McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures and Parallel Systems

Co-Scheduling Hardware and Software Pipelines

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

ACAPS Technical Memo 92
January 12, 1995

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

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

Abstract

Classical pipeline theory was well developed two decades ago in the context of uniprocessor and vector architectures. Since then tremendous progress has been made in computer architecture to exploit higher instruction level parallelism on highly pipelined architectures with (or without) structural hazards. Simultaneously, advances have been made in compiling techniques, such as *software pipelining*, that expose instruction level parallelism for the architecture. However, the scheduling of hardware and the software pipelines hasn't been done together to take greater advantage of available parallelism.

In this paper, we demonstrate that this can be done for a special class of pipelines, viz. *arithmetic pipelines* by *Co-Scheduling*, a framework for simultaneous design of hardware pipelines structures and software pipelined schedules. We introduce and develop two components of the Co-Scheduling framework:

- A theory of pipeline architectures which governs hardware pipeline design to meet the needs of periodic (or software pipelined) schedules. Reservation tables, forbidden latencies, collision vectors, and state diagrams from classical pipeline theory are revisited and extended for solving the new problems.
- Based on the extended pipeline theory, we present an efficient method to perform (a) software pipeline scheduling and (b) hardware pipeline (delay) reconfiguration which are mutually "compatible".

The proposed method has been implemented and preliminary experimental results for 1013 kernel loops are reported. Co-scheduling successfully obtains a schedule for 98% of these loops. Over 60% of these schedules have the minimum possible initiation interval, and the median time to obtain these schedules is less than 0.3 seconds on a Sparc-10/30.

A salient feature of the *Co-Scheduling* framework is that it is amenable to the well-established delay insertion technique to increase the number of initiations in a function unit which, in turn, can improve the software pipeline initiation interval.

Keywords: Pipeline Architecture, Software Pipelining, Classical Pipeline Theory, Co-Scheduling, VLIW/Superscalar

Contents

List of Figures

List of Tables

1 Introduction

Pipelining is one of the most efficient means of improving performance in high-end processor architectures. Historically, hardware pipeline design techniques have been used successfully in vector and pipelined supercomputers. Classical hardware pipeline design theory developed more than 2 decades ago was driven by this need [13, 10]. Recent advances in VLSI technology make it feasible to design even more aggressive pipelines to exploit higher instruction level parallelism in high-performance (superscalar, VLIW, superpipelined) microprocessor architectures.

In the meantime, a compiling technique — known as software pipelining — has become increasingly popular for aggressive loop scheduling. A software pipelined schedule overlaps operations from different loop iterations in an attempt to fully exploit instruction level parallelism. In fact, as pointed out by Hennessy and Patterson, software pipelining is among "the most significant open research areas in pipelined processor design" [7].

Software pipelining techniques must take into account the function unit constraints of those machines. A variety of software pipelining algorithms [16, 8, 11, 1, 2, 6, 9, 21, 3, 5] have been proposed which operate under resource constraints. An excellent survey of these algorithms can be found in [15]. The software pipelining methods cited above are capable of handling simple as well as complex resource usage patterns, and are specifically tuned towards *instruction pipelines* [10], where structural hazards exist due mainly to sharing of resources, e.g. *Result Buses* or *Register Ports* that are shared by different stages of an instruction pipeline.

In this paper we consider software pipelining for architectures where each function unit — often treated previously as one stage in an instruction pipeline — may itself be an independent *arithmetic pipeline* [10] with complex structural hazards. (We often use the term *hardware pipeline* to refer to such pipelines.) This adds another dimension of complexity to that encountered with structural hazards arising from resource sharing between different stages of an instruction pipeline. We demonstrate that when such hardware pipelines are present, the use of classical pipeline theory can greatly improve both the quality of the schedule and the time required for its construction. This is achieved by integrating the scheduling of hardware and software pipelines in a unified framework. We term this framework *Co-Scheduling.* The basic observations that lead to the Co-Scheduling framework are:

Observation 1: Use of hardware pipeline scheduling theory provides key heuristics for improving software pipelining. In particular, by analyzing the reservation table of a *hardware* (arithmetic) pipeline, we can derive "good" initiation sequences that better utilize the pipeline. Use of such initiation sequences narrows the search space of *software* pipelined schedules, which in turn, facilitates obtaining schedules with a shorter initiation interval (**II**), as well as reduces the time to construct a schedule. In Section 2, we give a number of motivating examples.

Observation 2: Classical pipeline scheduling theory cannot be directly applied in software pipelining. The primary missing link is that hardware and software pipeline cycles may have different periods. Motivating examples for this are also provided in Section 2.

Observation 3: It may be possible to reconfigure the hardware pipeline (by the introduction of delays) to improve the number of initiations, thereby reducing the initiation interval **II** of the schedule. However, from **Observation 2**, such tuning cannot be performed without knowing the software pipelining **II**.

We introduce and develop two components of the *Co-Scheduling* framework:

(1) We extend classical pipeline theory by embedding the information on the software pipeline initiation interval **II** in the hardware pipeline structures. This is accomplished by extending reservation tables to *cyclic reservation tables (CRT)*. Important concepts such as *forbidden latencies, collision vectors,* and *state diagrams* from classical pipeline theory are revisited for CRTs.

(2) We use a *Modulo Initiation Table* (MIT) which represents *only* the initiation pattern of a software pipelined schedule: the resource usage and the structural constraints imposed by the reservation table are captured succinctly by the CRT. Based on the MIT, we present an efficient method for both constructing the software pipeline schedule **S** and "adjusting" the configuration of hardware pipeline delays to "match" **S**.

There is an important difference between our CRT/MIT and the notion of *modulo reservation table* (MRT) used in [16, 11, 3, 9, 21, 5]. In the presence of structural hazards, the MRT combines the resource usage of all hardware units (e.g. pipeline stages) used by operations in the software pipelined loop body. We on the other hand, use the MIT, to describe only the software pipelined schedule, while letting the CRT describe the hardware (pipeline stage) usage pattern from periodic initiation of each operation. Use of both the CRT and MIT is important: the (extended) pipeline theory based on CRT provides key insights and guidelines in constructing software schedules based on MIT. Conversely, the requirements from software pipelining based on MIT guides the adjustment of hardware structures to accommodate the best possible software pipelining initiation rate (by judiciously introducing delays in the pipeline). Such an approach greatly simplifies software pipelining and is especially attractive for scheduling pipelined function units.

The proposed method has been implemented and preliminary experimental results are reported. The major observations include (1) by making use of the permissible latencies discovered by Co-Scheduling, a schedule was found for 61% of the loops at the minimum possible initiation interval **II**, (2) rough, simply calculated bounds for **II** are as good as more elaborately computed bounds and (3) our delay approach was a success—it allowed greater pipeline utilization than would otherwise been the case, thus permitting shorter initiation intervals to be achieved. Furthermore in doing this, delay insertion never increased the latency of a pipeline.

Co-Scheduling is not just yet another software pipelining method. It has the potential to make of use of classical techniques such as delay insertion to improve pipeline utilization. **Observation 3** alludes to such techniques. In *reconfigurable* pipelines, buffer (delay) stages can be added to improve the utilization of the non-buffer stages. Such delay insertion techniques have a wider application and usefulness when done in conjunction with software pipelining. The increasing amount of instruction level parallelism exposed by Co-Scheduling and other compiler techniques encourages the design of complex pipelines with structural hazards. This is because such pipelines are space efficient thereby permitting a larger number of pipelines and registers to be placed on one chip. This in turn provides more opportunities to exploit a greater amount of instruction-level parallelism.

