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 (a 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 makes 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 heuristic-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.

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 \textit{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 \textit{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 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, Jourend 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 \( L \) 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 \( i_3 \) (Integer FU)

\(^1\) 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.
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 FUs, 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 *initialization rate of successive iterations*. In the following discussion, we often use the reciprocal of the initiation rate, the *initialization interval* $T$. Let us first establish a lower bound for $T$ — i.e., the shortest initialization 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:

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

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:

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

In our example,

$$T_{res}(\text{Integer FU}) = \frac{1}{3}$$

$$T_{res}(\text{FP Unit}) = \frac{1}{2}$$

$$T_{res}(\text{Load/Store Unit}) = \frac{1}{2} = 2$$

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

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

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

$$T_{init} = \max \{ [T_{dep}]_1, [T_{res}]_1 \} = \max \{ 2, 2 \} = 2$$

That is, any schedule of loop $\mathcal{L}$ that obeys the resource constraint will have a period greater than or equal to $T_{init} = 2$. The smallest iteration period $T_{min} \geq T_{init}$, 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 $\frac{1}{T_{init}}$ 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
$T_{j+t_i}$, 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 $T_j + t_i$, 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 ($T_{\text{min}} = 2$).

Next let us compute the register requirement for this schedule. In Schedule $A$, the instruction $i_0$ fires six times before the first $i_0$ 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 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 $i_1$. 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.

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

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

The buffer requirements for this schedule are as shown below:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Total Buffers</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$i_0$</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>$i_1$</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>$i_2$</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$i_3$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$i_4$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$i_5$</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

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 $T_{\text{res}}$
lower bound for non-pipelined FUs is:

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

where $I(r)$ 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,

$$T_{res}(\text{Integer FU}) = \frac{1}{3}$$

$$T_{res}(\text{FP Unit}) = \frac{2 + 2 + 2}{2} = 3$$

$$T_{res}(\text{Load/Store Unit}) = \frac{2 + 1}{1} = 3$$

Thus,

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

The lower bound $T_b$ is

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

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 $i_2$ at time step 9, and the second FP unit to $i_4$ at time step 10, then we have the first FP unit free at time step 11 and the second FP unit free at time step 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 we are interested in is only 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 $L$, 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 II**

<table>
<thead>
<tr>
<th>Iteration = 0</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i_0$</td>
<td>$i_1$</td>
<td>$i_2$</td>
<td>$i_3$</td>
<td>$i_4$</td>
<td>$i_5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iteration = 1</td>
<td>$i_0$</td>
<td>$i_1$</td>
<td>$i_2$</td>
<td>$i_3$</td>
<td>$i_4$</td>
<td>$i_5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iteration = 2</td>
<td>$i_0$</td>
<td>$i_1$</td>
<td>$i_2$</td>
<td></td>
<td>$i_4$</td>
<td>$i_5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iteration = 3</td>
<td>$i_0$</td>
<td>$i_1$</td>
<td>$i_2$</td>
<td></td>
<td>$i_3$</td>
<td></td>
<td>$i_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iteration = 4</td>
<td>$i_0$</td>
<td>$i_1$</td>
<td>$i_2$</td>
<td></td>
<td>$i_3$</td>
<td></td>
<td>$i_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Iteration = 5</td>
<td>$i_0$</td>
<td>$i_1$</td>
<td>$i_2$</td>
<td></td>
<td>$i_3$</td>
<td></td>
<td>$i_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
TABLE III

<table>
<thead>
<tr>
<th>Schedule C for Non-Pipelined Execution Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>Iteration = 0</td>
</tr>
<tr>
<td>Iteration = 1</td>
</tr>
<tr>
<td>Iteration = 2</td>
</tr>
<tr>
<td>Iteration = 3</td>
</tr>
</tbody>
</table>

TABLE IV

<table>
<thead>
<tr>
<th>Schedule D with Fixed FU Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>Iteration = 0</td>
</tr>
<tr>
<td>Iteration = 1</td>
</tr>
<tr>
<td>Iteration = 2</td>
</tr>
<tr>
<td>Iteration = 3</td>
</tr>
</tbody>
</table>

