McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures and Parallel Systems

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

Erik R. Altman
R. Govindarajan†
Guang R. Gao

ACAPS Technical Memo 91
January 2, 1995

A version of this paper appears in the Proceedings of the 1995 SIGPLAN PLDI Conference.

{erik,gao}@acaps.cs.mcgill.ca,govind@cs.mun.ca

†Department of Computer Science
Memorial University of Newfoundland
St. Johns, Newfoundland A1C 5S7, CANADA

Abstract

Recently, software pipelining methods based on an ILP (Integer Linear Programming) framework have been successfully applied to derive rate-optimal schedules under resource constraints. Such ILP based methods provide a way to establish bounds based on optimal solutions, which can be used by compiler developers to improve the performance of existing and proposed scheduling methods.

However, like much other previous work on software pipelining, ILP based work has focused on resource constraints of simple function units: e.g. "clean pipelines" — pipelines without structural hazards. The problem for architectures beyond such clean pipelines remains open. One challenge is how to represent such resource constraints for unclean pipelines (e.g. non-pipelined or pipelined but having structural hazards) and their assignment (mapping) simultaneously under a unified ILP framework.

In this paper, we propose a method to construct *rate-optimal* software pipelined schedules for pipelined architectures with structural hazards. A distinct feature of this work is that it gives a unified treatment to two challenging and interlated aspects of software pipelining — assignment (mapping) and scheduling for available function units — under a unified ILP framework.

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

Contents

List of Figures

List of Tables

1 Introduction

Software pipelining overlaps operations from different loop iterations in an attempt to fully exploit instruction level parallelism. To be successful on real machines, the function unit constraints of those machines must be taken into account. A variety of software pipelining algorithms [22, 11, 16, 1, 2, 7, 27, 17, 29, 13, 28, 18, 5, 20] have been proposed which operate under resource constraints. An excellent survey of these algorithms can be found in [21].

More recently, integer linear programming (ILP) based approaches for finding an optimal periodic schedule on machines with simple resource usage have been proposed [9, 8, 6]. While such a method may not be acceptable for use in a practical compilation process, we believe that it is very useful to compiler designers to establish an optimal bound so as to compare and improve their heuristic methods of software pipelining. [1] Furthermore, such an approach can be provided as a compiler option for those users who may want to explore the best possible schedules for their (performance-critical) code on a given machine even if this may mean a substantially longer compile time.

Despite significant progress in resource-constrained software pipelining, dealing with function units with complex usage patterns (i.e. function units which are either unpipelined or pipelined but contain (arbitrary) structural hazards — we call them *unclean pipelines* for lack of better term) is a challenging task. To our knowledge, obtaining an optimal schedule under these conditions remains open. Intuitively, a key difficulty here is that one must solve the scheduling problem and resource assignment (mapping) problem simultaneously. For example, once an instance of an instruction is initiated in a pipeline, it should "flow" through all the subsequent stages in the *same* pipeline and according to the reservation table of the pipeline. This is because, it is unrealistic to switch function units in the middle of pipelined execution. This results in a mapping problem: that is the scheduler must simultaneously order the instructions and map them to function units. The challenge here is how such "mapping" constraints can be integrated into a unified problem formulation for software pipelining such that an optimal solution can be defined.

In this paper, we address this mapping problem from a general viewpoint: we ensure that the mapping of an instruction i to function unit $FU(i)$ is *fixed* in that instruction i always executes on $FU(i)$ throughout the loop. Such fixed FU assignment is interesting for future processors with multiple register sets, each with its own set of function units, and for architectures like VLIW, where a specific operation (or part) of the instruction word is linked to a function unit.

The main contributions of this paper are:

- We extend our integer programming method to architectures with more complex resource constraints, namely unclean pipelines. We show that the mapping constraint

[1] For example, in [8], we demonstrated that even for some simple kernel loops, significant improvement is possible over the heuristic approachs in [7, 28] and to a lesser extent over the approach proposed in [13].

described above can be identified as an instance of the *circular-arc graph coloring* problem [10] and can be represented by integer linear constraints and smoothly integrated in our overall ILP framework.

- We have implemented this in our (extended) framework and have performed tests on 1066 loops found in typical benchmark programs such as the `SPEC92`, the `NAS kernels`, `linpack`, and the `livermore loops`. In these experiments, we obtained an optimal schedule for greater than 75% of the loops. We ran the experiments on a Sparc 10/30, which required a median of 2 seconds and a geometric mean of only 5 seconds per loop to compute these schedules. To our knowledge, this is the first report of experimental results on optimal ILP-based schedules for pipelines with structural hazards on a large number of real loops. We hope that these results will help the compiler community in the assessment of ILP based methods.

The scope of periodic schedules and the ILP based solution framework can be extended to a large variety of loops with conditionals. For example, conditional statements can be handled by techniques discussed in [30], while code size reduction can be achieved when using special hardware support such as rotating register files and predicated execution [23]. Furthermore, register constraints can also be integrated in the framework as demonstrated in [8].

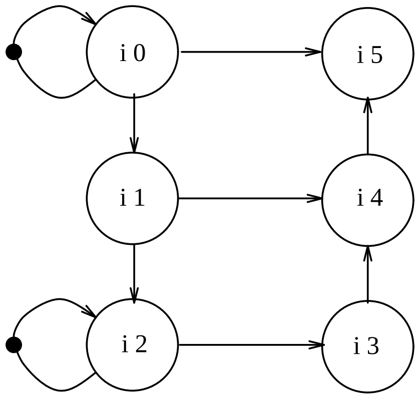
The rest of this paper is organized as follows. In the following section we motivate our approach to the scheduling problem in the presence of unclean pipelines with an example. Section 3 deals with the formulation of the problem for non-pipelined execution units. In Section 3 we also develop the (coloring) formulation for the mapping problem for the (special) case of unpipelined function units: i.e. mapping an instruction to a fixed (unpipelined) function unit on which the instruction is executed during *every* iteration of the loop. In Section 4 we extend our formulation to pipelined units with arbitrary structural hazards. Experimental results involving 1066 loops are reported in Section 5. We compare our work with other related work in Section 6. Concluding remarks are presented in Section 7.

2 Background and Motivation

In this section, we highlight the issues encountered in formulating the software pipelining problem in the presence of unclean pipelines. In section 2.1, we address the issues when the execution units are non-pipelined — a special, but very important case of unclean pipelines. In section 2.2, we examine execution units which are pipelined but have structural hazards.

We use the simple example loop in Fig. 1 as a running example throughout. Both C language and instruction level representations are given in Fig. 1(b) while the data dependence graph (DDG) is depicted in Figure 1(a).

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



(a) Dependence Graph

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

(b) Program Representation

Figure 1: An Example Loop

In this paper we focus on periodic *linear schedules* for software pipelining, under which an instruction i in iteration j is initiated at time $j * T + t_i$, where T is the period (or *initiation interval* — often abbreviated as *II* in the literature) of the schedule and t_i is a constant. The reciprocal of T is the *initiation rate* of schedule. For more background information on linear scheduling, readers are referred to the survey paper by Rau and Fisher [21].

2.1 Non-pipelined Execution Units

For purpose of discussion in this section, we consider an architecture with 3 **Integer** units, 2 **floating-point** (**FP**) units and 1 **load/store** unit. We assume that the **Integer** units perform all integer operations and likewise the **FP** units perform all floating point operations. Further let the execution time of an **Integer add** ($i0$) be 1 cycle, while that of an **FP multiply** ($i2$, $i3$, and $i4$) be 2 cycles. Assume also that **load** ($i1$) takes 2 cycles and **store** ($i5$) takes 1 cycle.[2] Lastly we assume that all function units are non-pipelined, i.e. the next operation cannot be initiated on an FU until the previous execution is completed.