The Co-Scheduling framework developed in this paper should be viewed as a complement to other related work in resource-constrained software pipelining. Although it is most effective for handling architectures with hardware (arithmetic) pipelines, we anticipate it can be used together with *modulo scheduling techniques* (e.g. [16, 11, 9, 15, 18]) to deal situations where both types of hazards (i.e. hazards due to resource sharing between stages of instruction pipelines and hazards internal to the arithmetic pipelines) co-exist.

In the following section we motivate the need for the Co-Scheduling framework with a number of examples. In Section 3, the classical pipeline theory is revisited in the context of software pipelining. We present the Co-Scheduling framework in Section 4. Implementation of Co-Scheduling and some preliminary results are discussed in Section 5. Section 6 compares our approach to other related work. Discussion on future work is presented in Section 7 and concluding remarks in Section 8.

2 Background and Motivation

In this section, we highlight the issues in constructing software pipelined schedules for pipelined architectures with structural hazards. We illustrate the issues with the help of several examples that motivate our approach to *co-scheduling* the hardware and software pipelines. First we briefly review elements of software pipelining important to our approach and introduce several terms used in this paper.

2.1 Background

In software pipelining, we focus on periodic *linear schedules* under which an instruction i in iteration j is initiated at time $j * \mathbf{II} + t_i$, where **II** is the *initiation interval* or period of the schedule and t_i is a constant. For more background information on linear scheduling, refer to the survey paper by Rau and Fisher [15]. The minimum initiation interval (**MII**) is constrained by both *loop-carried* dependences (or recurrences) and available resources [16, 11, 9, 15, 3]. Loop-carried dependences put a lower bound, **RecMII**, on **MII**. The value of **RecMII** is determined by the critical (dependence) cycle(s) [19] in the Data Dependency Graph (DDG) of the loop. Specifically

$$\textbf{RecMII} = \left\lceil \frac{\text{sum of instruction execution times}}{\text{sum of dependence distances}} \right\rceil \tag{1}$$

along the critical cycle(s)[1].

Another lower bound **ResMII** on **MII** is enforced by resource constraints. Suppose an instruction uses a pipeline stage s of a function unit (`FU`) type r (e.g. ADDER), for $d_{r,s}$ cycles. If there are N_r instructions that execute on `FU` type r and there are F_r `FU`s, then clearly any schedule will have **II** greater than $\lceil N_r * d_{r,s}/F_r \rceil$. Thus **ResMII** is the maximum of this bound taken over all stages and `FU` types:

$$\textbf{ResMII} = \max_r \left(\max_s \left\lceil \frac{N_r * d_{r,s}}{F_r} \right\rceil \right) \tag{2}$$

Lastly, the Minimum Initiation Interval **MII** is the maximum of **RecMII** and **ResMII**. That is,

$$\textbf{MII} = \max \ (\textbf{RecMII}, \textbf{ResMII}). \tag{3}$$

However there may or may not exist a schedule with a period **MII** satisfying the given resource constraints.

Like most software pipelining methods, we assume that an instruction i (from all iterations) will always be executed on the same function unit (`FU`) during the course of the loop execution. This fixed mapping of instruction (task) to `FU` is essential in a VLIW architecture where a specific operation (part) of the instruction word is linked with a function unit in the architecture.

In hardware pipelines, the resource usage of various pipeline stages are represented by a two dimensional *Reservation Table* [10]. If two operations entering a pipeline f cycles apart would subsequently require one (or more) of the pipeline stages at the same time, f is termed a **forbidden latency**. Operations separated by **permissible latencies** have no such conflicts.

A **collision vector** has length equal to the pipeline latency and contains a **1** at all forbidden latencies, and a **0** at all permissible latencies. Assume the leftmost position in the collision vector represents time **0** and that the pipeline is currently empty. When an operation is begun, (1) The collision vector is copied to a new vector **v**; (2) At each cycle, **v** is shifted left by **1**; (3) If the leftmost bit of **v** is **0**, a new operation may be initiated, otherwise not; (4) If a new operation is initiated, **v** is `OR`'ed with the initial collision vector. A **state diagram** may be constructed listing all such sequences of initiations.

Analysis of this state diagram reveals what initiation sequence(s) or **latency sequences** maximize the utilization and throughput of the pipeline. This, the classical theory of hardware pipelines, was well developed and effectively employed to achieve high performance in pipeline and vector architectures. Further details can be found in [10]. For quick reference, we present a summary of important terms used in (hardware) pipeline scheduling in Appendix A.

[1]It may be possible to unfold the loop a number of times and thus obtain a shorter (better) **RecMII**. In this paper we do not consider any unfolding of the loop, though our techniques can be applied after such unfolding.

2.2 Need for Pipeline Theory

The application of software pipelining to loops executed on pipelines with structural hazards can benefit a great deal from an adaptation of classical pipeline theory. To illustrate this point, we first briefly review existing software pipelining methods that are capable of handling structural hazards [3, 14, 11, 9].

In general, these methods construct a Modulo Reservation Table (MRT) to keep track of the resources used by already scheduled instructions. The MRT contains **II** rows and one column for each resource. If an `FU` contains structural hazards, each pipeline stage must be included in the MRT. Given the MRT, all methods just cited proceed roughly as follows:

> **General Scheduling Algorithm:** (1) Schedule operations one at a time. (2) Use a priority function (e.g. level or slackness) to pick which operation to schedule next. (3) Schedule the high-priority operation so that the resulting partial schedule satisfies all resource and dependency constraints. (4) When an operation cannot be scheduled, selectively unschedule a number of operations and try again.

To the best of our knowledge, none of the the software pipelining approaches cited above makes any explicit use of classical pipeline theory. With the help of a few examples, we demonstrate that rectifying this omission can greatly improve the schedule produced.

Scheduling instructions based purely on the availability of resources is similar to using a **latency cycle** that is just **permissible**. Classical pipeline theory shows that such an approach does not always lead to efficient use of resources (e.g. `FU`'s). As an example consider the reservation table shown in Figure 1(a). The execution time of an instruction in this `FU` is *6* cycles. Suppose we are interested in constructing a schedule with **MII** = 9. To do this we extend the reservation table as shown in Figure 2(a). Notice from Figure 2(a) that a latency of 1 cycle is **permissible** [10]. However, as the Table indicates, once a second operation is started in the pipeline, no further operations may be started until the next iteration begins at time 9. (Notice that starting an operation at time step *6* will require the use of Stage 1 at time *10* (i.e. 6 cycles after the third `x` in the Table 2(a)). However, this will interfere with an operation (1*) from the second iteration, and hence is impermissible. More generally, this requirement — that no resource be used by a single iteration at times that are congruent modulo **II** — is known as the *modulo scheduling constraint* [8, 13].) Thus starting an operation after 1 cycle limits throughput to two operations every 9 cycles.

