A Framework for Resource-Constrained Rate-Optimal Software Pipelining

R. Govindarajan, **Erik R. Altman**, and **Guang R. Gao**

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

Keywords— **Instruction-Level Parallelism, Instruction Scheduling, Integer Linear Programming, Software Pipelining, Superscalar and VLIW Architectures.**

I. 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], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14]. 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.

R. Govindarajan is with the Supercomputer Education and Research Center, and Department of Computer Science and Automation, Indian Institute of Science, Bangalore, 560 012, India. E-mail:govind@serc.iisc.ernet.in. Erik Altman is with the IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, U.S.A. E-mail:erik@watson.ibm.com. Guang Gao is with the School of Computer Science, McGill University, 3480 University Street, Montreal, H3A 2A7, Canada. E-mail:gao@cs.mcgill.ca. This work was done when the first two authors were at McGill University. This research was partly funded by research grants from MICRONET – Network Centres of Excellence, Canada and NSERC, Canada.

In this paper, we are interested in addressing the following software pipelining problem:

> **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 [15], [11]. Based on the periodic scheduling framework, we express resource constraints as integer 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 [11], a unified Integer Linear Program (ILP) formulation for the *OPT-T* problem is obtained. As in [11], we use FIFO buffers to model register requirement in this paper. (The relationship between the Ning/Gao formulation and ours can be better understood by examining Fig. 2 (page 1138) in which the tradeoff between buffer and function unit optimality is depicted.)

Readers who are familiar with related work in this field will find the optimality objective in the above problem for-

mulation 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 [16] and alias analysis [17]) 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 [18]. 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.

Thus the usefulness of the techniques proposed in this paper should be viewed in the light of items (1) to (3) above.

We have implemented the solution method and tested it on 1,008 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 time with median ranging from 0.6 to 2.7 seconds for the different architectural 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 [7], Wang, Eisenbeis, Jourdan and Su's FRLC variant of DESP–Decomposed Software Pipelining [19], and Gasperoni and Schwiegelshohn's modified list scheduling approach [20]. 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 1,008 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 on loop bodies without conditional statements. Though it is possible to extend our approach to loops involving conditional statements using techniques discussed in[21], it is not clear whether the optimality objective will still hold. We defer this study to a future work. Further, in this work we focus only on architectures involving pipelined or non-pipelined function units. Function units having arbitrary structural hazards are dealt with in [22] by extending the formulation proposed for non-pipelined function units.

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 1138).

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 III. In Section IV, 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 V. Section VI deals with an iterative solution to the *OPT* problem. In Section VII, the results of scheduling 1,008 benchmark loops are reported. Our ILP schedules are compared with the schedules generated by other leading heuristic methods in Section VIII. In Section IX, we discuss other related work. Concluding remarks are presented in Section X.

II. 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.

A. 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 [13].

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

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

[2] For a small number of test cases, less than 4%, the ILP schedule was worse in terms of either initiation rate or buffer requirement. 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 VII.

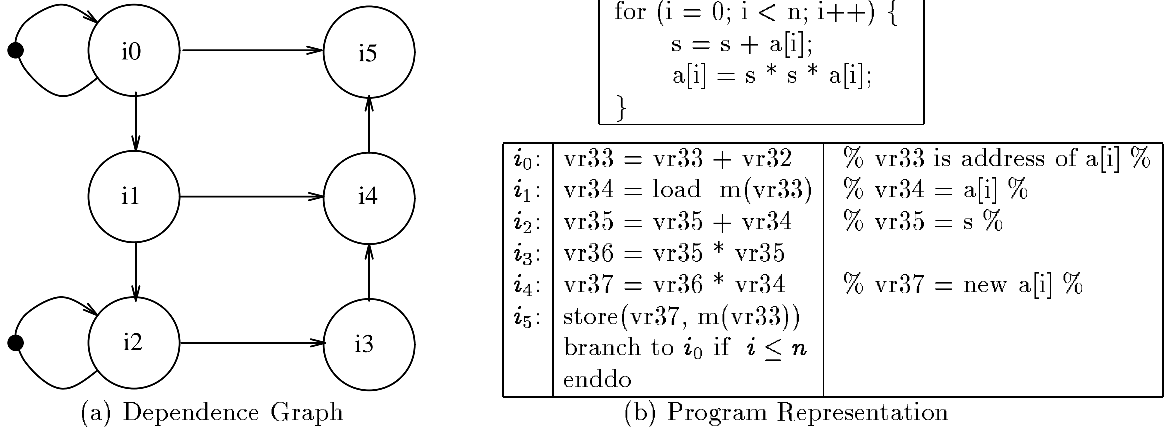


Fig. 1. An Example Loop

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 II-C.

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 [23]. 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:

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

In our example,

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

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

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

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.

B. 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 linear schedule considered in this paper, the j-th instance of an instruction i begins execution at time

TABLE I
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_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

$T \cdot j + t_i$, 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 I 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 [24], or be mapped to physical registers (with appropriate register moves) on conventional architectures as described in [8], [25]. In [25], [26], 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.

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 II which uses only 14 buffers. This schedule is also resource constrained with an iteration period 2. The values of t_i for the instructions are

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

The buffer requirements for this schedule are as shown below:

Instruction						Total Buffers
i_0	i_1	i_2	i_3	i_4	i_5	
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 formulation outlined in Section IV-C.