It is well known that the maximum initiation rate of successive iterations of a loop is constrained by both *loop-carried* dependences and available resources [22, 16, 13, 21, 3]. Namely,

- The loop-carried dependences put a lower bound, T_{dep}, on T. The value of T_{dep} is determined by the critical (dependence) cycle(s) in the loop [25]. Specifically

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

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

along the critical cycle(s). For the DDG in Fig.1(a), T_{dep} is 2 corresponding to the self loop at instruction $i2$.

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

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

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

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

The shortest initiation interval T_{lb} is the maximum of T_{res} and T_{dep}. However there may or may not exist a schedule with a period T_{lb} satisfying the given resource constraints. T_{min} is the minimum period for which a schedule exists. [3]

In order to ensure that, in the course of execution, an instruction does not switch from one FU to another, which is impractical, we enforce a stronger constraint on the schedule: that an instruction s always be executed on the same FU during the course of the loop execution. This fixed instruction (task) to FU mapping may be essential in a multiprocessor system (especially in a Non-Uniform Memory Access (NUMA) machine), as well as in future microprocessor architectures, in which multiple register sets are provided, with each dedicated to one (or a group of) execution units. In addition, variable FU assignment does not accurately depict VLIW architectures in which function units are generally linked to a specific operation (part) of the instruction word.

To illustrate the significance of this fixed assignment constraint, examine Schedule A in Table 1. This schedule is for the DDG in Figure 1(a) and is obtained from the linear schedule form $T \cdot i + t_s$, with $T = 3$, $t_{i0} = 0$, $t_{i1} = 1$, $t_{i2} = 3$, $t_{i3} = 5$, $t_{i4} = 7$, and $t_{i5} = 9$. We use the notation $\_\_i1$ to indicate that the execution of instruction $i1$ continues from the previous time step. Since the FUs are assumed to be non-pipelined, instructions continuing from previous time step, continue to engage the FU. Schedule A has a prolog (from $[0,6]$), a repetitive pattern ($[7,9]$), and an epilog. Note that the resource constraints are met in the repetitive pattern — no more than 1 **Integer**, 2 **FP**, and 1 **Load/Store** unit are needed at any one time. (Times 7 and 8: 2 **FP** and 1 **Load/Store**; Time 9: 1 **Integer**, 2 **FP**, 1 **Load/Store**.) In other words, Schedule A satisfies both the dependence constraints and the resource constraints of our architecture.

Unfortunately, Schedule A does not satisfy the fixed FU assignment criteria. To illustrate this, consider the repetitive pattern starting at time step 7. If we assign the first **FP** unit to

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

	Time Steps													
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Iteration = 0	i0	i1	__i1	i2	__i2	i3	__i3	i4	__i4	i5				
Iteration = 1				i0	i1	__i1	i2	__i2	i3	__i3	i4	__i4	i5	
Iteration = 2							i0	i1	__i1	i2	__i2	i3	__i3	i4
Iteration = 3										i0	i1	__i1	i2	__i3
Iteration = 4													i0	i1

Table 1: Schedule A for the Motivating Example

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

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

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

One important observation we have made is to identify the FU assignment problem as a circular-arc coloring problem [10]. We introduce this coloring as a linear constraint in an integer linear programming formulation to be presented in the next section.

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

2.2 Pipelined Execution Units with Structural Hazards

In this subsection we illustrate how the software pipelining problem is altered by the presence of structural hazards in function unit pipelines. A structural hazard is caused by one or more stages of the pipeline being used at multiple times during the execution of an instruction. For example, a **shifter** may be used to normalize inputs at the start of a floating point divide, and

	Time Steps														
	0	1	2	3	4	5	6	7	8	*9*	*10*	*11*	*12*	13	14
Iteration = 0	$i0$	$i1$	__$i1$	$i2$	__$i2$	$i3$	__$i3$	$i4$	__$i4$			i5			
Iteration = 1					$i0$	$i1$	__$i1$	$i2$	__$i2$	i3	__i3	i4	__i4		
Iteration = 2									$i0$	i1	__i1	i2	__i2	$i3$	__$i3$
Iteration = 3													i0	$i1$	__$i1$

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

be reused again at the end of the divide to normalize the output. The times at which each stage is used are generally represented by a reservation table [15].

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

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

Reservation Table for **FP** Unit

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

Reservation Table for **Load/Store** Unit

In order to understand the resource requirements of a schedule for pipelined function units with structural hazards, we may model each stage of a pipeline as a separate resource. We depict the basic software pipelined schedule (Schedule C) in Table 3. Unlike Tables 1 and 2, Table 3 shows only when an operation is initiated. The resource requirement is shown separately in a set of *resource-usage tables*. The resource usage tables for **FP multiply** during the repetitive pattern of the loop are depicted in Table 4.

In order to verify that Schedule C satisfies the resource constraints, we need to check the resource requirement for each type of FU. We concentrate only on the repetitive pattern since the resource requirement in the prolog and epilog is less than for any periodic pattern. Further, we concentrate only on the **FP** unit. The requirements for **Integer** and **Load/Store** units can be estimated easily from Table 3 as there is no overlapping execution in those FUs. The resource usage table for each stage of the **FP** unit can be derived from Table 3 and the corresponding rows in the **FP** reservation table. For example, in stage 2 of Table 4, the $i2$ entry denotes the fact that $i2$ from iteration 2 is using stage 2 at time step 13 (or equivalently time step 0 in the repetitive pattern)

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

Table 3: Schedule C for Pipelines with Structural Hazards

	Stage 1				Stage 2				Stage 3			
	Time Steps				Time Steps				Time Steps			
	13	14	15	16	13	14	15	16	13	14	15	16
Iter. = 0												
Iter. = 1		$i4$					$i4$		$i3$		$i4$	$i4$
Iter. = 2			$i3$		$i2$			$i3$	$i2$	$i2$		$i3$
Iter. = 3				$i2$								
Iter. = 4												

Table 4: Resource Usage Tables for **FP** Unit

Note that there is no overlap in the usage of stages 1 and 2 of the **FP** pipeline. However, in stage 3 $i2$ and $i3$ overlap, and hence cannot share the same **FP** unit. However, since there is no overlap between instructions $i2$ and $i4$, a single **FP** unit can be assigned to both. This leads to our second Problem, that of formulating within a single framework, instruction to FU mapping and resource constrained software pipelining for function units with structural hazards.

> **Problem 2:** Given a DDG and an architecture with structural hazards, construct a schedule that has the shortest initiation interval, using only the available resources, and mapping each instruction i to the same function unit $FU(i)$ in every iteration.

As can be easily seen, the fixed FU assignment problem (in the presense of structural hazards) is also related to the coloring problem. In fact, this can be derived by an extension of the formulation of the non-pipelined case discussed earlier. Combining the classical reservation table (e.g. modulo reservation table) based method with graph coloring provides the key to an integrated problem formulation. Problems 1 and 2 are treated formally in the next two Sections.

3 Scheduling for Non-Pipelined Execution Units

In this section, we develop a formulation for the software pipelining problem for architectures involving non-pipelined execution units. The formulation is based on our earlier work on integer program formulation for clean pipelines [9]. The details of the earlier formulation are presented in Appendix A. We briefly review and borrow some essential notation from [9] in Section 3.1 that is relevant to the formulation for non-pipelined architectures. In Section 3.2, we develop a graph coloring formulation which guarantees a schedule with fixed FU assignment for each instruction. The result of this integration is a unified formulation for scheduling and mapping.

3.1 The Basic ILP Formulation

In this paper, we consider linear periodic schedules of the form $j \cdot T + t_i$, where T is the period and t_i is the time at which node i in the 0-th iteration is initiated.