However classical pipeline theory reveals that initiations at time steps *0, 3, 6* are **permissible** and better utilize the pipeline—handling 3 operations every 9 cycles as depicted in Figure 2(b). The **General Scheduling Algorithm** (**GSA**) outlined above uses any permissible latency sequence, not necessarily an optimal one such as *0, 3, 6*. If the `FU` in this example

Stage	Time Steps					
	0	1	2	3	4	5
1	x		x		x	
2		x				x
3				x		

(a)

Stage	Time Steps					
	0	1	2	3	4	5
1	x		x			
2		x			x	
3				x		x

(b)

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

(c)

Figure 1: Example Reservation Tables

is a critical resource, the **GSA** may have to make a large number of retries or may even fail to produce a valid schedule even though one exists at the given **II**. Software pipelining should calculate and make use of optimal latency sequences so as to produce better schedules and produce them faster (i.e. in less compile time).

Stage	Time Steps											
	0	1	2	3	4	5	6	7	8	9	10	11
1	x	1	x	1	x	1				x*	1*	x*
2		x	1			x	1				x*	1*
3				x	1							

(a) *0,1* Initiation. ($*$ ops from 2nd Iteration)

Stage	Time Steps											
	0	1	2	3	4	5	6	7	8	9	10	11
1	x		x	3	x	3	6	3	6	x*	6	x*
2		x			3	x		6	3		x*	6
3				x			3				6	

(b) *0,3,6* Initiation. ($*$ ops from 2nd Iteration)

Figure 2: Extended Reservation Tables with Multiple Iterations

Since individual stages in an `FU` are modeled as separate resources, the number of resources for a modern VLIW architecture may be more than 50, while the number of `FU`s in the architecture may be less than 10! Using the **GSA** on the resulting large MRT can increase the scheduling time. Use of pipeline theory permits use of a much smaller MRT: one of the best possible **latency cycles** can be used directly by the scheduler, thereby allowing a function unit to be modeled as a single resource instead of a number of pipeline stages.

Use of classical pipeline theory rules out several possible initiations which lead to conflicts in one or more stages. As an example, consider the reservation table shown in Figure 1(b). The most straightforward extension of earlier software pipelining approaches to handle hazards might schedule an instruction in this `FU` at time step t, and try to place subsequent initiations at time steps $t+1$, and $t+4$, only to find out that the last initiation causes a collision in a later stage of the pipeline. However, using the state diagram [10] of possible initiations immediately rules out any initiations with a latency 3.

Lastly, the **ResMII** obtained from Equation 2 is only a loose bound. For example, in the reservation table of Figure 1(c), the maximum usage of any stage is only 2. Thus the **ResMII** for this `FU`, as computed using Equation 2, will be $\lceil \frac{N_r * 2}{F_r} \rceil$. However, two initiations need to be separated by at least 4 cycles, as indicated by the state diagram of the `FU`. Hence the actual **ResMII** for this `FU` is $\lceil \frac{N_r * 4}{F_r} \rceil$, roughly twice as large as the bound given by Equation 2.

This section has clearly shown the advantage of using classical pipeline theory in existing software pipelining methods to improve both schedule quality and scheduling speed.

2.3 The need for Co-Scheduling

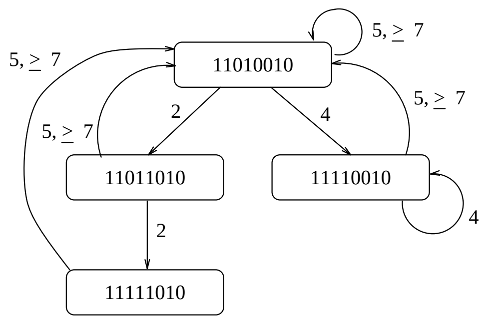
Can one directly use classical pipeline theory in the context of software pipelining? We answer this question in this subsection which motivates our proposed *Co-Scheduling* approach.

A VLIW architecture contains different types of function units, e.g. **Integer**, **Floating Point**, and **Load/Store**. Each `FU` type may have a different reservation table and therefore the **latency cycles** which achieve maximum utilization of the stages of the (hardware) pipeline may have different periods. (The sum of the latency values in the **latency cycle** is referred to as the *period* of the **latency cycle**. To distinguish between this *period* (of the hardware pipeline) from the *period* of the software pipelined schedule, we refer to latter as *initiation interval*, **II**, or **MII** depending on the context. **The term "*period*" henceforth refers to the hardware pipeline.**) These periods may not be related to the **II** of software pipelining. As a consequence some of the legal **latency cycles** predicted by the classical pipeline theory may violate the modulo scheduling constraint for the given **II**. We illustrate this with the help of another example.

Consider the reservation table and its state diagram shown in Figure 3. Assume **II** = 8. In the state diagram the **latency cycle** $(2, 2, 5)$ has *Minimum Average Latency*, $MAL = \frac{2+2+5}{3} = 3$. However, commencement of instructions at time steps *2, 4* and *9* will violates the modulo scheduling constraint at time step *10*. In fact starting any two instructions 5 cycles apart violates the modulo scheduling constraint and hence causes a collision in a modified reservation table (with 8 columns). This is not unexpected since the state diagram is obtained

Stage	Time Steps							
	0	1	2	3	4	5	6	7
1	x						x	
2		x			x			
3			x	x				
4						x		

(a) Reservation Table



(b) State Diagram

Figure 3: An Example Reservation Table and its State Diagram

for a reservation table with 7 columns and was derived without a "wrap-round" resource usage in mind. Further the classical pipeline theory [13, 10] *does* indicate that *5* is an impermissible latency for any cycle with period 9, since *5* is the complement of the **forbidden latency** *3* in the modulo space with **II** = 8. However, the focus in these works [13, 10] is on how to reconfigure the hardware pipelines for a *given latency cycle* [2]. Whereas here we are interested in *finding the "best" latency cycle* given the initiation interval **II**.

The latency cycle (2,5) has the next best average latency (3.5). However it also violates the modulo scheduling constraint and hence cannot be considered when **II** = 8. However, the (self) cycle from state `11110010` to itself has an average latency of 4 and does not violate modulo scheduling constraint. In fact this is the "best" latency cycle for **II** = 8. As this example shows, the state diagram constructed using the classical pipeline theory does not account for the software pipelining **II**. As a consequence the modulo scheduling constraint may be violated by some **latency cycles** identified as legal by the state diagram. In the following section we show how to extend the classical pipeline theory to achieve the simultaneous scheduling of hardware and software pipelines.

3 Cyclic Pipelines

In this section we revisit classical pipeline theory in the context of software pipelining. To differentiate our approach from the classical pipeline theory, we refer to our pipelines as *cyclic* pipelines, since, with our assumption of fixed `FU` assignment, the hardware pipeline is scheduled at the **II** of the software pipeline. We define the terms *reservation table*, *forbidden latency*,

[2]This approach will also be useful in the context of *co-scheduling* when the hardware pipelines are reconfigured to (further) improve the initiation interval of the software pipeline schedule. We discuss this further in Section 7.

collision vector, and *state diagram* as they apply to cyclic pipelines. We then develop the theory behind cyclic pipelines which in turn forms the basis for our co-scheduling framework.

3.1 Preliminaries: Cyclic Reservation Table and Related Concepts

In this paper we restrict our attention to *single-function* pipelines whose resource usage pattern can be described by a single reservation table. The reservation table of a hardware pipeline is represented by an $m_r \times l_r$ reservation table where m_r is the number of stages in the pipeline and l_r is the execution time (latency) of an operation executing on `FU` r. We use the symbol $d_{r,s}$ to denote the number of cycles stage s in the pipeline is used. Define $d_{max}(r)$ as the maximum of $d_{r,s}$ over all stages in the pipeline.