C. 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}

TABLE II

SCHEDULE B 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_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	

lower bound for non-pipelined FUs is:

$$T_{res}(r) = \frac{\sum_{i \in \mathcal{I}(r)} d_i}{\text{no. of FUs of type } r},$$

where $\mathcal{I}(r)$ represent the set of instructions that execute in FU type r and d_i represent the execution time of instruction i. For the motivating example of Section II-A,

$$\begin{aligned} T_{res}(\textbf{Integer FU}) &= \frac{1}{3} \\ T_{res}(\textbf{FP Unit}) &= \frac{2+2+2}{2} = 3 \\ T_{res}(\textbf{Load/Store Unit}) &= \frac{2+1}{1} = 3 \end{aligned}$$

Thus,

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

The lower bound T_{lb} is

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

A schedule, Schedule C, for non-pipelined FUs is shown in Table III. 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 V discusses such a formulation for non-pipelined FUs.

Table IV 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 $T = 4$.

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.

III. 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 II-A. 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.

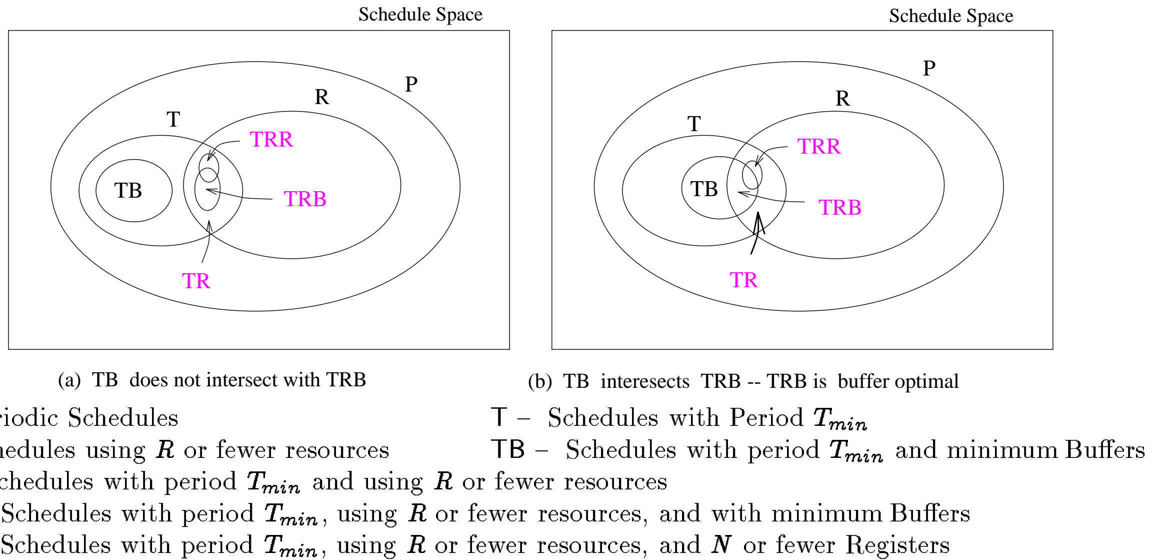
The intersection of the sets T and R refers to the set of schedules with a period T_{min} and using R or less function units. This is denoted by the region labeled TR. The schedules in TR are rate-optimal under the resource constraint R — that is there is no schedule which uses not more than R resources, 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.

TABLE III
SCHEDULE C FOR NON-PIPELINED EXECUTION UNITS

	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 IV
SCHEDULE D WITH FIXED FU ASSIGNMENT

	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



(a) TB does not intersect with TRB (b) TB interesects TRB -- TRB is buffer optimal

P – Periodic Schedules
T – Schedules with Period T_{min}
R – Schedules using R or fewer resources
TB – Schedules with period T_{min} and minimum Buffers
TR – Schedules with period T_{min} and using R or fewer resources
TRB – Schedules with period T_{min}, using R or fewer resources, and with minimum Buffers
TRR – Schedules with period T_{min}, using R or fewer resources, and N or fewer Registers

Fig. 2. Schedule Space of a Given Loop

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. 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.

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. However, for a compiler writer, the TRB region is only of indirect interest in the following sense. A compiler writer is more interested in finding a schedule with the shortest period T using R or fewer FUs and not requiring more than N registers, the available registers in the machine. Such schedules form the TRR region shown in Fig. 2. The region TRR may be contained in, may contain, may intersect, or may be disjoint with TRB[3] . One of the four relationships is possible due to the following reasons.

(1) There is no guarantee that there exists a schedule with period T and using N or fewer registers. In this case TRR is null[4]. (2) As mentioned in Section II-B, as logical buffers provide a good approximation to physical registers, one can easily see that when a TRR schedule exists, it is possible to have either all TRR schedules to be in TRB or

[3]For the sake clarity, in Fig. 2 we show only the case that TRR intersects TRB.