A highlight of our ILP formulation is the representation of each t_i variable as

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

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

$$\mathcal{K} = \begin{bmatrix} k_0 \\ k_1 \\ \vdots \\ k_{N-1} \end{bmatrix}, \quad \mathcal{T} = \begin{bmatrix} t_0 \\ t_1 \\ \vdots \\ t_{N-1} \end{bmatrix} \text{ and } \mathcal{A} = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,(N-1)} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,(N-1)} \\ \vdots & \vdots & & \vdots \\ a_{(T-1),0} & a_{(T-1),1} & \cdots & a_{(T-1),(N-1)} \end{bmatrix} \tag{2}$$

The $\mathcal{A}$ matrix succinctly captures the repetitive pattern: it is a 1-0 matrix where $a_{t,i}$ is 1 if and only if instruction i is scheduled at time step t. Otherwise $a_{t,i} = 0$. The $\mathcal{A}$ matrix resembles the *modulo-reservation table* [22, 16] and specifies the time step (in the repetitive pattern) at which an instruction is *initiated.*

However, to determine resource requirements with non-pipelined function units, we need to know not just when each instruction is *initiated*, but also how long each executes. Having written the t_i variables in the form shown in Equation (1), we can do just that. If an instruction is initiated at time step t in the repetitive pattern, it will execute until time step $(t + d_i - 1) \ \% \ T$, where % is the `mod` function as in the C language. Consequently, instruction i will require this resource until time step $(t + d_i - 1) \ \% \ T$. 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. We can define this more formally by defining the matrix U:

$$u_{t,i} = \sum_{l=0}^{(d_i-1)} a_{((t-l)\%T),i}, \quad \forall t \in [0, T-1], \text{ and } \forall i \in V \tag{3}$$

Notice that if the execution time $d_i = 1$ cycle, then $u_{t,i} = a_{t,i}$. [4] Since, $\sum_{t=0}^{T-1} a_{t,i} = 1$ as in Equation (12 in the ILP Formulation in Appendix A) $u_{t,i} \in \{0, 1\}$.

Now to compute the resource requirement at time step t, we need to add the $u_{t,i}$'s for all i. However since different instructions may need different types of function units, we compute the resource requirement for each type of function unit separately. To accomplish this, we consider $\mathcal{I}(r)$: the set of all instructions that require a function unit of type r.

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

$$R_r \geq \sum_{i \in \mathcal{I}(r)} u_{t,i}, \quad \forall t \in [0, T-1] \tag{4}$$

The ILP formulation attempts to minimize a weighted sum of the R_r for the various function unit types. The weight (or cost) associated with a function unit type r is C_r. The objective function, then, is

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

The formulation minimizes the objective function subject to the dependency constraints (Equation (15) in Appendix A) and resource constraints (Equation (4), for all $r \in [0, h-1]$). In addition, Equations (12) and (16) in Appendix A are also necessary to guarantee that each instruction is scheduled at least once in the repetitive pattern. The complete integer programming formulation is shown in Fig. **??** in Appendix B.

We will show the ILP formulation for our motivating example in Appendix D. In the following section we develop the formulation necessary for guaranteeing fixed FU assignment.

3.2 Coloring Formulation for Fixed FU Assignment

Unfortunately, the ILP formulation presented in the previous section sometimes underestimates the number of function units required to execute a loop under fixed FU assignment. For instance, for the schedule shown in Table 1, it appears that only two **FP** units are needed at times, 7, 8, 9, (or 0, 1, and 2, if we count 7 as 0 in the repetitive pattern). (Note that $i2$'s function unit requirement wraps around from time 2 to time 0 in the repetitive pattern.) This 2 **FP** unit "thickness" of the resource-usage matrix U is used by Equation 17 in the ILP formulation as the total number of function units required. However, for this schedule, we claim that no fixed function unit assignment exists that uses only 2 **FP** units for nodes $i2, i3$, and $i4$! Readers are encouraged to verify this claim as an exercise (Hint: Even though $i2$ wraps around, we still need to assign the same **FP** unit to it under the fixed FU assignment constraint!).

[4] 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, $u_{t,i} = a_{t,i}$.

Acute readers will recognize that this can be stated as a graph coloring problem — the *circular arc coloring* problem [10], and indeed it can be. The challenge here is how to express the coloring problem in a format compatible with the rest of the ILP formulation.

One natural way of representing this coloring is

$$|c_i - c_j| \geq \frac{u_{t,i} + u_{t,j} - 1}{2} \tag{5}$$

since $u_{t,i}$ and $u_{t,j}$ represent the use of a function unit by instructions i and j respectively at time step t. Here the colors c_i and c_j, which represent the colors for nodes i and j, are positive integers, and they must be different if nodes i and j overlap during loop execution. If both $u_{t,i}$ and $u_{t,j}$ are 1 at some time, t, the nodes overlap. In such cases the value of the right hand side at time t is $\frac{1}{2}$, thus forcing c_i and c_j to be different. Conversely if only one of $u_{t,i}$ and $u_{t,j}$ is 1 at time t or if both are 0, then the right hand side is 0 or $-\frac{1}{2}$, thus allowing c_i and c_j to be the same (if i and j do not overlap at any other time).

Alas, absolute value is not a linear operation, so this constraint does not fit with the rest of our ILP framework. To surmount this problem, we adopt an approach outlined by Hu [12]. We introduce a set of $w_{i,j}$ integer, 0-1 variables, with one such variable for each pair of nodes using the same type of function unit. Roughly speaking these $w_{i,j}$ variables represent the sign of $c_i - c_j$, as discussed in Appendix C. Using them, Equation (5) can be represented with the following equations:

$$c_i - c_j \geq \frac{u_{t,i} + u_{t,j} - 1}{2} - N \cdot w_{i,j} \tag{6}$$

$$c_j - c_i \geq \frac{u_{t,i} + u_{t,j} - 1}{2} - N \cdot (1 - w_{i,j}) \tag{7}$$

$$1 \leq c_k \leq N \quad \forall k \in [0, N-1] \tag{8}$$

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

Theorem 3.1 *Equations 6 – 8 require that two nodes i and j be assigned different colors (mapped to different function units) if and only if they overlap.*

For a proof, please see Appendix C.

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

$$\min_{i \in \mathcal{I}(r)} \left(\max_{i \in \mathcal{I}(r)} c_i \right)$$

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

$$\min Q$$

subject to

$$Q \geq c_i \quad \forall i \in \mathcal{I}(r)$$

However, for our ILP formulation we do not minimize colors directly. Instead we require that there be at least as many function units of each type as colors. Hence we replace Equation 17 with:

$$R_r \geq c_i \quad \forall i \in \mathcal{I}(r), \ \forall r \in [0, h-1] \tag{9}$$

The complete ILP formulation with fixed FU assignment (i.e. for **Problem 1**) is obtained by adding Equations (9), (6), and (7) to the formulation shown in Section 3.1 (and eliminating Equation 17 as redundant).

In Appendix D, we show the ILP formulation for the example loop considered in Section 2.1.

4 Scheduling for Pipelines with Structural Hazards

Having developed a unified formulation for software pipelining (scheduling) and fixed FU assignment (mapping) problems for non-pipelined execution units (**Problem 1**), we now extend the formulation to pipelined function units with structural hazards (**Problem 2**).

In order to represent the usage of the various stages of a pipeline by each instruction, we use reservation tables [15]. A reservation table has s stages and d columns, where d is the execution time of an instruction in the pipeline. For simplicity sake we will assume that an FU of a particular type r has a single reservation table describing its use of various stages. This assumption is only to simplify discussion and is not a limitation of the formulation, as will be discussed in Section 4.1. The number of time steps over which a particular stage s in a pipeline is used (not necessarily continuously) during the execution of an instruction is denoted by d_s. Clearly, for fixed FU assignment, $T \geq d_s$. Further we will assume that no stage in a pipeline is used by an instruction at two time steps that are congruent modulo T, (Two time steps are congruent modulo T if they are separated by a multiple of T.) Previous authors made a similar assumption known as the modulo scheduling constraint [21, 11, 5]. In Section 4.1 we discuss how to relax this modulo scheduling constraint and the associated cost.