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

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 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. 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.
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

\[ T \]

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,

\[ \text{TRR} \cap \text{TRB} = \emptyset. \]

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 \( T_{\text{min}} \) that use the minimum number of registers. That is, for the initiation interval \( T_{\text{min}} \), 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 \( L \), 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:

\[ t_j - t_i \geq d_i - T \cdot m_{ij} \]

(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 \( T_{\text{min}} \) is always an integer. If not, the given DDG can be unrolled a suitable number of times, such that the resulting (unrolled) DDG has an 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 \( A \) matrix. The matrix \( A \) 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 \( A[t,i] \) 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 \( A \) matrix is:

\[
A = \begin{bmatrix}
1 & 0 & 0 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 & 0
\end{bmatrix}
\]

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:

\[
\sum_{t \in I[r]} a_{t,i} \leq F_r \quad \text{for all } t \text{ and for all } r
\]

(2)

where \( I[r] \) 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:
Next we concentrate on some constraints on the $A$ 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 $A$ matrix to be 1. This can also be expressed as a linear constraint as:

$$\sum_{t=0}^{T-1} a_{t,i} = 1 \text{ for all } i \in [0, N - 1]$$  \hspace{1cm} (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 $A$ matrix to the $t_i$ variables. For this purpose we can rewrite each $t_i$ as:

$$t_i = k_i \times T + \alpha_i \text{ such that } \alpha_i \in [0, T - 1]$$  \hspace{1cm} (4)

In other words, $k_i$ and $\alpha_i$ are defined as:

$$k_i = t_i \text{ div } T \quad \text{and} \quad \alpha_i = t_i \text{ mod } T$$

where % represents the modulo operation. For Schedule $B$,

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

$$\alpha_0 = 0; \quad \alpha_1 = 1; \quad \alpha_2 = 1; \quad \alpha_3 = 0; \quad \alpha_4 = 0; \quad \alpha_5 = 0;$$

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

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

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

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

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

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

Or, in general,

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

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 execution 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 $(t_j + T \cdot m_{ij} - t_i)/T$ 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

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

Rewriting Equation (8), we get

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

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:

$$t_1 - t_0 \geq 1 \quad t_2 - t_0 \geq 1 \quad t_2 - t_1 \geq 2$$

$$t_4 - t_1 \geq 2 \quad t_3 - t_2 \geq 2 \quad t_4 - t_3 \geq 2$$

$$t_5 - t_4 \geq 2$$

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

$$a_{0,0} + a_{1,0} = 1 \quad a_{0,1} + a_{1,1} = 1 \quad a_{0,2} + a_{1,2} = 1$$

$$a_{0,3} + a_{1,3} = 1 \quad a_{0,4} + a_{1,4} = 1 \quad a_{0,5} + a_{1,5} = 1$$

(17)
[ILP Formulation for Pipelined FUs]

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

subject to

\[
\sum_{t \in \mathcal{T}(\tau)} a_{t,i} \leq F_{\tau}, \quad \text{for all } t, \in [0, T-1] \ \forall \ \tau
\]

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

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

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

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

\[
b_i \geq 0, t_i \geq 0, k_i \geq 0, a_{t,i} \geq 0 \text{ are integers}
\]

\[
\forall i \in [0, N-1], \forall t \in [0, T-1]
\]

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 \( \mathbf{A} \) matrix), but also how long each executes. For example, instruction \( i_2 \) 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 \( \text{FP Multiply} \) 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 \( i_5 \) requires an FU at time steps 2 and 0 in the repetitive pattern. Thus we need to define a usage matrix \( \mathbf{U} \) from the \( \mathbf{A} \) matrix to represent the usage of non-pipelined FUs.