[4]In this case either the next higher value of T needs to be considered or some register spilling is required.

all TRB schedules to be TRR schedules. (3) Though minimum buffer requirement provides a very tight upper bound for register requirement, a minimum register schedule need not necessarily be a minimum buffer schedule. Thus TRR intersects TRB and TRR is not contained in TRB. (4) Last, though very unlikely, it is possible that none of the TRR schedules are in TRB. In this case,

$$\mathsf{TRR} \cap \mathsf{TRB} = \phi.$$

As will be seen later, it is possible to modify our formulation in Sections IV and V to find a TRR schedule using the approach followed in [26], [27]. The details of these approaches are beyond the scope of this paper. The reader is referred to [26] for further details. Due to the additional complexity introduced by the above approach in modeling register requirements directly, we restrict our attention in this paper to finding a TRB schedule.

Lastly, 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 [11]. 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.

IV. *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 IV-C, the *OPT-T* formulation for the motivating example of Fig. 1 is shown.

A. 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 IV.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 II. A periodic schedule is said to be *feasible* if it obeys all dependence constraints imposed by the DDG. The following lemma due to Reiter [23] characterizes feasible periodic schedules.

Lemma IV.1 (Reiter [23]) 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 [7]. For loops involving conditionals, we assume a hardware model that supports predicated execution as in [24]. If-conversion [28] can be performed to support this model. As well, in [13] it was shown that predicated execution simplifies code generation after modulo scheduling.

B. 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 1 or 0 depending on whether or not 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:

$$\mathcal{A} = [a_{t,i}] = \left[\begin{array}{cccccc} 1 & 0 & \mathbf{0} & \mathbf{1} & \mathbf{1} & 1 \\ 0 & 1 & \mathbf{1} & \mathbf{0} & \mathbf{0} & 0 \end{array} \right]$$

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:

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

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 [11]. 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.

C. 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 II.

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:

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

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

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

[ILP Formulation for Pipelined FUs]

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

subject to

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

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

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

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

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

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

Fig. 3. ILP formulation for Pipelined FUs

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

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

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:

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

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.

V. *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 II-C, 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 V-B.

A. 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. 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.

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

and

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

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 of $\mathcal{U}$ 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:

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

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:

$$\begin{aligned} u_{0,i_1} &= a_{0,i_1} + a_{2,i_1} & u_{1,i_1} &= a_{1,i_1} + a_{0,i_1} \\ u_{2,i_1} &= a_{2,i_1} + a_{1,i_1} & u_{0,i_2} &= a_{0,i_2} + a_{2,i_2} \\ u_{1,i_2} &= a_{1,i_2} + a_{0,i_2} & u_{2,i_2} &= a_{2,i_2} + a_{1,i_2} \\ u_{0,i_3} &= a_{0,i_3} + a_{2,i_3} & u_{1,i_3} &= a_{1,i_3} + a_{0,i_3} \\ u_{2,i_3} &= a_{2,i_3} + a_{1,i_3} & u_{0,i_4} &= a_{0,i_4} + a_{2,i_4} \\ u_{1,i_4} &= a_{1,i_4} + a_{0,i_4} & u_{2,i_4} &= a_{2,i_4} + a_{1,i_4} \end{aligned}$$

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,

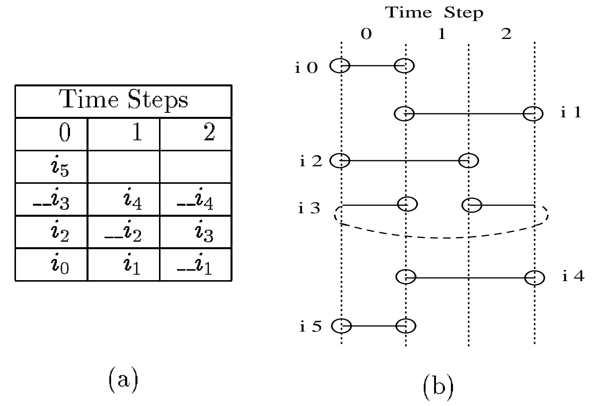
$$\sum_{i \in \mathcal{I}(r)} u_{t,i} \leq F_r \quad \text{for all} \;\; t \in [0, T-1] \text{ and for all } r \tag{24}$$

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 II-C, the complete formulation must include the mapping part (fixed FU assignment) as well. Otherwise the schedules produced by the formulation may require the switching of instructions between FUs during the course of execution[5]. In the following subsection we show how the mapping problem can also be formulated under the same framework.

B. Fixed FU Assignment

Consider Schedule C shown in Table III. 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 i_3 wraps around from time 2 to 0. This is a problem. At time 2, i_3 begins executing on the function unit that was used by i_2 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 i_3 is still executing on the FU needed by i_2. 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.



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)

Fig. 4. A Repetitive Pattern and its Resource Usage

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 [29]. 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 [30]. 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 $c_i - c_j$.