For each function unit whose execution time $d < T$, we extend the reservation table of the function unit to T columns by adding $(T - d)$ zero (column) vectors. Since $d < T$, each entry in the *modified reservation table* is at most 1. For the case $d > T$, we fold the reservation table for the function unit to T columns such that the t-th column in the original reservation table is 'merged' with the $(t \ \% \ T)$-th column in the modified reservation table. The 'merge' operation results in adding the respective entries in two columns. Use of the modulo scheduling constraint guarantees that each entry in the modified reservation table is at most 1.

The s-th row in the modified reservation table of function unit type r specifies the usage of stage s. Let us denote this row by RT_r^s, a vector of length T. An entry $RT_r^s[t]$ is 1 if stage s is required t time steps after the initiation of an instruction ($\texttt{mod}\ T$). Using this vector, and the $\mathcal{A}$ matrix we can define for each instruction i which uses an FU of type r and for each of the s_r stages in r, the resource usage matrix U as follows:

$$U^s[t,i] = \sum_{l=0}^{(T-1)} a_{((t-l)\%T),i} \cdot RT_r^s[l], \qquad \forall t \in [0, T-1], \forall s \in [0,(s_r-1)], \text{ and } i \in \mathcal{I}(r), \qquad (10)$$

Once again, a similarity can be observed between Equations (3 and 10). The introduction of RT_r^s facilitates selectively choosing appropriate $a_{t,i}$'s. Using the resource usage matrix defined by Equation 10, the resource requirement of stage s in function unit type r at time step t in the repetitive pattern is

$$R_r^s(t) = \sum_{i\in\mathcal{I}(r)} U^s[t,i] = \sum_{i\in\mathcal{I}(r)} \sum_{l=0}^{(T-1)} a_{((t-l)\%T),i} \cdot RT_r^s[l]$$

The number of FUs of type r required for the schedule must be greater than or equal to $RT_r^s(t)$ for $t \in [0, T-1]$. That is, if R_r is the number of FU's of type r required for this schedule, then

$$R_r \geq R_r^s \quad \forall s \in [0,(s_r-1)]$$

which in turn can be written as

$$R_r \geq \sum_{i\in\mathcal{I}(r)} \sum_{l=0}^{(T-1)} a_{((t-l)\%T),i} \cdot RT_r^s[l], \quad \forall t \in [0, T-1], \text{ and } s \in [0,(s_r-1)]$$

As before, we will minimize a weighted sum of R_r for each function unit of type r. Lastly, we adopt the coloring formulation developed in Section 3.2 to ensure fixed FU assignment. The complete formulation for the software pipelining problem with structural hazards is shown in Appendix B.

4.1 Remarks

In this section we briefly mention our approach when the reservation table does not obey the modulo scheduling constraint. In these cases since $d_s \leq T$, it possible to introduce sufficient delays in the pipelines [15] such that the modulo constraint is satisfied in the modified reservation table. However, this may increase the execution time of the instructions in the DDG and hence may affect T_{dep} which in turn affects T. The following approach is used to choose the right T value.

We start with the smallest T value such that $T \geq T_{lb}$. By introducing delays, we make sure that the modulo scheduling constraint is satisfied by all function unit types r. Once appropriate delays have been introduced in the pipe, we recompute T_{dep}. Note that T_{res} will not be affected by the introduction of delays. If the new T_{dep} value is less than or equal to T_{lb}, we can proceed to the schedule the loop for the current value of T. Otherwise, we increment T by 1 and recompute the delays and repeat the above process. We follow this approach (that of incrementing T by 1 rather than substituting T by the new T_{dep} value) in order not to miss any lower value of T for which a schedule exists under the given assumptions. Our experiments have shown that though delays were introduced in the pipeline in 118 out of the 1066 cases, in none of the cases did the introduction of delays increase T.

Lastly, we will briefly mention how the above formulation can handle FUs with multiple reservation tables as in practice a Floating Point unit may support many operations, such as **FP-add**, **FP-multiply**, and **FP-divide**, each having a different resource usage pattern. In this case, the resource usage table RT is associated with the instruction rather than the function unit. That is, for each instruction i, we use the RT_i^s which is determined from the appropriate reservation table.

5 Experimental Results

In this section we provide results of experiments using our framework. We use the PowerPC-604 [14] architecture as a basis for these experiments. Except for the **divide** unit (which has no pipelining), the function units on the 604 are fully pipelined. To test our approach, we have changed this and assumed that the pipelines have structural hazards represented by the reservation tables in Appendix E. [5]

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

In order to deal with the fact that our ILP approach 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 $[T_{lb}, T_{lb} + 5]$ were tried if necessary—as soon as a schedule was found before $T_{lb} + 5$, we did not try any greater values of T. Using this approach and these parameters, we were able to obtain optimal solutions in the large majority of cases. However, in the future we plan to more systematically investigate how the values of these parameters effect the optimality of the results.

[5] We would also liked to have looked at performance when none of the function units were pipelined. Alas we have been unable to finish this before the submission deadline.

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

Table 5: Scheduling Performance for Schedules Found within Time Limit

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

Table 6: Difference between T and T_{min} for Schedules Found within Time Limit

To be specific, our approach found an optimal schedule $T = T_{min}$ for at least 75% of the loops. In fact, the optimal schedule is at the lower bound ($T = T_{lb}$) of what is possible in 69% of the loops as summarized in Table 5. As the gap indicates, 74 or 6% of the schedules were T_{min} but not T_{lb}. That is there was no feasible solution to the ILP at T_{lb} meaning that the lower bound was not tight in those cases. This fraction is similar to what others have found [16, 13] for function units without structural hazards. This still leaves 51 loops for which a *possible* T_{min} schedule was not found. We say a *possible* T_{min} here since there is no evidence — CPLEX' 3 minute time limit expired without indicating whether or not a schedule exists for this T_{min} period! Table 6 indicates for these 51 loops how far the schedule found was from a possible T_{min} schedule.

The third column of Table 5 is the arithmetic mean of the number of nodes in the DDG. As indicated by the third column our ILP approach *does* produce good schedules when the mean number of nodes in the DDG is less than 20.

Another important consideration in evaluating these schedules is the number of registers they require. More than 85% have fewer than 64 variable lifetimes while 81% have fewer than 32 variable lifetimes. These numbers do not include any coloring, i.e. they ignore the fact that some of the variable lifetimes might be able to share the same physical register. In addition, they lump together both integer and floating point values. Note that we can easily include register requirements in the ILP framework as demonstrated in [8], but we choose not to do so in this discussion for simplicity. [6]

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

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

Table 7: Time (in seconds) required for scheduling

Finally, how long did it take to get these schedules? The median time was 2 seconds, while the geometric mean was 5 seconds. Full results are summarized in Table 7.

6 Related Work

Software pipelining has been extensively studied [1, 3, 4, 5, 7, 13, 16, 17, 18, 22, 23, 26, 27, 28, 30]. Rau and Fisher provide a comprehensive survey of these works in [21]. As stated in [21], software pipelining methods vary in several aspects: (1) whether or not they consider finite resources, (2) whether they model simple or complex resource usage, (3) whether the algorithm is one-pass, iterative, incremental, or exhaustive search based. In particular, in terms of resource constraints, the work reported in [1, 27, 18, 20], does not consider any resource constraint while the methods reported in [22, 11, 2, 7, 17, 29, 28] deal with function unit constraints but with simple resource usage. Both function unit and register resource constraints are considered in [13, 5]. Software pipelining methods for complex usage patterns with limited function units was dealt with by [3, 16, 19]. The methods proposed in these works that aim to be implementable in a practical compiler all use heuristic approaches.