Lemma 3.1 *Under fixed* `FU` *assignment, if* `FU` *type r is used by the schedule, the initiation interval of a software pipelined schedule* $\mathbf{II} \geq d_{max}(r)$.

Proof: Follows from the fact that different instances of an instruction need to be assigned to the same `FU`. □

With cyclic pipelines, each instruction must be initiated in the pipeline every **II** cycles. Therefore it would be appropriate to use a reservation table with **II** (rather than l_r) columns. Notice that Lemma 3.1 only requires **II** to be greater than the $d_{max}(r)$ value of every `FU` type r used in the schedule. However, the relationship between l_r and **II** could be (1) $\mathbf{II} > l_r$, (2) $l_r > \mathbf{II}$, or (3) $l_r = \mathbf{II}$. In case (1), the reservation table may be extended to **II** columns (with the additional columns all empty). In case (2) the reservation table may be folded. Thus for stage s an `X` mark at time step t in the original reservation table appears at time step t `mod` **II** in the folded reservation table. In case (3), nothing need be changed. We call the resulting reservation table the **cyclic reservation table (CRT)**. An entry in the *CRT* is denoted by $CRT_r[s,t]$.

With the folding required in case (2), multiple `X` marks may be placed in the same column of the CRT. However, preventing operations from overlapping at times that are congruent modulo **II** is exactly what the *modulo scheduling constraint* tries to avoid. Clearly fixed `FU` assignment requires that the modulo scheduling constraint be satisfied. One way to do this and avoid the violation possible in case (2) is to modify the hardware so as to delay all but one of the operations mapping to the same time t [3].

Lemma 3.2 *As long as there are no inter-row (i.e. inter-stage) dependencies (in the reservation table), it is always possible to satisfy the modulo scheduling constraint by introducing appropriate delays in the reservation table.*

[3] As this would need to be done on a loop by loop basis, we hope that hardware designers will consider making such a capability available in the instruction set of future processors.

Proof: The proof is by construction. Every violating `X` mark in the reservation table is delayed by the *minimal* amount that satisfies the modulo scheduling constraint. Since to satisfy Lemma 3.1, the total number of `X` marks is less than or equal to **II**, it is always possible to find a delay such that the `X` may be placed in a heretofore empty column (time). Since there are no inter-row dependencies in the reservation table, this method of introducing delays can work with one stage of the pipeline at a time. □

A consequence of introducing delays to satisfy the modulo scheduling constraint is that the execution time of an operation may be increased. This in turn can affect **RecMII** and hence **MII**. However, for purposes of this Section, we assume that even after the introduction of delays in the pipeline, **II** $\geq$ **MII**. In Section 4.2, we discuss a simple algorithm that eventually arrives at the smallest **II** at which the modulo scheduling constraints as well as the resource and dependency constraints are satisfied.

Figure 3.1(a) shows the CRT of the reservation table shown in Fig. 1(a), when **II** $= 9$.

Observation 3.1 *The number of* `X` *marks in row s of a reservation table equals that in row s of the corresponding* **CRT***.*

Next we define several terms.

Definition 3.1 (Cyclic Forbidden Latency) *A latency $f \leq$* **II** *is said be a* **cyclic forbidden latency** *if there exists at least one row in the CRT where two entries (*`X` *marks) are separated by f columns (considering the wrap-around of columns). More precisely, there exists a stage s such that both $CRT[s,t]$ and $CRT[s,(t+f) \texttt{ mod } \mathbf{II}]$, contain an* `X` *mark.*

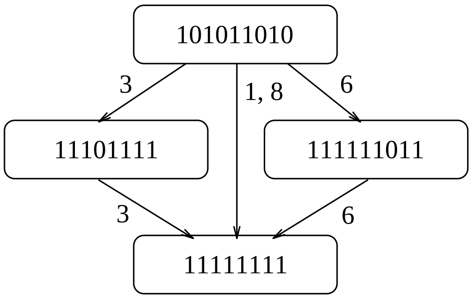
It can be easily seen that in a cyclic pipeline latency values f greater than **II** are equivalent to f `mod` **II**. Hence, for cyclic pipelines, we will only consider latency values less than **II**. The set of all cyclic forbidden latencies is referred to as the cyclic forbidden latency set. The latency values 2 and 4 are forbidden in the CRT in Fig. 3.1(a) as there are entries in the first row at time steps 0, 2, and 4. Further, latency 5 is also forbidden since the distance between the entries in columns 4 and 0 (with the columns wrapped around) in the first row is 5. The cyclic forbidden latency set is $\{0, 2, 4, 5, 7\}$.

Definition 3.2 (Cyclic Permissible Latency) *A latency $f \leq$* **II** *is said to be a* **cyclic permissible latency** *if f is not in the cyclic forbidden latency set.*

For the CRT in Fig. 3.1(a), the cyclic permissible latencies are 1, 3, 6, and 8. From the above definitions it can be easily observed that:

Stage	Time Steps								
	0	1	2	3	4	5	6	7	8
1	x		x		x				
2		x				x			
3				x					

(a) Cyclic Reservation Table



(b) State Diagram for the **CRT**

Observation 3.2 *The cyclic permissible latency set is the* complement *of cyclic forbidden latency set.*

The next Lemma follows immediately from Definitions 3.1 and 3.2. [4]

Lemma 3.3 *If $c1$ is a forbidden latency, then $c2 = \mathbf{II} - c1$ is also a forbidden latency.*

Proof: If $c1$ is forbidden then there must exist a stage s in the CRT where two entries are separated by $c1$ columns. Let the entries be at columns t and $(t + c1)$ $\texttt{mod}$ **II**. By Definition 3.1, the gap from $t + c1$ $\texttt{mod}$ **II** to t is also forbidden. The size of that gap is $\{t - (t + c1 \texttt{ mod } \mathbf{II})\} \texttt{ mod } \mathbf{II}) = \mathbf{II} - c1$. □

Lemma 3.4 *If $c1$ and $c2$ are numbers such that $c1 + c2 = \mathbf{II}$, then either both $c1$ and $c2$ are cyclic permissible latencies or both are cyclic forbidden latencies for a CRT with* **II** *columns.*

Proof: There are two cases to consider.

Case 1 ($c1$ is *forbidden.*) Since $c2 = \mathbf{II} - c1$, by Lemma 3.3 $c2$ is also forbidden.

Case 2 ($c1$ is *permissible.*) If $c1$ is permissible, then we need to prove that $c2$ is also permissible. Assume $c2$ is forbidden. But by Case 1, this implies that $c1$ is also forbidden which contradicts our assumption. Hence $c2$ must be permissible. □

[4] Lemma 3.4 somewhat similar to Lemma 3-6 in [10] (page 97). This is to be expected since Lemma 3-6 in [10] deals with a latency cycle for hardware pipelines with a period p.

3.2 State Diagram for Cyclic Reservation Tables

Our interest in co-scheduling is to obtain latency sequences that maximize the number of initiations in **II** cycles. In order to derive this, we construct the *state diagram* for a CRT, in much the way as is done in classical pipeline theory. We use the term **Modulo-Scheduled State Diagram (MS-state diagram)** to distinguish the state diagram for a CRT from the state diagram of a Reservation Table. The initial state in the MS-state diagram represents an initiation at time step 0. We are interested in finding how many more initiations are possible in this pipeline, and at what latencies. We define *cyclic collision vector* to represent the state after a particular initiation.