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

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

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

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

In [22] 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

[5] Alternatively, it may be possible to unroll the loop a number of times and use different FU assignment for the same instruction in the unrolled iterations. However, the extent of unrolling required may be large and may not be known a priori.

units as colors. Hence we replace Equation 24 with Equations 25 – 27 and

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

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

$$c_i \le F_r \quad \forall i \in [0, N-1] \text{ and } i \in \mathcal{I}(r) \qquad (28)$$

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

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

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

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

$$c_i - c_j \ \ge \ \frac{u_{t,i} + u_{t,j} - 1}{2} - N \cdot w_{i,j} \qquad (33)$$

$$c_j - c_i \ \ge \ \frac{u_{t,i} + u_{t,j} - 1}{2} - N \cdot (1 - w_{i,j})$$
$$\forall i, j \in \mathcal{I}(r) \text{ and } r \qquad (34)$$

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

$$1 \le c_k \le N \quad \text{for all } k \in [0, N-1] \qquad (36)$$

$$b_i \ge 0, t_i \ge 0, k_i \ge 0, \text{ and } a_{t,i} \ge 0 \text{ are integers}$$
$$\forall i \in [0, N-1], \text{ and } t \in [0, T-1] \qquad (37)$$

Fig. 5. ILP formulation for Non-Pipelined FUs

VI. 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} [8], [7]. 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 [11], 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 [11] does not include resource constraints, the obtained schedule may or may not satisfy resource constraints.

VII. Performance of ILP Schedules

In this section we present the performance results of the ILP scheduler. Section VIII 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 [7]. 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. Once 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. 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 architec-

TABLE V
LATENCIES OF INSTRUCTIONS.

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

tures.

In a large majority of cases, the ILP approach found an optimal schedule close to T_{min} as shown in Table VI. 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 VI indicates how far the schedule found was from a *possible* optimal schedule.

TABLE VI
SCHEDULE QUALITY IN TERMS OF ITERATION PERIOD

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

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 minute 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 schedule obtained from the Ning-Gao formulation [11]. 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 $B_{TR} \geq B_{TRB} \geq B_{TB}$. To compare the quality of schedules, we take the 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 optimistic lower bound.

Table VII 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**.

TABLE VII
SCHEDULE QUALITY IN TERMS OF BUFFER REQUIREMENT

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

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 workstation. The geometric mean, arithmetic mean, and median of the execution time for the 6 architectural configurations are shown in Table VIII. A histogram of the execution time for various architectural configurations is shown in Figure 6. From Table VIII we observe that the geometric mean of execution time is less than is less than 2 seconds for architectures with homogeneous

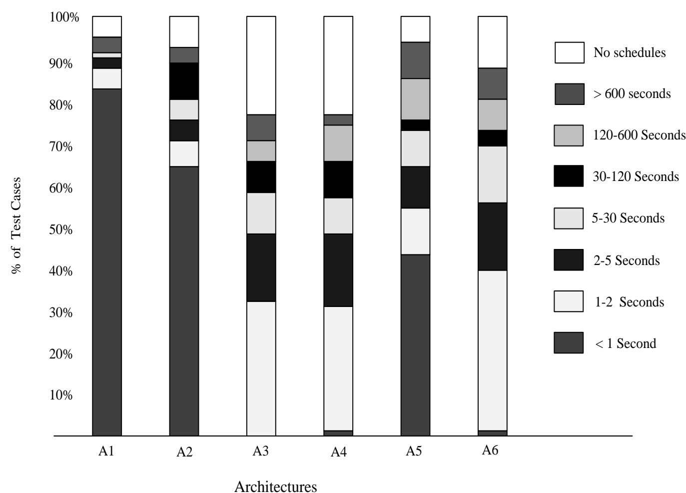


Fig. 6. Histogram of Execution Time

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.

TABLE VIII
Average Execution Time to obtain ILP Schedules

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

We conclude this section by noting that even though our ILP based scheduling method was successful in a large majority of test cases, it still could not find a schedule for 15% to 20% of the test cases in the given time limit and the number of tries. For these cases, there are a number of alternatives: (1) allow the ILP more than 3 minutes, (2) change the order in which the ILP solver attempts to satisfy the constraints, (3) move to some other exact approach such as enumeration [26], (4) fall back to some heuristic. We have made no systematic investigation of (1) and (2), although have found that each is successful for some loops. Enumeration achieves about the same number of loops scheduled as the ILP approach described here, although the loops successfully scheduled by the two approaches are not identical [26]. The ILP approach can be used as the basis for some heuristics. For example, heuristic limits on the scheduling times of each node could be added as constraints to the ILP.

VIII. 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 [7], Wang, Eisenbeis, Jourdan and Su's FRLC [31], and Gasperoni and Schwiegelshohn's Modified List Scheduling [20]. 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 IX compares the computation rate and buffer requirements of ILP schedules with those of the heuristic methods for various architectural configurations. In particular, columns 3 and 4 tabulate the number of loops in which the ILP schedules did better and the percentage improvement in T_{min} achieved. Similarly columns 8 and 9 represent the improvements in buffer requirements. 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. Columns 5 and 6 represent, respectively, the number of test loops and the percentage improvement in T_{min} achieved by the heuristic methods. Columns 10 and 12 in Table IX are for buffer improvements. Note that the buffer requirements are compared only when the corresponding schedules had the same iteration period.

As can be seen from Table IX, 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 3% to 8% of the test cases, especially when the FUs are homogeneous and non-pipelined. However, with heterogeneous FUs, ILP schedules are faster in 13% to 20% of the test cases for architectures **A5** and **A6**. In these cases, the ILP schedules are faster on the average by 13% to 15% as shown in column 4 of Table IX. Further, the high computation costs of ILP schedules pay significant dividends in terms of buffer requirements for all architecture configurations. In more than 45% 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. The geometric mean of the improvement (in buffer requirements) achieved by the ILP schedules range from 15% to 22%.

