use of such an approach in production compilers is debatable.

8.1 A Case for ILP based Exact Method

We hope that the experimental results presented in this and in the previous section will help the compiler community in the assessment of the ILP based exact method. Despite a reasonably good performance in a large majority of the test cases, the use of ILP based exact methods in production compilers remains questionable. However, in the course of our experiments, we noticed that many loop bodies occur repeatedly in different programs. We developed a tool that analyzes whether two DDGs are similar in the sense that they (1) execute the same operations — or at least execute operations with the same latency and on the same function unit, and (2) have the same set of edges and dependence distances between those operations.

We found that out of our 1008 test cases, there are only 415 loops that 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.

9 Related Work

Software pipelining has been extensively studied [2, 7, 11, 10, 21, 29, 34, 35, 14, 30, 19, 34, 36], and a variety of techniques have been suggested for finding a good schedule with bounded function units. The readers are referred to [28] for a comprehensive survey.

Lam [21] proposed a resource-constrained software pipelining method using list scheduling and hierarchical reduction of cyclic components. Our A matrix is similar to her modulo resource reservation table, a concept originally due to Rau and Glaeser [29]. Both as she put it, "represent the resource usage of the steady state by mapping the resource usage of time t to that of $t \ mod \ T$." Lam's solution of the OPT problem was also iterative. Huff's Slack Scheduling [19] is also an iterative solution to the OPT problem. His heuristics (i) give priority to scheduling nodes with minimum *slack* in the time at which they can be scheduled, and (ii) try to schedule a node at a time which minimizes the combined register pressure from node inputs and outputs. He reported extremely good results in addressing the OPT problem. Other heuristic-based scheduling methods have been proposed by Wang et al [36], and Gasperoni and Schwiegelshohn [15]. We have compared how the ILP schedules perform against these three scheduling methods in Section 8.

The FPS compiler [29], the Cydra 5 compiler, $Cydrix^{TM}$ *Fortran* [6, 7], and the HP-PA compiler [27] are production compilers based on heuristic methods implementing resource-constrained software pipelining. Rau et al. [30] have addressed the problem of register allocation for modulo scheduled loops. In their method register allocation is performed on already scheduled loops. Different code generation schema for modulo scheduled loops have been discussed in [31]. In [26], a Petri net based approach to Software pipelining loops in the presence of resource constraints has been presented. Ebcioglu et al. have proposed the technique of *enhanced software pipelining* with resource constraints [11, 10, 9]. Related work in scheduling graphs in the presence of conditionals have been reported in [38]. Ning and Gao [24] proposed an efficient method of obtaining a software-pipelined schedule using minimum buffers for a fixed initiation rate. However, they did not address function unit requirements in their formulation. In comparison to all these, our approach tries to obtain fastest computation rate and minimum buffers under the given resource constraint.

In [13] Feautrier independently gave an ILP formulation similar to our method. However his method does not include FU mapping for non-pipelined execution units. Hwang et al. have proposed an integer programming formulation for scheduling acyclic graphs in the context of high-level synthesis of systems [20].

10 Conclusions

In this paper we have proposed a method of constructing software pipelined schedules that use minimum buffers and run at the fastest iteration rate for the given resource constraints. A graph coloring method can be applied to the obtained schedule to get a schedule that uses minimum registers. Our approach is based on an integer programming formulation. The formulation is quite general in that (1) it can be used to provide a compiler option to generate faster schedules, perhaps at the expense of longer compilation time, especially for performance-critical applications; and (2) since our formulation has precisely stated optimality objectives, it can be used to ascertain the optimal solution and hence evaluate and improve existing/newly proposed heuristic methods.

We have empirically established the usefulness of our formulation by applying it to 1008 loops extracted from common scientific benchmarks on six different architecture models with varying degrees of instruction-level parallelism and pipelining. Our experimental results based on these benchmark loops indicate that our method can find an optimal schedule — optimal in terms of both computation rate and register usage — for a large majority of test cases reasonably fast. The geometric mean time to find a schedule was less than 5 seconds and the median was less than 3 seconds. Even though our ILP method takes longer, it produced schedules with smaller register requirements in more than 60% of the test cases. ILP schedules are faster (better computation rate) than their counterparts in 14% of the test cases (on the average). We believe that the results presented in this paper will be helpful in assessing the tradeoffs of ILP based exact methods for software pipelining.

Acknowledgments

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 wish to thank Qi Ning, Vincent Van Dongen, and Philip Wong for the fruitful discussions we had with them. 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. Compaction-based parallelization. Tech. Rep. TR 88-922, Dept. of Comp. Sci., Cornell U., Ithaca, N. Y., Jun. 1988. PhD thesis.

[2] 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. ACM SIGPLAN. *SIGPLAN Notices,* 23(7), Jul. 1988.

[3] A. Aiken and A. Nicolau. A realistic resource-constrained software pipelining algorithm. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, Res. Monographs in Parallel and Distrib. Computing, chapter 14, pages 274–290. Pitman Pub. and the MIT Press, London, England, and Cambridge, Mass., 1991. Selected papers from the Third Work. on Languages and Compilers for Parallel Computing, Irvine, Calif., Aug. 1–3, 1990.

[4] 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*, pages 177–189, Austin, Tex., Jan. 24–26, 1983. ACM SIGACT and SIGPLAN.

[5] E. R. Altman, R. Govindarajan, and G. R. Gao. Scheduling and mapping: Software pipelining in the presence of structural hazards. In *Proc. of the SIGPLAN '95 Conf. on Programming Language Design and Implementation*, La Jolla, Calif., Jun. 18–21, 1995. ACM SIGPLAN.