Definition 3.3 (Cyclic Collision Vector) *A Cyclic Collision vector is a binary vector of length* **II***, with the bits numbered from* 0 *to* $\mathbf{II} - 1$*. If* f *is forbidden in the current state then the* f*-th bit in the cyclic collision vector is 1. Otherwise it is 0.*

For the CRT in Figure 3.1(a), the the initial cyclic collision vector is `101011010`. The construction of the MS-state diagram proceeds as follows.

Procedure 1 Construction of State Diagram:

Step 1 Start with the initial cyclic collision vector.

Step 2 For each permissible latency p in the current state, i.e. all bits p in the collision vector whose value is 0, derive a subsequent state as follows.

(a) `Rotate-left` the collision vector by p bits.

(b) Logically `OR` the resulting vector with the initial cyclic collision vector. The resulting collision vector is a subsequent state.

(c) Place an arc with value p from the previous state to the new state.

Observe that there is a very close resemblance of **Procedure 1** to the state diagram construction in the classical pipeline theory. The main difference is that in Step 2(a) of **Procedure 1** a `Rotate-left` operation is performed rather than a `shift-left` operation. The intuitive reason for this is that in *co-scheduling*, every instruction is reissued every **II** cycles— once each iteration. Thus an instruction scheduled p cycles later will not only have to share resources for the first $\mathbf{II} - p$ time steps, with instructions scheduled so far in this software pipeline cycle (or any previous software pipeline cycle), but also with the instructions initiated in the first p cycles of the next software pipelining cycle.

Theorem 3.1 *The collision vector of every state* S *in the MS-state diagram derived according to* **Procedure 1** *represents all permissible (and forbidden) latencies in that state, taking into account all initiations made so far to reach the state* S*.*

. The proof of this theorem is presented in Appendix B. It essentially follows from the argument given for why `rotate-left` operation is performed in the construction.

The MS-state diagram for the CRT in Figure 3.1(a) is shown in Figure 3.1(b). In drawing the MS-state diagram we have avoided the repetition of *identical* states to make the diagram concise. Further, multiple arcs from state S_i to S_j are represented by means of a single arc with multiple latency values, e.g. in Figure 3.1(b), the state `111111111` can be reached from the initial state with a latency value of either 1 or 8.

A path in the MS-state diagram is a set of latency values, one associated with each arc along a path, from the initial state to the current state. For example, there is a path with latency values $\{3, 3\}$ from `101011010` to `111111111` in Figure 3.1(b). Likewise there is a path with latency values $\{6, 6\}$ and with latency values $\{1\}$ and $\{8\}$. The MS-state diagram indicates that initiations corresponding to these latency values are possible after the initial state. Since the initial state itself represents as initiation at time step 0, we represent all latency sequences with an initial 0 value, as in $\{0,\ 3,\ 3\}$ or $\{0,\ 8\}$. The *length* of a path is the number of states encountered on the path. It is equal to the cardinality of the latency sequence. Further, the longest path corresponds to the maximum number of initiations possible in a pipeline within a software pipeline cycle. For the MS-state diagram shown in Figure 3.1(b), the maximum number of initiations is 3 and the corresponding latency sequences are $\{0, 3, 3\}$ and $\{0,6,6\}$.

3.3 Analyzing the State Diagrams

Next we investigate certain properties of the MS-state diagram which relate to the number of initiations possible in pipeline during a software pipeline cycle. While the analysis of state diagrams in classical pipeline theory [10] concentrates on obtaining "best" initiation sequences of hardware pipelines (without any consideration of software pipeline schedules), the analysis of MS-state diagrams reveals "best" initiation sequences of hardware pipelines for the given **II**. Hence the analysis developed in this section is of direct use in the co-scheduling framework.

Lemma 3.5 *If there is an arc from state S_i to S_j in the MS-state diagram, then the number of `1`'s in the collision vector of state S_i is* strictly *greater than that in state S_j.*

Proof: The `rotate-left` operation used in the construction of the MS-state diagram preserves the number of `1`'s. Further, the `OR` operation guarantees that the number of `1`'s in state S_i is greater than or equal to the number of `1`'s in state S_j. However, we need to prove the strictly greater than relation. Let p be the latency associated with the arc from S_i to Sj. Thus the p-bit in the collision vector of S_i must be `0`. The `rotate-left` operation (by p bits) aligns the p-th bit of S_i with the 0-th bit of the initial cyclic collision vector. Since 0 is a forbidden latency (in all single-function pipelines), the p-th bit of S_i is `OR`-ed with the 0-th bit of the initial cyclic collision vector which results in `1`. Thus after the `rotate-left` and `OR`, at least one of the `0` bits of S_i is made `1` in state S_j. Thus the number of `1`'s in the collision vector in state S_i is strictly greater than that in S_j. $\Box$.

Lemma 3.6 *There are no directed cycles in the MS-state diagram.*

Proof: Suppose there exists a directed cycle in the MS-state diagram involving some states S_1, S_2, ..., S_m. That is there exists a path $S_1 \rightarrow S_2 \rightarrow \ldots \rightarrow S_m \rightarrow S_1$. Then by Lemma 3.5,

$$\text{No. of 1's in } S_1 < \text{No. of 1's in } S_2 < \ldots < \text{No. of 1's in } S_m < \text{No. of 1's in } S_1$$

which is a contradiction. Hence the lemma. □

Lemma 3.7 *The* final *state of the state diagram of any CRT with* **II** *columns contains* **II** `1`*'s.*

Proof: The proof of this lemma is in 3 steps:

1. By Theorem 3.1 the collision vector of any state represents the permissible and forbidden latencies at that state. Thus, if the p-th bit of the collision vector is `0` then the latency p is permissible and hence an initiation can be made p cycles later, leading to a new state.

2. Lemma 3.6 ensures that there are no cycles in the state diagram.

3. From Lemma 3.5 the number of 1's in the collision vector is at least 1 more than that in collision vector of the previous (predecessor) state.

Thus (1) can be applied repeatedly until there are no `0`'s in the collision vector. □

The following theorem establishes an upper bound for the maximum number of initiations in a pipeline.

Theorem 3.2 *The maximum number of initiations made in a pipeline during a software pipeline cycle is*

$$\min\left((m+1), \left\lceil \frac{\mathbf{II}}{d_{max}} \right\rceil\right)$$

where m is the number of `0`*'s in the initial cyclic collision vector and d_{max} is the maximum number of* `X` *marks in any row in the reservation table.*

In order to the prove the Theorem, first we establish two lemmas.

Lemma 3.8 *If the initial cyclic collision vector of the CRT contains m* `0`*'s then any path in the MS-state diagram will have a length of at most $(m+1)$.*

Proof: This follows from Lemma 3.5 and Lemma 3.7. □

Notice that the number of 0's in the initial collision vector corresponds to the cardinality of the cyclic permissible latency set. We can also establish another upper bound for the maximum number of initiations.

Lemma 3.9 *If the maximum number of* `X` *marks in any row in a CRT is* d_{max} *then the maximum number of initiations is bounded above by* $\left\lceil \frac{\mathbf{II}}{d_{max}} \right\rceil$.