Compared to Gasperoni's modified list scheduling and Wang, et al's FRLC method, ILP produced faster sched-

TABLE IX
COMPARISON WITH HEURISTIC METHODS

		T_{min}					Buffer Requirements				
		ILP Better		Heuristic Better		Same	ILP Better		Heuristic Better		Same
Architecture	Heuristic Method	# of Loops	% Impr.	# of Loops	% Impr.	# of Loops	# of Loops	% Impr.	# of Loops	% Impr.	# of Loops
Pipelined Architectures											
A1	Huff	0	0	7	38	946	618	15	35	16	293
	FRLC	211	47	6	19	736	640	24	5	6	91
	Gasperoni	223	48	4	27	726	604	17	13	7	101
A2	Huff	0	0	39	28	882	557	17	64	13	261
	FRLC	187	36	13	19	721	613	25	20	8	88
	Gasperoni	194	35	14	18	699	551	19	39	11	109
A5	Huff	137	13	35	6	766	210	18	55	11	210
	FRLC	250	33	35	4	665	586	29	5	5	74
	Gasperoni	394	26	28	4	533	463	20	13	5	57
Non-Pipelined Architectures											
A3	Huff	0	0	63	14	713	491	20	34	12	188
A4	Huff	1	75	85	22	689	478	21	39	13	172
A6	Huff	190	15	9	10	638	478	22	14	12	146

ules in 18% to 40% (or 187 to 394) of the test cases for the various architectural configurations considered. The improvement in T_{min} achieved by the ILP schedules are significant, 26% to 48%. This means that the schedules generated by the ILP method can run 50% faster than those generated by the FRLC method or the modified list scheduling method. These heuristic methods score well in a small fraction (up to 3%) of the test cases. Once again the buffer requirements of ILP schedules are better (by 17% to 29%) than FRLC or modified list scheduling in 460 to 640 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 though 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.

A. Remarks

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! The repetition of loop bodies, on the one hand, implies that our benchmark suite consists only of 415 unique test cases (rather than 1008); on the other hand, it suggests the number of distinct loops appearing in scientific programs is limited, and 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.

The complexity of the tool to analyze whether two DDGs are similar is $O(E^4)$ in the worst case, but $O(E)$ in the average case, where E is the number of edges in the DDG, and in most cases $E \approx N$, the number of nodes in the DDG. 53 seconds were required on a Sun/Sparc20 to find the 415 unique loops out of the 1008, i.e. about 53 milliseconds per loop For practical use, the tool requires that a database of DDGs and their schedules stored in an encoded form.