By contrast, in this paper, we develop a clear mathematical formulation of the software pipelining problem. Even though the use of an integer programming method may be unacceptable in a practical compiler, it can be used for performance-critical kernel loops. The proposed method is also useful in the context of a scheduling testbed where our optimal scheduling method can be used to judge how well other existing/newly proposed heuristics perform, and hence to improve them. In [8] we compared three heuristic algorithms with an ILP based scheduling method for clean pipelines. In [6] Feautrier independently gave an ILP formulation similar to our earlier work. In this work we have extended our ILP formulation to non-pipelined execution units and pipelines with structural hazards. The proposed formulation is able to schedule and map simultaneously and obtains a fixed FU assignment where each instruction i is executed on the same function unit $FU(i)$ throughout the loop execution. Our method is unique in that it combines scheduling and mapping in a unified framework and attempts to achieve an optimal solution. An advantage of our method is that it can be extended to handle multi-function pipelines as well. It can incorporate minimizing buffers (logical registers) as in [18] or minimizing the maximum number of live values at any time step in the repetitive pattern, as in the method proposed by Eichenberger, Davidson and Abraham [5].

7 Conclusions

In this paper we have proposed a method to formulate scheduling and mapping problems in a unified framework. Our method can handle non-pipelined execution units and pipelines with structural hazards. The schedules obtained are optimal (in terms of initiation interval) and run with a minimum number of function units (for a fixed FU assignment of each instruction in the loop). The proposed method can also handle multi-function pipelines. While the use of integer programming methods in practical compilers may not be acceptable, we feel that the proposed formulation is interesting and useful in its own way. It can be used to evaluate other heuristic methods, and also to derive optimal schedules for performance-critical applications where the user is willing to pay a high compilation cost for savings in runtime. The proposed method can be easily extended to incorporate minimizing other resources such as processor registers in a single framework.

We have implemented our scheduling method and run experiments on a set of 1066 loops taken from various benchmark suites such as `SPEC92`, the `NAS kernels`, `linpack` and `livermore`. Our initial experiments with scheduling these loops on an extended PowerPC-604 architecture were encouraging, yielding rate-optimal schedules for more than 75% of the loops.

In this paper we have considered only loops consisting of a single basic block. For loops involving conditionals, our method can be applied if suitable hardware support exists for predicated execution as in [24], or if Reverse-If-Conversion [30] is used on our scheduled code. We plan to investigate this in future.

Acknowledgments

Kemal Ebcioglu and Mayan Moudgill were instrumental in completing this paper. We also wish to thank Qi Ning, Vincent Van Dongen, and Philip Wong for the fruitful discussions we had with them. The authors are also thankful to IBM for its technical support of this work. The authors acknowledge Natural Science and Engineering Research Council (NSERC) and MICRONET, Network Centres of Excellence, for supporting this work.

References

[1] Alexander Aiken and Alexandru Nicolau. Optimal loop parallelization. In *Proc. of the SIGPLAN '88 Conf. on Programming Language Design and Implementation*, pages 308–

317, Atlanta, Georgia, Jun. 22–24, 1988. ACM SIGPLAN. *SIGPLAN Notices,* 23(7), Jul. 1988.

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

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

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

[5] Alexandre E. Eichenberger, Edward S. Davidson, and Santosh G. Abraham. Minimum register requirements for a modulo schedule. In *Proc. of the 27th Ann. Intl. Symp. on Microarchitecture*, pages 75–84, San Jose, Calif., Nov. 30–Dec.2, 1994. ACM SIGMICRO and IEEE-CS TC-MICRO.

[6] P. Feautrier. Tba. In *Cornell Compiler Workshop*, 1994.

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

[8] R. Govindarajan, Erik R. Altman, and Guang R. Gao. A framework for resource-constrained rate-optimal software pipelining. In Bruno Buchberger and Jens Volkert, editors, *Proc. of the Third Joint Intl. Conf. on Vector and Parallel Processing (CONPAR 94 – VAPP VI)*, number 854 in Lec. Notes in Comp. Sci., pages 640–651, Linz, Austria, Sep. 6–8, 1994. Springer-Verlag.

[9] R. Govindarajan, Erik R. Altman, and Guang R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proc. of the 27th Ann. Intl. Symp. on Microarchitecture*, pages 85–94, San Jose, Calif., Nov. 30–Dec.2, 1994. ACM SIGMICRO and IEEE-CS TC-MICRO.

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

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

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

[13] Richard A. Huff. Lifetime-sensitive modulo scheduling. In *Proc. of the SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pages 258–267, Albuquerque, N. Mex., Jun. 23–25, 1993. ACM SIGPLAN. *SIGPLAN Notices,* 28(6), Jun. 1993.

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

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

[16] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proc. of the SIGPLAN '88 Conf. on Programming Language Design and Implementation*, pages 318–328, Atlanta, Georgia, Jun. 22–24, 1988. ACM SIGPLAN. *SIGPLAN Notices,* 23(7), Jul. 1988.

[17] Soo-Mook Moon and Kemal Ebcioğlu. An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. In *Proc. of the 25th Ann. Intl. Symp. on Microarchitecture*, pages 55–71, Portland, Ore., Dec. 1–4, 1992. ACM SIGMICRO and IEEE-CS TC-MICRO.

[18] Q. Ning and G. R. Gao. A novel framework of register allocation for software pipelining. In *Conf. Rec. of the Twentieth Ann. ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 29–42, Charleston, South Carolina, Jan. 10–13, 1993.

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

[20] J. Ramanujam. Optimal software pipelining of nested loops. In *Proc. of the 8th Intl. Parallel Processing Symp.*, pages 335–342, Cancún, Mexico, Apr. 26–29, 1994. IEEE Comp. Soc.

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

[22] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proc. of the 14th Ann. Microprogramming Work.*, pages 183–198, Chatham, Mass., Oct. 12–15, 1981. ACM SIGMICRO and IEEE-CS TC-MICRO.

[23] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. In *Proc. of the SIGPLAN '92 Conf. on Programming Language Design and Implementation*, pages 283–299, San Francisco, Calif., Jun. 17–19, 1992. ACM SIGPLAN. *SIGPLAN Notices,* 27(7), Jul. 1992.

[24] B. Ramakrishna Rau, David W. L. Yen, Wei Yen, and Ross A. Towle. The Cydra 5 departmental supercomputer. *Computer*, 22(1):12–35, Jan. 1989.

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

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

[27] V. Van Dongen, G. R. Gao, and Q. Ning. A polynomial time method for optimal software pipelining. In *Proc. of the Conf. on Vector and Parallel Processing, CONPAR-92*, number 634 in Lec. Notes in Comp. Sci., pages 613–624, Lyon, France, Sep. 1–4, 1992. Springer-Verlag.

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

[29] Nancy J. Warter, John W. Bockhaus, Grant E. Haab, and Krishna Subramanian. Enhanced modulo scheduling for loops with conditional branches. In *Proc. of the 25th Ann. Intl. Symp. on Microarchitecture*, pages 170–179, Portland, Ore., Dec. 1–4, 1992. ACM SIGMICRO and IEEE-CS TC-MICRO.

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

A Scheduling for Clean Execution Units

In this section, we briefly review our ILP formulation for clean pipelines. For more details readers are referred to [9].