First we illustrate the \( \mathbf{A} \) matrix and the usage matrix \( \mathbf{U} \) for Schedule \( C \).

\[
\mathbf{A} = \begin{bmatrix}
1 & 0 & 1 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 \\
\end{bmatrix}
\]

and

\[
\mathbf{U} = \begin{bmatrix}
1 & 0 & 1 & 1 & 0 & 1 \\
0 & 1 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 1 & 0 \\
\end{bmatrix}
\]

Notice that the \( \text{FP} \) instructions and the \( \text{Load} \) 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 \( \mathbf{U} \) gives the FU requirement for type \( r \).

How do we obtain the \( \mathbf{U} \) matrix from \( \mathbf{A} \)? 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 \( \mathbf{U}[t, i] \) as:

\[
\mathbf{U}[t, i] = a_{t,i} \sum_{t=0}^{(d_i-1)} a_{[t(\%T)+i],s}, \quad \forall t \in [0, T-1], \forall i \in V
\]

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 \( a_{t,i}, k_i, t_i, \text{ and } b_i \) are non-negative integers. Solving this integer program formulation yields Schedule \( B \).
Notice that if the execution time $d_t = 1$ 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

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

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:

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

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

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

The requirement for type $r$ FUs at time step $t$ is

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

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

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

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. 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 FUs 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 [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 (function unit to which they are mapped) 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 $u_{i,j}$ integer, 0-1 variables, with one such variable for each pair of nodes using the same type of function unit. Roughly speaking these $u_{i,j}$ variables represent the sign of $c_i - c_j$.

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

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

$$1 \leq c_k \leq N \quad \forall k \in [0, N - 1] \quad (27)$$

$N$, 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
units as colors. Hence we replace Equation 24 with Equations 25 - 27 and

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

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

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

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

\[ T \cdot K + A^{\text{transpose}} \times [0, 1, \ldots, T - 1]^{\text{transpose}} = T \quad (30) \]

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

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

\[ c_i - c_j \geq \frac{u_{t,i} + u_{t,j} - 1}{2} - N \cdot w_{i,j} \quad (33) \]

\[ c_j - c_i \geq \frac{u_{t,i} + u_{t,j} - 1}{2} - N \cdot (1 - w_{i,j}) \quad \forall i, j \in I(r) \text{ and } r \quad (34) \]

\[ t_j - t_i \geq d_k - T \cdot m_{ij} \quad \text{for all } (i, j) \in E \quad (35) \]

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

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

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 $T_0$ until we find a schedule satisfying the function unit constraint. In other words, $T_{\text{min}}$ is the smallest value greater than or equal to $T_0$ for which a schedule obeying the resource constraint exists. We want to solve the OPT-T formulation with iteration period $T_{\text{min}}$.

It has been observed that in most cases, $T_{\text{min}}$ is at or near $T_0$ [8], [7]. Thus using an iterative search, starting at $T_0$ we can quickly converge to $T_{\text{min}}$.

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 $[T_{\text{min}}, T_{\text{min}} + 5]$ were tried if necessary. Once a schedule was found before $T_{\text{min}} + 5$, 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:

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>6 pipelined homogeneous FUs</td>
</tr>
<tr>
<td>A2</td>
<td>4 pipelined homogeneous FUs</td>
</tr>
<tr>
<td>A3</td>
<td>6 non-pipelined homogeneous FUs</td>
</tr>
<tr>
<td>A4</td>
<td>4 non-pipelined homogeneous FUs</td>
</tr>
<tr>
<td>A5</td>
<td>Pipelined heterogeneous FUs (2 Integer FUs and one of Load/Store, FP Add, Multiply and Divide units.)</td>
</tr>
<tr>
<td>A6</td>
<td>Same as A5, but function units are non-pipelined</td>
</tr>
</tbody>
</table>

The 1008 loops were scheduled for each of these architec-
In a large majority of cases, the ILP approach found an optimal schedule close to $T_{\text{min}}$ 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 $T_{\text{min}}$. That is, in these cases, the obtained schedule is a possible optimal schedule. We say a possible $T_{\text{min}}$ and possible optimal schedule here since there is no evidence — CPLEX’s 3 minute time limit expired without indicating whether or not a schedule exists for a lower value of $T_{\text{min}}$. Table VI indicates how far the schedule found was from a possible optimal schedule.