Proof: From the classical pipeline theory, we know that the Minimum Average Latency (MAL) is greater than or equal to d_{max}. The reciprocal of MAL is the maximum number of initiations per unit time. Thus,

$$MAL \geq d_{max} \qquad \text{or} \qquad \frac{1}{MAL} \leq \frac{1}{d_{max}} \tag{4}$$

The number of initiations made in a software pipeline cycle, with an initiation interval **II**, is at most $\frac{\mathbf{II}}{MAL}$. From Equation 4,

$$\text{Maximum number of initiations} = \frac{\mathbf{II}}{MAL} \leq \frac{\mathbf{II}}{d_{max}}$$

Since the left hand side of this equation is an integer,

$$\text{Maximum number of initiations} \leq \left\lceil \frac{\mathbf{II}}{d_{max}} \right\rceil \; \square$$

Proof: (of Theorem 3.2) Follows from Lemma 3.8 and Lemma 3.9. □

In the MS-state diagram shown in Figure 3.1(b), there are four `0`'s in the initial cyclic collision vector. However the length of the longest path is 3. This corresponds to at most 3 initiations, at time steps 0, 3 and 6. Further, the CRT (Figure 3.1) for the MS-state diagram contains at most three `X` marks in a row. Hence the maximum number of initiations is at most $\lceil \frac{9}{3} \rceil = 3$. In the MS-state diagram (Figure 3.1(b)) we observe that the length of the longest path is 2 which corresponds to the maximum number of initiations, namely 3.

4 Co-Scheduling the Hardware and Software Pipelines

In this section we detail how our Co-scheduling approach generates schedules for `FU`s with structural hazards. The next Section provides an overview of Co-Scheduling, while Sections 4.2 and 4.3 detail the two major components of Co-Scheduling. Section 4.4 then tries to put everything in perspective.

4.1 Overview of Co-Scheduling

Figure 4 outlines our approach. We begin by calculating **MII** from Equation 3. Using the framework just outlined in Section 3, we then attempt to find (one of) the longest *latency sequence(s)*, i.e. we try find at what times operations should be introduced into the pipeline so as to keep it maximally utilized. As the example in Section 2.2 showed, the **MII** from Equation 3 is sometimes too low. In the next Section, we detail how to handle this situation.

Once a *latency* sequence providing maximum pipeline utilization is found, software pipelining must be performed. To do this we introduce **SCS**, a slackness based co-scheduling algorithm based on Huff's *Slack Scheduling* [9]. The key distinction between **SCS** and Huff's approach is that **SCS** makes use of the *latency sequence* just calculated. Instead of placing operations at any *permissible* time, **SCS** places them only at times given by the *latency sequence*. In this way, **SCS** is able to deal with pipeline hazards in a much more efficient way than the original *Slack Scheduling*.

If **SCS** fails to find a schedule or fails to find one within a reasonable period of time, **II** is incremented, and the whole process is tried again. Once a schedule is found, the corresponding pipeline delays must be noted. That is the prologue code to the loop must contain instructions telling the hardware pipelines to place delays (buffers) at the necessary locations.

Note that if there are multiple optimal *latency sequences*, only one is tried. Thus our approach may miss a possible optimal schedule in some instances. We plan to investigate this possibility in future work.

4.2 Determining the Minimum II

We use the following notation throughout this section. Our aim is to construct a software pipelined schedule for a loop L to run on an architecture with h different types of `FU`s (e.g. **Adder**, **Multiplier**, etc.). The usage of resources (pipeline stages) in `FU` type r is specified by a *single* reservation table [5] RT_r. The execution time of an instruction that executes on `FU` type r is same as the length of the reservation table l_r. Further, each `FU` type r has F_r pipelines. The loop L has N_r instructions that are executed on the F_r pipelines of `FU` type r.

As mentioned in Section 2.1. the Minimum **II**, **MII**, is the maximum of **ResMII** and **RecMII**. However, the example in Section 2.2 indicates that the **ResMII** bound is loose. Further Lemma 3.1 requires any initiation interval **II** must be greater than or equal to $d_{max}(r)$, the maximum number of `X` marks in any row of the reservation table of every `FU` type r used by the schedule. Lastly, for an initiation interval **II**, the CRT has length **II** and must satisfy

[5] Even though we say that this work concentrates on *single function pipelines*, it is this property — supporting a single reservation table — that is needed. For example, it is common to have the same resource usage pattern for two operations (e.g. **FP Add** and **FP Subtract**). and hence these operations can be executed on a single `FU` type. The co-scheduling framework developed in this paper is applicable in those cases as well.

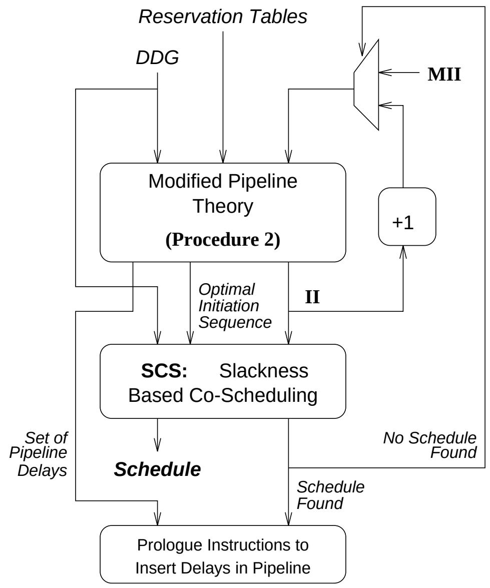


Figure 4: Outline of *Slack Co-Scheduling* Algorithm.

the modulo scheduling constraint. This may introduce delays in the CRT which, in turn, can increase the execution time of instructions. As a consequence **RecMII** can be affected. Starting from the **MII** value obtained from Equation 3 (in Section 2.1), we use the following iterative procedure to determine the smallest **II**.

Procedure 2 Find_Minimum_II (**II**)

Step 1 $\mathbf{II} = \max\{\mathbf{II}, \max_r[d_{max}(r)]\}$, where $d_{max}(r)$ is the maximum number of `X` marks in any row of RT_r.

Step 2 Repeat Steps 2.1 to 2.6 until a valid **II** satisfying resource, recurrence and modulo scheduling constraints is found

Step 2.1 For each `FU` type r do

Step 2.1.1 If $l_r = \mathbf{II}$ then CRT_r is same as RT_r. Otherwise construct CRT_r as follows.

Step 2.1.2 If $l_r < \mathbf{II}$ then the CRT is constructed by adding $(II - l_r)$ empty columns.

Step 2.1.3 If $l_r > \mathbf{II}$ then, every `X` mark in $RT_r[s,t]$ is placed in $CRT_r[s, t \bmod \mathbf{II}]$. If any $RT_r[s,t]$ violates the modulo scheduling constraint, then that `X` mark is put in the next available column in CRT_r (considering wrap-around of columns).

Step 2.1.4 If the introduction of delays has increased the execution time, then l_r is set appropriately.

Step 2.2 **RecMII** is calculated with the new values of l_r, since the introduction of delays might have increased the execution time l_r.

Step 2.3 If the new **RecMII** > **II**, increment **II** by 1 and go back to Step 2.1.

Step 2.4 Else, derive the MS-state diagram for the CRTs, CRT_1 to CRT_h. Let Max_r represent the maximum number of initiations possible in `FU` type r and L_r be a corresponding latency sequence that achieves the maximum initiations.

Step 2.5 If, for each `FU` type r, $F_r \geq \left\lceil \frac{N_r}{Max_r} \right\rceil$ go to Step 3;

Step 2.6 Else, increment **II** by 1 and go back to Step 2.1.

Step 3 End.