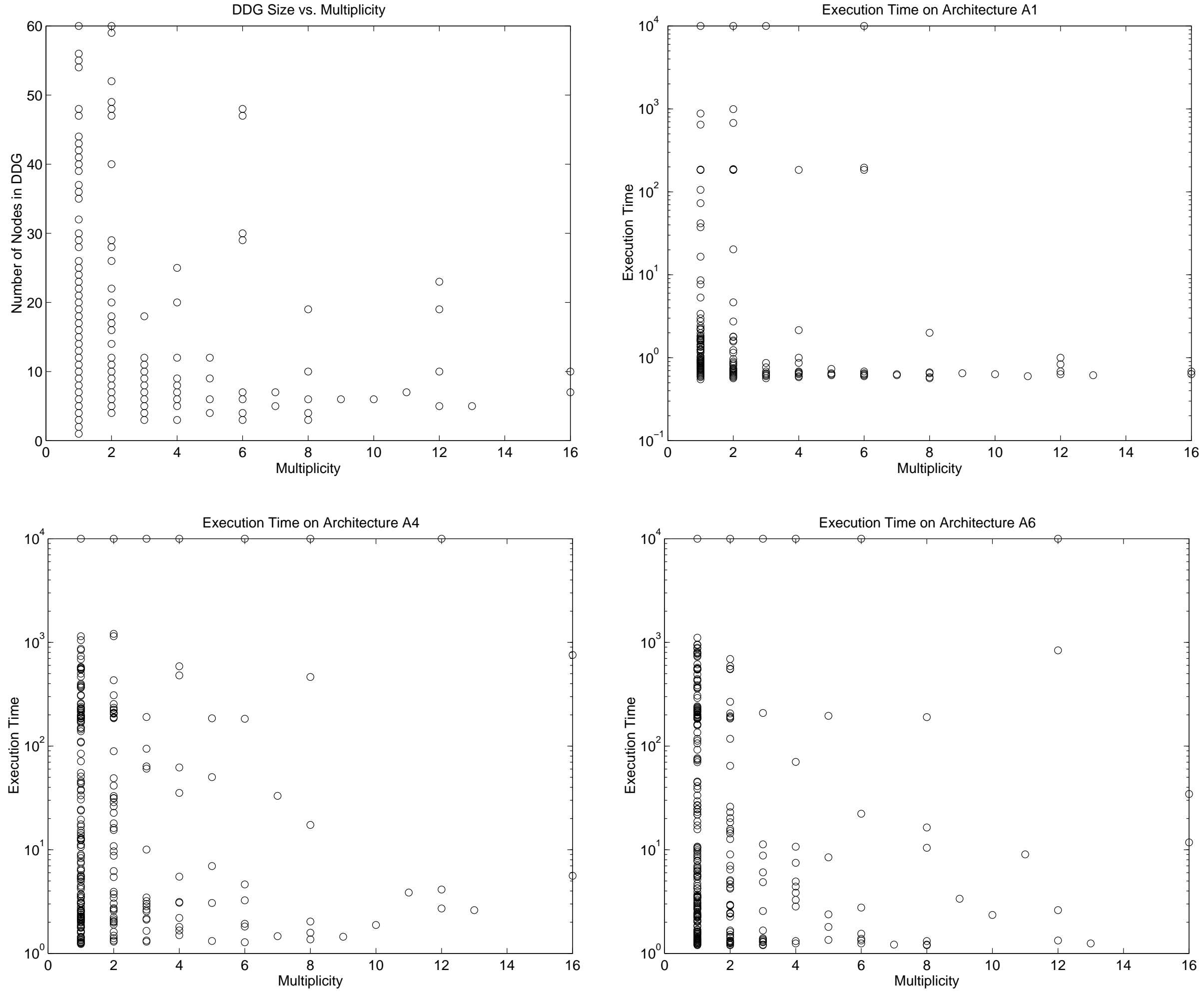


Fig. 7. Analysis of DDGs in Benchmark Suite

The number of DDGs (in the database) that are compared with a given loop can be drastically reduced by a simple comparison of the number of nodes and the number of arcs of the DDGs.

One last question remains on the usefulness of such a database of DDGs and their precompiled schedules: How many of these (precompiled) schedules required a longer compilation time? This question is relevant because if the database of DDGs only contain loops for which the schedule can anyway be found in a shorter compilation time, it perhaps will take lesser time to determine the schedule than to search the database. We investigate this by plotting the compilation time of the 415 unique loops against *multiplicity* — how often does this DDG repeat in the benchmark suite. We also plot the size of the DDGs versus multiplicity in Fig. 7.

As can be seen from Figure 7, though the repetition of DDGs is more common when the size of the DDG is small, large DDGs do repeat, perhaps with a low degree of multiplicity (2 to 6). The plots on compilation time of DDGs (for various architectural configurations) against multiplicity also indicate similar results; i.e. though a majority of the database is likely to contain DDGs that take shorter compilation time, there does exist DDGs which require longer compilation time and repeat in the benchmark suite, perhaps with a low degree of multiplicity. This is especially true for architectural configurations **A3** to **A6**.

Our initial results only show that DDGs that require longer compilation time do repeat, though with a lower degree of multiplicity. However, it does not study the tradeoff involved in the cost of storing database of loops with their precompiled schedules and the advantage in obtaining optimal schedules quickly. Such a tradeoff determines the usefulness of the database approach. Further study is required to derive stronger and conclusive results.

IX. Related Work

Software pipelining has been extensively studied [2], [4], [5], [6], [8], [12], [14], [15], [32], [13], [7], [14], [19], and a variety of techniques have been suggested for finding a good schedule with bounded function units. Readers are referred to [33] for a comprehensive survey.

Lam [8] proposed a resource-constrained software pipelining method using list scheduling and hierarchical re-

duction of cyclic components. Our $\mathcal{A}$ matrix is similar to her modulo resource reservation table, a concept originally due to Rau and Glaeser [12]. Both as she put it, "represent the resource usage of the steady state by mapping the resource usage of time t to that of $t \bmod T$." Lam's solution of the *OPT* problem was also iterative. Huff's Slack Scheduling [7] 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 [19], and Gasperoni and Schwiegelshohn [20]. We have compared how the ILP schedules perform against these three scheduling methods in Section VIII.

The FPS compiler [12], the Cydra 5 compiler, *Cydrix*^{TM} *Fortran* [34], [4], and the HP-PA compiler [35] are production compilers based on heuristic methods implementing resource-constrained software pipelining. Rau et al. [13] 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 [36]. In [37], 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 [5], [6], [38]. Related work in scheduling graphs in the presence of conditionals have been reported in [21]. Ning and Gao [11] 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 constraints.

In [39] Feautrier independently gave an ILP formulation similar to our method. However his method does not include FU mapping for non-pipelined execution units. Eichenberger, Davidson and Abraham [27] have proposed a method to minimize the maximum number of live values at any time step for a given repetitive pattern by formulating the problem as a linear programming problem. However, their approach start with a repetitive pattern that already satisfies resource constraints. It is possible to incorporate their approach in our formulation and model register directly, rather than through logical buffers. Such an approach was independently developed and incorporated in our formulation by Altman [26]. Hwang et al. have proposed an integer programming formulation for scheduling acyclic graphs in the context of high-level synthesis of systems [40].

X. 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, Vincent Van Dongen, and Philip Wong 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, "Compaction-based parallelization," Technical Report TR 88-922, Department of Computer Science, Cornell University, Ithaca, New York, June 1988. PhD thesis.

[2] A. Aiken and A. Nicolau, "Optimal loop parallelization," in *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, (Atlanta, Georgia), pp. 308–317, June 22–24, 1988. *SIGPLAN Notices,* 23(7), July 1988.