Let us show the software pipelining problem for a value of $T \geq T_{lb}$ for architectures with clean pipelines. We are interested in linear periodic schedules of the form $j \cdot T + t_i$. The value of t_i variables are chosen such that the the data dependency constraints specified in the DDG are satisfied. Reiter showed this translates to the following linear constraint on t_i values [25].

$$t_j - t_i \geq d_i - T \cdot m_{ij}, \qquad \forall (i,j) \in E \tag{11}$$

In order to represent the resource constraints in a linear form, define a pair of values k_i and o_i for each t_i, such that

$$k_i = \left\lfloor \frac{t_i}{T} \right\rfloor \quad \text{and} \quad o_i = t_i \mod T.$$

Further, we write the o_i variable in terms of a set of 0-1 integer variables as shown below.

$$o_i = [a_{0,i}, a_{1,i}, \cdots, a_{(T-1),i}] \times [0, 1, \cdots (T-1)]^{Transpose}$$

The variable $a_{t,i}$ is 1 if any only if instruction i is scheduled at time step t in the repetitive pattern. Otherwise $a_{t,i} = 0$. Since each instruction i must be scheduled for execution at some time step t in the repetitive pattern, we have the condition

$$\sum_{t=0}^{T-1} a_{t,i} = 1, \forall i \in V \tag{12}$$

In a similar way, all the t_i variables can be related to k_i and $a_{t,i}$.

$$\mathcal{T} = T \cdot \mathcal{K} + \mathcal{A}^{\text{Transpose}} \times [0, 1, \cdots, T-1]^{\text{Transpose}} \tag{13}$$

where

$$\mathcal{T} = \begin{bmatrix} t_0 \\ t_1 \\ \vdots \\ t_{N-1} \end{bmatrix}, \ \mathcal{K} = \begin{bmatrix} k_0 \\ k_1 \\ \vdots \\ k_{N-1} \end{bmatrix} \text{ and } \mathcal{A} = \begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,(N-1)} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,(N-1)} \\ \vdots & \vdots & & \vdots \\ a_{(T-1),0} & a_{(T-1),1} & \cdots & a_{(T-1),(N-1)} \end{bmatrix}$$

Now we need to determine the resource requirement at each time step in the repetitive pattern. Since each function unit is fully pipelined, the number of resources required at the t-th time step in the repetitive pattern is equal to the number of instructions initiated at time step t. An instruction i is initiated at the t-th time step in the repetitive pattern if $a_{t,i} = 1$. Thus, the sum of $a_{t,i}$ for all instructions i which are executed on a function unit type r gives the resource requirement at time step t. If R_r is the number of function units of type r required for a schedule, then clearly

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

where $\mathcal{I}(r)$ is the set of all instructions that require function units of type r.

The ILP formulation is to minimize the a weighted sum of R_r for each function unit type r. If the weight associated with function unit type r is C_r. then the objective function is

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

The complete integer programming formulation is shown below.

[ILP Formulation for Clean Pipelines]

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

subject to

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

$$\mathcal{T} = T \cdot \mathcal{K} + \mathcal{A}^{\text{Transpose}} \times [0, 1, \cdots, T-1]^{\text{Transpose}} \tag{14}$$

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

$$t_j - t_i \geq d_i - T \cdot m_{ij} \quad \forall (i,j) \in E \tag{15}$$

$$t_i \geq 0, k_i \geq 0, \text{ and } a_{t,i} \geq 0 \text{ are integers} \quad \forall i \in [0, N-1], \forall t \in [0, T-1] \tag{16}$$

We will show the complete formulation for our motivating example in Section D. In the following section we develop the formulation necessary for guaranteeing fixed FU assignment.

B Formulations

[ILP Formulation for Non-Pipelined Architectures]

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

subject to

$$\sum_{i \in \mathcal{I}(r)} u_{t,i} \leq R_r, \quad \forall r \in [0, h-1] \text{ and } \forall t \in [0, T-1] \tag{17}$$

$$u_{t,i} = \sum_{l=0}^{(d_i-1)} a_{((t-l)\%T),i} \quad \forall t \in [0, T-1], \forall i \in E \tag{18}$$

$$\mathcal{T} = T \cdot \mathcal{K} + \mathcal{A}^{\text{Transpose}} \times [0, 1, \cdots, T-1]^{\text{Transpose}} \tag{19}$$

$$\sum_{t=0}^{T-1} a_{t,i} = 1 \quad \forall i \in [0, N-1] \tag{20}$$

$$t_j - t_i \geq d_i - T \cdot m_{ij} \quad \forall (i,j) \in E \tag{21}$$

$$t_i \geq 0, k_i \geq 0, a_{t,i} \geq 0 \text{ and } u_{t,i} \geq 0 \text{ are integers} \quad \forall i \in [0, N-1], \forall t \in [0, T-1] \tag{22}$$

[ILP Formulation for Pipelines with Hazards]

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

subject to

$$R_r \geq \sum_{i \in \mathcal{I}(r)} U^s[t,i], \quad \forall s \in [0, s_r - 1], \forall t \in [0, T-1], \text{ and } \forall r \in [0, h-1] \tag{23}$$

$$c_i \leq R_r \quad \forall i \in \mathcal{I}(r), \text{ and } r \in [0, h-1] \tag{24}$$

$$c_i - c_j \geq (U^s[t,i] + U^s[t,j] - 1)/2 - N \cdot w_{ij} \quad \forall i,j \in \mathcal{I}(r), \ \forall t \in [0, T-1], \text{ and } \forall r \in [0, h-1] \tag{25}$$

$$c_j - c_i \geq (U^s[t,i] + U^s[t,j] - 1)/2 - N \cdot (1 - w_{ij}) \quad \forall i,j \in \mathcal{I}(r), \ \forall t \in [0, T-1], \text{ and } \forall r \in [0, h-1] \tag{26}$$

$$\mathcal{T} = T \cdot \mathcal{K} + \mathcal{A}^{\text{Transpose}} \times [0, 1, \cdots, T-1]^{\text{Transpose}} \tag{27}$$

$$\sum_{t=0}^{T-1} a_{t,i} = 1 \quad \forall i \in [0, N-1] \tag{28}$$

$$t_j - t_i \geq d_i - T \cdot m_{ij} \quad \forall (i,j) \in E \tag{29}$$

$$U^s[t,i] = \sum_{l=0}^{(T-1)} a_{((t-l)\%T),i} \cdot RT_r^s[l] \quad \forall t \in [0, T-1], \forall i \in \mathcal{I}(r), \forall s \in [0, s_r - 1] \text{ and } r \in [0, h-1] \tag{30}$$

$$t_i \geq 0, k_i \geq 0, a_{t,i} \geq 0, c_i \geq 1, 0 \leq d_{i,j} \leq 1, \text{ and } U^s[t,i] \geq 0 \text{ are integers} \quad \forall i \in [0, N-1], \forall t \in [0, T-1], \forall s \in [0, s_r - 1] \tag{31}$$

C Proof of Coloring Theorem

In this appendix we show that our coloring formulation is correct, that is we prove the Theorem 3.1.

Theorem 3.1 *Equations 6 – 8 require that two nodes i and j be assigned different colors (mapped to different function units) if and only if they overlap at some time.*

For clarity Equations 6 – 8 are repeated below:

$$c_i - c_j \geq \frac{u_{t,i} + u_{t,j} - 1}{2} - N \cdot w_{i,j} \tag{32}$$

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

$$1 \leq c_i \leq N \quad \forall k \in [0, N-1] \tag{34}$$

Next we establish a bound for $c_i - c_j$.

Lemma C.1 *For all nodes i and j,* $-N + 1 \leq c_i - c_j \leq N - 1$.

Proof: This lemma follows from Equation 34. □

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

Same Color: There are 3 subcases.

1. Nodes i and j overlap at some time t. This being the case, $u_{t,i}$ and $u_{t,j}$ are both 1 reducing Equations 32 and 33 to

$$c_i - c_j \geq \frac{1}{2} - N \cdot w_{i,j} \tag{35}$$

$$c_j - c_i \geq \frac{1}{2} - N \cdot (1 - w_{i,j}) \tag{36}$$

Noting that $c_i - c_j = 0$, rearranging terms yields

$$w_{i,j} \geq \frac{1}{2 \cdot N} \tag{37}$$

$$w_{i,j} \leq 1 - \frac{1}{2 \cdot N} \tag{38}$$

Recall that $w_{i,j}$ is an integer 0-1 variable. However, if $w_{i,j} = 0$, then Equation 37 is not satisfied: $0 \geq \frac{1}{2 \cdot N}$ is false since $N \geq 1$. Similarly if $w_{i,j} = 1$, then Equation 38 is not satisfied: $1 \leq 1 - \frac{1}{2 \cdot N}$ is false. Thus, as desired c_i cannot be equal to c_j when i and j overlap.