Step 1 computes an initial estimate for **II** based on Equation 3 and Lemma 3.1. Step 2.1 constructs the CRT for each `FU` type. In this process, delays may be introduced in the reservation table to satisfy the modulo scheduling constraint. If the delays have increased the execution time of an instruction, then it is set to its new value in Step 2.1.4. For the new values of l_r, the **RecMII** for the loop is computed again. If **RecMII** is greater than the current **II** value, then **II** does not obey recurrence constraints after the introduction of delays (to satisfy modulo scheduling constraint). We will establish in Theorem 4.1 that no valid schedule can exist for this **II** since either modulo scheduling constraints or recurrence constraints are violated. Hence **II** is incremented by 1 and Steps 2.1 to 2.6 are repeated. If the new **RecMII** is less than or equal to the current **II**, then we proceed to derive the MS-state diagram (Step 2.4) and the maximum number of initiations possible in each `FU` in a software pipeline cycle. Step 2.5 checks if **II** is a tight bound. If not we increment **II** by 1 and start from Step 2.1 again. Thus when **Procedure 2** terminates, the **II** value satisfies, dependency, resource and modulo scheduling constraints. Using the algorithm described in the following subsection, we attempt to construct a software pipelined schedule for this **II**. As will be discussed in Section 5, our experiments show that the introduction of delays (to satisfy the modulo scheduling constraint) never increased **II**.

We now establish that the **II** obtained from **Procedure 2** is the minimum **II**. The following remarks clarify the conditions under which such this claim is true.

1. Our schedules are restricted to fixed `FU` assignment.

2. We do not consider any unrolling of loops. Unrolling allows schedules which have an initiation interval that is [0,1) less than the **MinII** we obtain.

3. The introduction of delays is used *only* to satisfy the modulo scheduling constraint. (Section 7 has a discussion on the use of delays to improve the maximum number of initiations in a pipeline.)

Theorem 4.1 *There does not exist a resource-constrained schedule with an initiation interval* $\mathbf{II}' < \mathbf{II}$ *where* **II** *is the minimum* **II** *obtained from* **Procedure 2**.

Proof: The proof of this theorem follows by observing that:

1. The initial **II** given by **Step 1** is the smallest **II** under fixed `FU` assignment.
2. The introduction of delays in **Step 2.1.3** in the reservation table is *minimal* and is required for satisfying the modulo scheduling constraint. Hence the increase in l_r, if any, is also minimal.
3. Once the CRT is constructed, the recurrence constraints must be satisfied for the (new) l_r values as given by **Step 2.1.4. Step 2.3** ensures this.
4. It is easy to see that **Step 2.5** must be satisfied to construct a feasible schedule with fixed `FU` assignment.
5. Clearly, **Procedure 2** tries each **II** value beginning with the initial **II**. □

4.3 SCS Algorithm

As mentioned in the Introduction, our Co-scheduling framework uses a *Modulo Initiation Table (MIT)* to represent the modulo initiation time of different instructions. The CRT is used to keep track of the resource (pipeline stage) usage. Notice that our CRT, constructed as discussed in the previous section, does satisfy the modulo scheduling constraint, even in the presence of structural hazards.

After computing an **II** and a corresponding optimal *latency sequence* as detailed in **Procedure 2**, **SCS**, our slackness based co-scheduling algorithm is used to find a schedule fitting **II** and the *latency sequence*. The basic notion of Huff's original Slack Scheduling [9] was to schedule nodes in increasing order of their *slackness*: the difference between the earliest time at which a node may be scheduled and the latest. *Slack* is a dynamic measure and is updated after each node is scheduled. These points remain in **SCS**.

The difference lies in how a time is chosen within the slack range. The original Slack Scheduling permitted nodes to be placed anywhere in their slack range [6] **SCS**, on the other hand, allows nodes to be placed only at points given in the *latency sequence*, even if this means avoiding some times that would yield a legal partial schedule. In this way, **SCS** takes a more global view of when instructions should be scheduled and avoids getting trapped at greedy local maxima, the way the original can [7].

To clarify matters, we illustrate **SCS** through an example. Assume:

[6] Heuristics were used to guide exactly where.

[7] In fairness, we note that the original was applied only to function units with clean pipelines and function units (`divide`) with no pipelining.

- **II** = 5.
- One type of function unit.
- One copy of it.
- Latency Sequence from **Procedure 2**: **0, 3**.
- One operation already placed at time 1 (`mod` **II**).
- New operation must be placed at time 3 or 4 to obey data dependence constraints.

The *MIT* or *Modulo Initiation Table* for the function unit prior to scheduling the new operation is depicted below:

```
Time:  0   1   2   3   4
     +---+---+---+---+---+
     |   | X |   |   |   |
     +---+---+---+---+---+
```

Unlike the CRT, the *MIT* indicates only when operations commence [8]. Since the first operation has already been placed at time 1, this operation can be placed only at times matching the latency sequence and that have not already been used. Since time `1+0=1` has already been used, the only other legal value is `1+3=4`. Thus time 4 is chosen, even though time 3 may also have produced a legal partial schedule. Recall that SCS, at least the current implementation, works with only one latency sequence[9]. Thus a latency of 2 may also be legal, but SCS is not aware of it.

```
Time:  0   1   2   3   4
     +---+---+---+---+---+
     |   | X |   |   | X |
     +---+---+---+---+---+
```

We made one other modification from Huff's slack scheduling to account for the fact that the *slackness* of an operation depends not only upon the data dependency constraints, but also upon the placement of previous operations. In the example, the Huff *slack* value was 2, i.e. the operation meets dependence constraints if executed at times 3 or 4. However, the only place the second operation could go was time 4, meaning that in some sense, it really had a *slack* of only 1. In Section 5 we will compare the performance of the SCS algorithm based on these two *slack* measures.

[8] We use the *MIT* for simplicity of explanation. The actual implementation uses an augmented *CRT* for efficiency reasons.

[9] A discussion on using a single (best) latency sequence in the SCS algorithm is presented in Section 4.4.

We chose to use *Slack Co-Scheduling* because of the good performance achieved by the original Slack Scheduling [9]. However, modified versions of other modulo scheduling techniques citeLam88,GasperoniSch93,WangEtAl92,Rau94 using a fixed **II** could have been used instead. The main novelty of our approach lies in **Procedure 2**. The fact that it can work with variants of many other approaches indicates its versatility.

4.4 Remarks and Discussion

A few remarks on the construction of the MS-state diagram are in order. The MS-state diagram can become very large even for a reasonable values of **MII** (e.g. 20), especially when the reservation table has only a few forbidden latencies. In those cases it is very likely that the maximum number of initiations will be equal to the upper bound specified by Equation 3.2. Further, a path corresponding to this maximum number of initiations may be obtained quite early in the construction, especially if the algorithm used in the state diagram construction is depth-first. Thus once a (permissible) latency sequence yielding the maximum number of initiations is found, the construction of the remaining part of the MS-state diagram can be aborted. Notice that our SCS algorithm only uses (one of) the best latency sequence(s) to construct a schedule.

Another approach to pruning large MS-state diagrams is to limit the number of states examined to a maximum number of values (e.g. 10,000). Though an optimal latency sequence cannot be guaranteed with such an approach, empirical results, presented in Section 5, establish that latency sequences which are at most 1 away from the maximum number of initiations can be obtained.

As discussed earlier, our SCS algorithm works only with the best latency sequence. In certain cases, it may be desirable to consider sequences that do not result in maximum initiations, yet obtain a resource constrained schedule at **II**. The use of a single latency sequence in the SCS algorithm may also effect:

1. The existence of a resource-constrained schedule for a given **II**. This is true, even if the choice is made only among latency sequences that result in maximum initiations.