[6] J. C. Dehnert, P. Y.-T. Hsu, and J. P. Bratt. Overlapped loop support in the Cydra 5. In *Proc. of the Third Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 26–38, Boston, Mass., Apr. 3–6, 1989. ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Comp. Soc. *Computer Arch. News,* 17(2), Apr. 1989; *Operating Systems Review*, 23, Apr. 1989; *SIGPLAN Notices,* 24, May 1989.

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

[8] J. J. Dongarra and A. R. Hinds. Unrolling loops in FORTRAN. *Software — Practice and Experience*, 9(3):219–226, Mar. 1979.

[9] K. Ebcioglu and T. Nakatani. A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, Res. Monographs in Parallel and Distrib. Computing, chapter 12, pages 213–229. Pitman Pub. and the MIT Press, London, England, and Cambridge, Mass., 1990. Selected papers from the Second Work. on Languages and Compilers for Parallel Computing, Urbana, Ill., Aug. 1–3, 1989.

[10] K. Ebcioglu and A. Nicolau. A *global* resource-constrained parallelization technique. In *Conf. Proc., 1989 Intl. Conf. on Supercomputing*, pages 154–163, Crete, Greece, Jun. 5–9, 1989. ACM.

[11] K. Ebcioğlu. A compilation technique for software pipelining of loops with conditional jumps. In *Proc. of the 20th Ann. Work. on Microprogramming*, pages 69–79, Colorado Springs, Colorado, Dec. 1–4, 1987. ACM SIGMICRO and IEEE-CS TC-MICRO.

[12] P. Feautrier. Dataflow analysis of scalar and array references. *Intl. J. of Parallel Programming*, 20(1):23–53, 1991.

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

[14] G. Gao and Q. Ning. Loop storage optimization for dataflow machines. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proc. of the Fourth Intl. Work. on Languages and Compilers for Parallel Computing*, number 589 in Lec. Notes in Comp. Sci., pages 359–373, Santa Clara, Calif., Aug. 7–9, 1991. Intel Corp., Springer-Verlag. Publ. in 1992.

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

[16] L. J. Hendren and G. R. Gao. Designing programming languages for analyzability: A fresh look at pointer data structures. In *Proc. of the 1992 Intl. Conf. on Computer Languages*, pages 242–251, Oakland, Calif., Apr. 20–23, 1992. IEEE Comp. Soc. Press.

[17] 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.

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

[19] 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. ACM SIGPLAN. *SIGPLAN Notices,* 28(6), Jun. 1993.

[20] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu. A formal approach to the scheduling problem in high-level synthesis. *IEEE Trans. on Computer-Aided Design*, 10(4):464–475, Apr. 1991.

[21] 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. ACM SIGPLAN. *SIGPLAN Notices,* 23(7), Jul. 1988.

[22] 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. ACM SIGMICRO and IEEE-CS TC-MICRO. *SIG MICRO Newsletter* 23(1–2), Dec. 1992.

[23] A. Nicolau, K. Pingali, and A. Aiken. Fine-grain compilation for pipelined machines. Tech. Rep. TR 88-934, Dept. of Comp. Sci., Cornell U., Ithaca, N. Y., 1988.

[24] 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.

[25] Q. Ning. *Register Allocation for Optimal Loop Scheduling.* PhD thesis, McGill U., Montréal, Qué., 1993.

[26] M. Rajagopalan and V. H. Allan. Efficient scheduling of fine grain parallelism in loops. In *Proc. of the 26th Ann. Intl. Symp. on Microarchitecture*, pages 2–11, Austin, Tex., Dec. 1–3, 1993. IEEE-CS TC-MICRO and ACM SIGMICRO.

[27] S. Ramakrishnan. Software pipelining in PA-RISC compilers. *Hewlett-Packard J.*, pages 39–45, Jun. 1992.

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

[29] 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. ACM SIGMICRO and IEEE-CS TC-MICRO.

[30] 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. ACM SIGPLAN. *SIGPLAN Notices,* 27(7), Jul. 1992.

[31] B. R. Rau, M. S. Schlansker, and P. P. Tirumalai. Code generation schema for modulo scheduled loops. In *Proc. of the 25th Ann. Intl. Symp. on Microarchitecture*, pages 158–169, Portland, Ore., Dec. 1–4, 1992. ACM SIGMICRO and IEEE-CS TC-MICRO. *SIG MICRO Newsletter* 23(1–2), Dec. 1992.

[32] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle. The Cydra 5 departmental supercomputer. *Computer*, 22(1):12–35, Jan. 1989.

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

[34] R. F. Touzeau. A Fortran compiler for the FPS-164 scientific computer. In *Proc. of the SIGPLAN '84 Symp. on Compiler Construction*, pages 48–57, Montréal, Qué., Jun. 17–22, 1984. ACM SIGPLAN. *SIGPLAN Notices,* 19(6), Jun. 1984.

[35] 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. Springer-Verlag.

[36] J. Wang, C. Eisenbeis, M. Jourdan, and B. Su. DEcomposed Software Pipelining: A new approach to exploit instruction-level parallelism for loop programs. Res. Rep. No. 1838, Institut Nat. de Recherche en Informatique et en Automatique (INRIA), Rocquencourt, France, Jan. 1993.

[37] J. Wang and E. Eisenbeis. A new approach to software pipelining of complicated loops with branches. Res. rep. no., Institut Nat. de Recherche en Informatique et en Automatique (INRIA), Rocquencourt, France, Jan. 1993.

[38] N. J. Warter, S. A. Mahlke, W. W. Hwu, and B. R. Rau. Reverse If-Conversion. In *Proc. of the SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pages 290–299, Albuquerque, N. Mex., Jun. 23–25, 1993. ACM SIGPLAN. *SIGPLAN Notices,* 28(6), Jun. 1993.