2. Nodes i and j do not ever overlap, and at time t, only one is active ($u_{t,i} = 1$ or $u_{t,j} = 1$, but not both). In either case, Equations 32 and 33 reduce to

$$\begin{aligned} c_i - c_j &\geq -N \cdot w_{i,j} & (39) \\ c_j - c_i &\geq -N \cdot (1 - w_{i,j}) & (40) \end{aligned}$$

Again noting that $c_i - c_j = 0$, rearranging terms yields

$$\begin{aligned} w_{i,j} &\geq 0 & (41) \\ w_{i,j} &\leq 1 & (42) \end{aligned}$$

Both Equation 41 and 42 are clearly true, whether $w_{i,j} = 0$ or $w_{i,j} = 1$.

3. Nodes i and j do not ever overlap, and at time t, neither is active ($u_{t,i} = 0$ and $u_{t,j} = 0$). In this case, Equations 32 and 33 reduce to

$$\begin{aligned} c_i - c_j &\geq -\frac{1}{2} - N \cdot w_{i,j} & (43) \\ c_j - c_i &\geq -\frac{1}{2} - N \cdot (1 - w_{i,j}) & (44) \end{aligned}$$

The usual rearrangement of terms yields

$$\begin{aligned} w_{i,j} &\geq -\frac{1}{2 \cdot N} & (45) \\ w_{i,j} &\leq 1 + \frac{1}{2 \cdot N} & (46) \end{aligned}$$

Again, both Equation 45 and 46 can be satisfied by choosing either $w_{i,j} = 0$ or $w_{i,j} = 1$.

It remains to show that Equations 32 and 33 are both satisfied whenever two nodes are assigned **different** colors. We do this by breaking the problem into four parts depending on whether $c_i - c_j > 0$ or $c_i - c_j < 0$ and whether $w_{i,j} = 0$ or $w_{i,j} = 1$. The possibilities are illustrated by the table below:

Condition	I	II	III	IV
$c_i - c_j > 0$	True	False	True	False
$w_{i,j}$	0	1	1	0
$-N \cdot w_{i,j}$	0	-N	-N	0
$-N(1 - w_{i,j})$	-N	0	0	-N
Eqn 32 Min LHS	1	**-N+1**	1	**-N+1**
Eqn 32 Max RHS	0.5	-N+0.5	-N+0.5	0.5
LHS $\geq$ RHS Always?	Yes	Yes	Yes	No
Eqn 33 Min LHS	**-N+1**	1	**-N+1**	1
Eqn 33 Max RHS	-N+0.5	0.5	0.5	-N+0.5
LHS $\geq$ RHS Always?	Yes	Yes	No	Yes
Eqn 32 Max LHS				-1
Eqn 32 Min RHS				-0.5
LHS $\geq$ RHS Never?				Yes
Eqn 33 Max LHS			-1	
Eqn 33 Min RHS			-0.5	
LHS $\geq$ RHS Never?			Yes	

The four boldface **-N+1** lower bounds of the *Left Hand Sides* are all due to Lemma C.1 which gives the minimum possible value of $c_i - c_j$. As is shown below, to be in the "proper operating region," Equations 32 and 33 require this lower bound (implied by Equation 34).

Now to see that both Equation 32 and 33 are always satisfied whenever two nodes are assigned different colors, first note that if $(c_i - c_j)$ is positive, then $w_{i,j} = 0$, and if $(c_i - c_j)$ is negative (the only other possibility if $c_i \neq c_j$), then $w_{i,j} = 1$. This is because when $(c_i - c_j)$ is positive and $w_{i,j} = 1$, (i.e. in column III), Equation 33 is never satisfied. As the last 3 rows of the table indicate, the *maximum* value of the left hand side of Equation 33 in this case is -1 while the *minimum* value of the right hand side is -0.5. In other words the left hand side is never greater than or equal to the right hand side as required by Equation 33. Equation 32 is likewise never satisfied when $(c_i - c_j)$ is negative and $w_{i,j} = 0$ (as illustrated in the second to last 3 rows of column IV in the Table).

Conversely in columns I and II of the table, both Equations 32 and 33 are always satisfied. Intuitively, this is what is wanted. For example if $c_i > c_j$ and $w_{i,j} = 0$ as in column I, then Equations 32 and 33 become:

$$c_i - c_j \geq 0.5 \tag{47}$$
$$c_j - c_i \geq 0.5 - N \tag{48}$$

Equation 47 is exactly the equation we would want if we knew (a) that i and j would overlap and (b) the sign of $c_i - c_j$ ahead of time, as Equation 47 requires c_i and c_j to be different. In this case, ($c_i > c_j$ and $w_{i,j} = 0$) Equation 48 imposes no additional constraint.

Conversely if $c_i < c_j$ and $w_{i,j} = 1$ as in column II, then Equations 32 and 33 become:

$$\begin{aligned} c_i - c_j &\geq 0.5 - N && (49) \\ c_j - c_i &\geq 0.5 && (50) \end{aligned}$$

In this case, it is the second equation, Equation 50 that gives the equation we would want if we knew that i and j would overlap and the sign of $c_i - c_j$ ahead of time — with Equation 49 imposing no additional constraint.

Thus Equations 32 and 33 behave as desired in all cases, requiring different colors when nodes overlap and permitting nodes to have the same color when they do not.

D Formulation for the Motivating Example

In this section we will show the formulation for the motivating example for the case non-pipelined function units. We will assume that $T = 4$ for which a valid schedule exists. First we will show the complete resource usage matrix U.

$$U = \begin{bmatrix} a_{0,0} & (a_{0,1} + a_{3,1}) & (a_{0,2} + a_{3,2}) & (a_{0,3} + a_{3,3}) & (a_{0,4} + a_{3,4}) & a_{0,5} \\ a_{1,0} & (a_{1,1} + a_{0,1}) & (a_{1,2} + a_{0,2}) & (a_{1,3} + a_{0,3}) & (a_{1,4} + a_{0,4}) & a_{1,5} \\ a_{2,0} & (a_{2,1} + a_{1,1}) & (a_{2,2} + a_{1,2}) & (a_{2,3} + a_{1,3}) & (a_{2,4} + a_{1,4}) & a_{2,5} \\ a_{3,0} & (a_{3,1} + a_{2,1}) & (a_{3,2} + a_{2,2}) & (a_{3,3} + a_{2,3}) & (a_{3,4} + a_{2,4}) & a_{3,5} \end{bmatrix} \quad (51)$$

If R_{Int}, R_{FP} and R_{LS} represent the number of integer, floating point, and load/store execution unit required for the schedule, then

$$\begin{array}{lll} R_{Int} \geq u_{0,0} & R_{LS} \geq u_{0,1} + u_{0,5} & R_{FP} \geq u_{0,2} + u_{0,3} + u_{0,4} \\ R_{Int} \geq u_{1,0} & R_{LS} \geq u_{1,1} + u_{1,5} & R_{FP} \geq u_{1,2} + u_{1,3} + u_{1,4} \\ R_{Int} \geq u_{2,0} & R_{LS} \geq u_{2,1} + u_{2,5} & R_{FP} \geq u_{2,2} + u_{2,3} + u_{2,4} \\ R_{Int} \geq u_{3,0} & R_{LS} \geq u_{3,1} + u_{3,5} & R_{FP} \geq u_{3,2} + u_{3,3} + u_{3,4} \end{array} \quad (52)$$