[3] A. Aiken and A. Nicolau, "A realistic resource-constrained software pipelining algorithm," in *Advances in Languages and Compilers for Parallel Processing* (A. Nicolau, D. Gelernter, T. Gross, and D. Padua, eds.), Research Monographs in Parallel and Distributed Computing, ch. 14, pp. 274–290, London, England, and Cambridge, Massachusetts: Pitman Publishing and the MIT Press, 1991. Selected papers from the Third Workshop on Languages and Compilers for Parallel Computing, Irvine, California, August 1–3, 1990.

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

[5] K. Ebcioğlu, "A compilation technique for software pipelining of loops with conditional jumps," in *Proceedings of the 20th Annual Workshop on Microprogramming*, (Colorado Springs, Colorado), pp. 69–79, December 1–4, 1987.

[6] K. Ebcioglu and A. Nicolau, "A *global* resource-constrained parallelization technique," in *Conference Proceedings, 1989 International Conference on Supercomputing*, (Crete, Greece), pp. 154–163, June 5–9, 1989.

[7] R. A. Huff, "Lifetime-sensitive modulo scheduling," in *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, (Albuquerque, New Mexico), pp. 258–267, June 23–25, 1993. *SIGPLAN Notices,* 28(6), June 1993.

[8] M. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, (Atlanta, Georgia), pp. 318–328, June 22–24, 1988. *SIGPLAN Notices,* 23(7), July 1988.

[9] S.-M. Moon and K. Ebcioğlu, "An efficient resource-constrained global scheduling technique for superscalar and VLIW processors," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, (Portland, Oregon), pp. 55–71, December 1–4, 1992. *SIG MICRO Newsletter* 23(1–2), December 1992.

[10] A. Nicolau, K. Pingali, and A. Aiken, "Fine-grain compilation for pipelined machines," Technical Report TR 88-934, Department of Computer Science, Cornell University, Ithaca, New York, 1988.

[11] Q. Ning and G. R. Gao, "A novel framework of register allocation for software pipelining," in *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (Charleston, South Carolina), pp. 29–42, January 10–13, 1993.

[12] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," in *Proceedings of the 14th Annual Microprogramming Workshop*, (Chatham, Massachusetts), pp. 183–198, October 12–15, 1981.

[13] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker, "Register allocation for software pipelined loops," in *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, (San Francisco, California), pp. 283–299, June 17–19, 1992. *SIGPLAN Notices,* 27(7), July 1992.

[14] R. F. Touzeau, "A Fortran compiler for the FPS-164 scientific computer," in *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, (Montréal, Québec), pp. 48–57, June 17–22, 1984. *SIGPLAN Notices,* 19(6), June 1984.

[15] V. Van Dongen, G. R. Gao, and Q. Ning, "A polynomial time method for optimal software pipelining," in *Proceedings of the Conference on Vector and Parallel Processing, CONPAR-92*, no. 634 in Lecture Notes in Computer Science, (Lyon, France), pp. 613–624, Springer-Verlag, September 1–4, 1992.

[16] P. Feautrier, "Dataflow analysis of scalar and array references," *International Journal of Parallel Programming*, vol. 20, no. 1, pp. 23–53, 1991.

[17] L. J. Hendren and G. R. Gao, "Designing programming languages for analyzability: A fresh look at pointer data structures," in *Proceedings of the 1992 International Conference on Computer Languages*, (Oakland, California), pp. 242–251, IEEE Computer Society Press, April 20–23, 1992.

[18] J. J. Dongarra and A. R. Hinds, "Unrolling loops in FORTRAN," *Software — Practice and Experience*, vol. 9, pp. 219–226, March 1979.

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

[20] F. Gasperoni and U. Schwiegelshohn, "Efficient algorithms for cyclic scheduling," Research Report RC 17068, IBM T. J. Watson Research Center, Yorktown Heights, New York, 1991.

[21] N. J. Warter, S. A. Mahlke, W. mei W. Hwu, and B. R. Rau, "Reverse if-conversion," in *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, (Albuquerque, New Mexico), pp. 290–299, June 23–25, 1993. *SIGPLAN Notices,* 28(6), June 1993.

[22] E. R. Altman, R. Govindarajan, and G. R. Gao, "Scheduling and mapping: Software pipelining in the presence of structural hazards," in *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, (La Jolla, California), pp. 139–150, June 18–21, 1995. *SIGPLAN Notices,* 30(6), June 1995.

[23] R. Reiter, "Scheduling parallel computations," *Journal of the ACM*, vol. 15, pp. 590–599, October 1968.

[24] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer – design philosophies, decisions, and trade-offs," *Computer*, vol. 22, pp. 12–35, January 1989.

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

[26] E. R. Altman, *Optimal Software Pipelining with Function Unit and Register Constraints.* PhD thesis, McGill University, Montréal, Québec, October 1995.

[27] A. E. Eichenberger, E. S. Davidson, and S. G. Abraham, "Minimum register requirements for a modulo schedule," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, (San Jose, California), pp. 75–84, November 30–December2, 1994.

[28] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, (Austin, Texas), pp. 177–189, January 24–26, 1983.

[29] L. J. Hendren, G. R. Gao, E. R. Altman, and C. Mukerji, "A register allocation framework based on hierarchical cyclic interval graphs," in *Proceedings of the 4th International Conference on Compiler Construction, CC '92* (U. Kastens and P. Pfahler, eds.), no. 641 in Lecture Notes in Computer Science, (Paderborn, Germany), pp. 176–191, Springer-Verlag, October 5–7, 1992.