### Table VI: Schedule Quality in Terms of Iteration Period

<table>
<thead>
<tr>
<th>Initiation Interval</th>
<th>Number of Loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T = T_{\text{min}}$</td>
<td>946 882 714 699 854 792</td>
</tr>
<tr>
<td>$T = T_{\text{min}} + 1$</td>
<td>1 4 37 39 60 9</td>
</tr>
<tr>
<td>$T = T_{\text{min}} + 2$</td>
<td>0 24 9 10 18 9</td>
</tr>
<tr>
<td>$T = T_{\text{min}} + 3$</td>
<td>0 4 7 5 13 9</td>
</tr>
<tr>
<td>$T = T_{\text{min}} + 4$</td>
<td>6 1 1 17 9 9</td>
</tr>
<tr>
<td>$T = T_{\text{min}} + 5$</td>
<td>0 6 8 5 1 9</td>
</tr>
<tr>
<td>No Schedule found</td>
<td>55 87 232 233 53 166</td>
</tr>
</tbody>
</table>

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 $B_{\text{min}}$ as $B_{TB}$ if a TR schedule is found and $B_{TR}$ otherwise. Thus, when a TR schedule is not found, $B_{\text{min}}$ 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 homogeneous 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

<table>
<thead>
<tr>
<th>Initiation Interval</th>
<th>Number of Loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B = B_{\text{min}}$</td>
<td>916 846 710 696 804 766</td>
</tr>
<tr>
<td>$B = B_{\text{min}} + 1$</td>
<td>0 1 32 28 33 17</td>
</tr>
<tr>
<td>$B = B_{\text{min}} + 2$</td>
<td>4 4 21 25 22 8</td>
</tr>
<tr>
<td>$B = B_{\text{min}} + 3$</td>
<td>1 5 7 8 20 33</td>
</tr>
<tr>
<td>$B = B_{\text{min}} + 4$</td>
<td>2 21 5 15 20 6</td>
</tr>
<tr>
<td>$B = B_{\text{min}} + 5$</td>
<td>28 27 0 2 50 6</td>
</tr>
</tbody>
</table>

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/SpaC20 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.
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.

![Histogram of Execution Time](image)

**Fig. 6. Histogram of Execution Time**

### 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 VIII**

**Average Execution Time to Obtain ILP Schedules**

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Geo. Mean</th>
<th>Median</th>
<th>Arith. Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>0.90</td>
<td>0.65</td>
<td>13.0</td>
</tr>
<tr>
<td>A2</td>
<td>1.80</td>
<td>0.65</td>
<td>35.9</td>
</tr>
<tr>
<td>A3</td>
<td>6.60</td>
<td>2.65</td>
<td>63.1</td>
</tr>
<tr>
<td>A4</td>
<td>7.40</td>
<td>2.73</td>
<td>74.5</td>
</tr>
<tr>
<td>A5</td>
<td>4.00</td>
<td>1.70</td>
<td>73.6</td>
</tr>
<tr>
<td>A6</td>
<td>4.70</td>
<td>2.35</td>
<td>55.7</td>
</tr>
</tbody>
</table>

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.
<table>
<thead>
<tr>
<th>Architecture</th>
<th>Heuristic Method</th>
<th>$T_{\text{min}}$</th>
<th>Buffer Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$T_{\text{min}}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Heuristic Better</td>
<td>Same</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I LP Better</td>
<td>Heuristic Better</td>
</tr>
<tr>
<td></td>
<td></td>
<td>% of Loops</td>
<td>% of Loops</td>
</tr>
<tr>
<td></td>
<td></td>
<td>% of Loops Impr.</td>
<td>% of Loops Impr.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pipelined Architectures</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>A2</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>A5</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Non-Pipelined Architectures</th>
</tr>
</thead>
<tbody>
<tr>
<td>A3</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>A4</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>A6</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

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.
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 trade-off 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 $A$ 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 $t \mod T$." 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, Cydriz™ 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. Ebcioğlu 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

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 1996, 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 Institute 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.