The coloring equation must deal with two sets one involving instructions s_1 and s_5 for the Load/Store function unit, and the other involving instructions s_2, s_3, and s_4 for the Floating point function units. Since there is only one instruction (s_0) that uses the integer function unit, it need not be considered for the coloring part. Equations (6) and (7) for the above two sets are:

$$
\begin{array}{ll}
c_1 - c_5 \geq (u_{0,1} + u_{0,5} - 1)/2 - 6w_{1,5} & c_5 - c_1 \geq (u_{0,1} + u_{0,5} - 1)/2 - 6(1 - w_{1,5}) \\
c_1 - c_5 \geq (u_{1,1} + u_{1,5} - 1)/2 - 6w_{1,5} & c_5 - c_1 \geq (u_{1,1} + u_{1,5} - 1)/2 - 6(1 - w_{1,5}) \\
c_1 - c_5 \geq (u_{2,1} + u_{2,5} - 1)/2 - 6w_{1,5} & c_5 - c_1 \geq (u_{2,1} + u_{2,5} - 1)/2 - 6(1 - w_{1,5}) \\
c_1 - c_5 \geq (u_{3,1} + u_{3,5} - 1)/2 - 6w_{1,5} & c_5 - c_1 \geq (u_{3,1} + u_{3,5} - 1)/2 - 6(1 - w_{1,5}) \\
 & \\
c_2 - c_3 \geq (u_{0,2} + u_{0,3} - 1)/2 - 6w_{2,5} & c_3 - c_2 \geq (u_{0,2} + u_{0,3} - 1)/2 - 6(1 - w_{2,3}) \\
c_2 - c_3 \geq (u_{1,2} + u_{1,3} - 1)/2 - 6w_{2,3} & c_3 - c_2 \geq (u_{1,2} + u_{1,3} - 1)/2 - 6(1 - w_{2,3}) \\
c_2 - c_3 \geq (u_{2,2} + u_{2,3} - 1)/2 - 6w_{2,3} & c_3 - c_2 \geq (u_{2,2} + u_{2,3} - 1)/2 - 6(1 - w_{2,3}) \\
c_2 - c_3 \geq (u_{3,2} + u_{3,3} - 1)/2 - 6w_{2,3} & c_3 - c_2 \geq (u_{3,2} + u_{3,3} - 1)/2 - 6(1 - w_{2,3}) \\
 & \\
c_2 - c_4 \geq (u_{0,2} + u_{0,4} - 1)/2 - 6w_{2,4} & c_4 - c_2 \geq (u_{0,2} + u_{0,4} - 1)/2 - 6(1 - w_{2,4}) \\
c_2 - c_4 \geq (u_{1,2} + u_{1,4} - 1)/2 - 6w_{2,4} & c_4 - c_2 \geq (u_{1,2} + u_{1,4} - 1)/2 - 6(1 - w_{2,4}) \\
c_2 - c_4 \geq (u_{2,2} + u_{2,4} - 1)/2 - 6w_{2,4} & c_4 - c_2 \geq (u_{2,2} + u_{2,4} - 1)/2 - 6(1 - w_{2,4}) \\
c_2 - c_4 \geq (u_{3,2} + u_{3,4} - 1)/2 - 6w_{2,4} & c_4 - c_2 \geq (u_{3,2} + u_{3,4} - 1)/2 - 6(1 - w_{2,4}) \\
 & \\
c_3 - c_4 \geq (u_{0,3} + u_{0,4} - 1)/2 - 6w_{3,4} & c_4 - c_3 \geq (u_{0,3} + u_{0,4} - 1)/2 - 6(1 - w_{3,4}) \\
c_3 - c_4 \geq (u_{1,3} + u_{1,4} - 1)/2 - 6w_{3,4} & c_4 - c_3 \geq (u_{1,3} + u_{1,4} - 1)/2 - 6(1 - w_{3,4}) \\
c_3 - c_4 \geq (u_{2,3} + u_{2,4} - 1)/2 - 6w_{3,4} & c_4 - c_3 \geq (u_{2,3} + u_{2,4} - 1)/2 - 6(1 - w_{3,4}) \\
c_3 - c_4 \geq (u_{3,3} + u_{3,4} - 1)/2 - 6w_{3,4} & c_4 - c_3 \geq (u_{3,3} + u_{3,4} - 1)/2 - 6(1 - w_{3,4})
\end{array}
\tag{53}
$$

The colors assigned must be less than the available function units in the corresponding type. This is specified by

$$
R_{LS} \geq c_1 \quad R_{LS} \geq c_5 \quad R_{FP} \geq c_2 \quad R_{FP} \geq c_3 \quad R_{FP} \geq c_4 \tag{54}
$$

The dependency constraints (refer to Equation (21)) for the DDG are:

$$
\begin{array}{llll}
t_1 - t_0 \geq 1 & t_5 - t_0 \geq 1 & t_2 - t_1 \geq 2 & t_4 - t_1 \geq 2 \\
t_3 - t_2 \geq 2 & t_4 - t_3 \geq 2 & t_5 - t_4 \geq 2 &
\end{array}
\tag{55}
$$

To guarantee that each instruction is scheduled once in the repetitive pattern,

$$
\begin{array}{lll}
a_{0,0} + a_{1,0} + a_{2,0} + a_{3,0} = 1 & a_{0,1} + a_{1,1} + a_{2,1} + a_{3,1} = 1 & a_{0,2} + a_{1,2} + a_{2,2} + a_{3,2} = 1 \\
a_{0,3} + a_{1,3} + a_{2,3} + a_{3,3} = 1 & a_{0,4} + a_{1,4} + a_{2,4} + a_{3,4} = 1 & a_{0,5} + a_{1,5} + a_{2,5} + a_{3,5} = 1
\end{array}
\tag{56}
$$

The t_i variables are related to $a_{t,i}$ as in the following equations.

$$
\begin{aligned}
4 \cdot k_0 + 0 \cdot a_{0,0} + 1 \cdot a_{1,0} + 2 \cdot a_{2,0} + 3 \cdot a_{3,0} &= t_0 \\
4 \cdot k_1 + 0 \cdot a_{0,1} + 1 \cdot a_{1,1} + 2 \cdot a_{2,1} + 3 \cdot a_{3,1} &= t_1 \\
4 \cdot k_2 + 0 \cdot a_{0,2} + 1 \cdot a_{1,2} + 2 \cdot a_{2,2} + 3 \cdot a_{3,2} &= t_2 \\
4 \cdot k_3 + 0 \cdot a_{0,3} + 1 \cdot a_{1,3} + 2 \cdot a_{2,3} + 3 \cdot a_{3,3} &= t_3 \\
4 \cdot k_4 + 0 \cdot a_{0,4} + 1 \cdot a_{1,4} + 2 \cdot a_{2,4} + 3 \cdot a_{3,4} &= t_4 \\
4 \cdot k_5 + 0 \cdot a_{0,5} + 1 \cdot a_{1,5} + 2 \cdot a_{2,5} + 3 \cdot a_{3,5} &= t_5 \qquad (57)
\end{aligned}
$$

The relation between the $u_{t,i}$ variables and the $a_{t,i}$ variables are as specified by the U matrix in Equation (51).

The ILP formulation is to

$$\text{minimize} \quad C_{Int} \cdot R_{Int} + C_{FP} \cdot R_{FP} + C_{LS} \cdot R_{LS}$$

subject to Equations (51) – (57). Solving the integer programming problem we get the schedule shown in Table 2.

E Reservation Tables used in Experiments

Reservation Tables used for 604 Function Units:

	Time Steps
	0
Stage 1	x

Integer ALU's

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

Floating Point Add Units

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

Floating Point Mult Units

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

Load Units

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

Store Units

	Time Steps																	
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Stage 1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Floating Point Div Units