[30] T. C. Hu, *Integer Programming and Network Flows*, p. 270. Addison-Wesley Publishing Company, 1969.

[31] J. Wang and E. Eisenbeis, "A new approach to software pipelining of complicated loops with branches," research report no., Institut National de Recherche en Informatique et en Automatique (INRIA), Rocquencourt, France, January 1993.

[32] G. Gao and Q. Ning, "Loop storage optimization for dataflow machines," in *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing* (U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, eds.), no. 589 in Lecture Notes in Computer Science, (Santa Clara, California), pp. 359–373, Intel Corp., Springer-Verlag, August 7–9, 1991. Published in 1992.

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

[34] J. C. Dehnert, P. Y.-T. Hsu, and J. P. Bratt, "Overlapped loop support in the Cydra 5," in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, (Boston, Massachusetts), pp. 26–38, April 3–6, 1989. *Computer Architecture News,* 17(2), April 1989; *Operating Systems Review*, 23, April 1989; *SIGPLAN Notices,* 24, May 1989.

[35] S. Ramakrishnan, "Software pipelining in PA-RISC compilers," *Hewlett-Packard Journal*, pp. 39–45, June 1992.

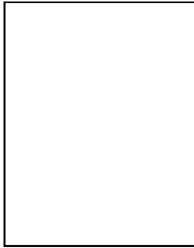
[36] B. R. Rau, M. S. Schlansker, and P. P. Tirumalai, "Code generation schema for modulo scheduled loops," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, (Portland, Oregon), pp. 158–169, December 1–4, 1992. *SIG MICRO Newsletter* 23(1–2), December 1992.

[37] M. Rajagopalan and V. H. Allan, "Efficient scheduling of fine grain parallelism in loops," in *Proceedings of the 26th Annual International Symposium on Microarchitecture*, (Austin, Texas), pp. 2–11, December 1–3, 1993.

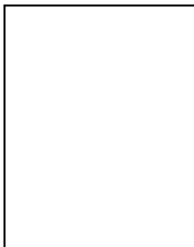
[38] K. Ebcioglu and T. Nakatani, "A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture," in *Languages and Compilers for Parallel Computing* (D. Gelernter, A. Nicolau, and D. Padua, eds.), Research Monographs in Parallel and Distributed Computing, ch. 12, pp. 213–229, London, England, and Cambridge, Massachusetts: Pitman Publishing and the MIT Press, 1990. Selected papers from the Second Workshop on Languages and Compilers for Parallel Computing, Urbana, Illinois, August 1–3, 1989.

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

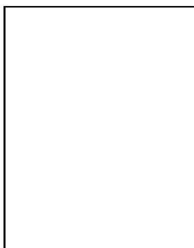
[40] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu, "A formal approach to the scheduling problem in high-level synthesis," *IEEE Transactions on Computer-Aided Design*, vol. 10, pp. 464–475, April 1991.



R. Govindarajan received his Ph.D. Degree in Computer Science from the Indian Institute of Science, Bangalore, India, in 1989. He received his Bachelor's degree in Engineering from the same institute earlier in 1984, and his Bachelor's degree in Science in 1981 from the University of Madras, Madras, India. He has worked as an Assistant Professor in the Department of Electrical Engineering, McGill University, Montreal, Canada from 1992-94, and in the Department of Computer Science, Memorial University of Newfoundland, St. John's, Canada, from 1994-95. Currently, he is working as an Assistant Professor in the Supercomputer Education and Research Center and in the Department of Computer Science and Automation, Indian Institute of Science, Bangalore, India. His research interests are in the areas of instruction scheduling, dataflow and multithreaded architectures, programming model and scheduling of DSP applications, and performance evaluation. R. Govindarajan is a member of the IEEE Computer Society.



Erik R. Altman received his M.S. and Ph.D. degrees in Electrical Engineering from McGill University in 1991 and 1995, respectively. He received his Bachelor's degree in Electrical Engineering and Computer Science from the Massachusetts Institute of Technology in 1985. From 1985 to 1989 he worked for several small firms: TEK Microsystems, Machine Vision International, and Bauer Associates. Currently he is a Research Staff Member at the IBM T.J. Watson Research Center in Yorktown Heights, New York. His research interests are in the areas of VLIW architectures and compilers, simulation, instruction scheduling, and caches.



Guang R. Gao received his SM and Ph.D degree from the Massachusetts Institutes of Technology in 1982 and 1986 respectively. He has been a faculty member at McGill University since 1987, where he is the founder and a leader of the Advanced Compilers, Architectures and Parallel Systems (ACAPS) laboratory. His research interests have centered around high-performance architectures and optimizing compilers. He has authored and co-authored more than 100 technical papers on the topics of computer architecture, parallelizing compilers, code optimization, parallel processing, data-flow and multithreaded program execution models and architectures. He has edited several research monographs and has chaired and co-chaired a number international meetings, workshops, and conferences in his research area. He has been a consultant for several computer industry and government research institutions. Currently, he is a co-editor of the Journal on Programming Languages. He has been a member of the program committees or organizing committee of many international conferences in his field.

Dr. Gao is a Senior Member of IEEE, and a member of ACM, SIGARCH and IEEE Computer Society.