2. The quality of the schedule in terms of the number of registers used by the schedule.

It is possible, in principle, to extend our SCS algorithm to consider multiple latency sequences and use one that fits best for the given loop and **II** to construct the software pipelined schedule. We leave the study of considering multiple latency sequences to a future work.

II – MII	Huff's Slack		Modified Slack	
	# of Cases	%-age	# of Cases	%-age
0	615	61	617	61
1	124	12	129	13
2	78	8	87	9
3	34	3	48	5
4	35	3	38	4
≥ 5	63	6	74	7
No Sched	64	6	20	2

Table 1: Difference between **II** achieved and **MII**.

5 Experimental Results

To evaluate Co-scheduling we implemented it and tested it on 1013 loops taken from a variety of benchmarks: `specfp92`, `specint92`, `livermore`, `linpack`, and the `NAS kernels`. All of these loops contain fewer than 64 operations, with a median of 7 and a mean of 12. For the experiments, we considered an architecture with 2 **Integer Units**, one each of the remaining units: **Load**, **Store**, **FP Add/Subtract**, **FP Multiply** and **FP Divide**. Our reservation tables use the latencies of units of the PowerPC-604, but we have added several structural hazards so as to better exercise our approach. The actual reservation tables are shown in Appendix C.

We restricted our implementation to take a maximum of 1 minute to construct the schedule. We also limited the size of the *MS-state diagram* generated to 2000 collision vectors. For most loops, neither of these limitations had any effect. In 993 out of 1013 loops (98%), a schedule was constructed within this time limit. To measure the performance of Co-Scheduling, we look at several statistics using both Huff's original *slack* measure and our modified *slack* measure. First Table 1 details how well the **II** compares to the lower bound **MII**. As can be seen, for the majority of loops (actually 61%), we achieve **II** = **MII**. As the Table also indicates, our new measure for slack considerably improves performance allowing Co-scheduling to find schedules for 44 loops in which the original measure yielded no schedule.

Additional statistics characterizing the loops and resulting schedules, obtained using our modified *slack* measure, are given in Table 2. The median time to schedule a loop was 0.29 seconds and the mean was 1.25 seconds on a Sparc 10/30. Scheduling based on Huff's *slack* measure was slightly faster with a median of 0.23 seconds and a mean of 1.16 seconds. However, this minor speed improvement comes at the price of signficantly worse schedules as indicated in Table 1.

The median **II** was 8 and the mean **II** was 14.6. 96.1% of the loops required no more than 32 registers. Thus in a large number of cases, the schedule produced by Co-scheduling

Measurement	Minimum	Frequency of Minimum (%)	Median	Mean	Maximum
No. of Nodes	1	0.1	7	12	60
II	1	11	8	15	237
II – MII	0	61	0	1.56	61
II/MII	1.00	61	1.00	1.08	1.95
Registers	1	0.1	6	17	2812
Time (sec)	0.12	0.2	0.29	1.25	33.04

Table 2: Comparison of **II** achieved to **MII**.

does not require any further (register) spill code. 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.

From our experiments, we observed that neither the introduction of delays (**Step 2.1.3** in **Procedure 2**) nor the bound (in **Step 2.5**) increased **II**. While we anticipated that the introduction of delays would not significantly affect **II**, we were a bit surprised to find that the bound (from Equation 3) is quite tight. Both are likely due to the particular reservation tables we chose and the fact that we assumed no dependence between different stages of the reservation table. We plan further investigation of these effects.

Finally, to study the effects of pruning MS-State diagrams on the longest path achieved, we looked at the *maximum number of initiations* for **II** from 1 to 64 for all the reservation tables. In order to get a better idea of the search space, we let the MS-state diagram grow to 10,000 (instead of 2000) collision vectors before taking the longest path. (However, it turned out the additional vectors yielded improvement in only one case.) For 54 values of **II**, we found a set of *permissible latencies* with the *maximum number of initiations* for all reservation tables. For the remaining 10 values of **II**, we found a set of permissible latencies with *maximum initiations* for all but one reservation table (and that reservation table had a set with *maximum initiations*−1).

In summary, the experiments establish that *Co-Scheduling* is indeed quite successful: it generates a schedule for pipelines with structural hazards in more than 98% of the test cases. There is no direct data (due to the unavailability of comparative results from other modulo scheduling methods) to support the claims made in **Observation 1** (that use of optimal latency sequence yields better schedules and yields them faster). However, the high quality of the results above provides strong evidence that **Observation 1** is indeed true. Further, the fact that the introduction of delays does not influence **MII** shows the advantages of using the CRT: without these delays **II** would have to be increased to satisfy the modulo scheduling constraint.

6 Related Work

Resource-constrained software pipelining has been studied extensively by several researchers and a number of *modulo scheduling algorithms* [1, 3, 4, 5, 6, 9, 11, 12, 16, 17, 18, 20, 21, 22] have been proposed in the literature. A comprehensive survey of these works is provided by Rau and Fisher in [15]. As mentioned in Section 4.3, the co-scheduling method discussed in this paper uses a variation of Huff's Slack Scheduling method [9].

The work presented in this paper is unique in the sense that it coordinates the scheduling of both hardware structures and software pipelined schedules in a single *co-scheduling* framework to achieve high instruction level parallelism. To the best of our knowledge, there is no explicit use of the well-developed classical pipeline theory (or its adaptation) in software pipelining methods. In contrast our co-scheduling approach does so. The co-scheduling framework complements other related work in resource-constrained software pipelining by considering a special class, viz. *instruction pipelines*. It is very effective for handling arithmetic pipelines.

There is another major difference between our co-scheduling. and other approaches. In co-scheduling, the software pipeline initiations are represented in a Modulo Initiation Table while resource conflicts of hardware pipeline structures are handled in the Cyclic Reservation Table. In contrast other modulo scheduling algorithms use a single Modulo Reservation Table to represent both resource conflicts and initiation time. Separating them, as in our method, facilitates achieving better and faster schedules.

An attractive feature of the co-scheduling framework is that it opens up new avenues by facilitating the use of classical delay insertion technique to improve instruction level parallelism. A brief discussion on this presented in the following section.

7 Future Work

In this paper, the reconfiguration of hardware pipelines is restricted to satisfying the modulo scheduling constraint. However it is possible to reconfigure, i.e. introduce delays in the pipelines, to improve the number of initiations. Theorem 3.2 establishes an upper bound for the maximum number of initiations based on two bounds, namely (B1) the number of 0's in initial cyclic collision vector (which is same as the cardinality of the cyclic permissible latency set) and (B2) the ratio of **II** to d_{max}. The second bound (B2), is stringent in the sense that it can never be increased no matter how delays are introduced in the pipeline. In contrast bound (B1) can be influenced by the introduction of non-compute delay stages. Thus whenever the maximum number of initiations in the MS-state diagram does not reach bound (B2), one could first choose a latency cycle, with the number of initiations equal to (B2) and period equal to **II**, and then reconfigure the pipeline to obtain the desired latency cycle. It is not clear whether such reconfiguration is always possible. Classical pipeline theory [13] hints that it is. However, a number of issues are still open. For example: How does one choose a desired latency cycle? What guarantee is there that the desired latency sequence will not increase **II**? How is the selection (of a latency sequence) done in the presence of multiple `FU` types? With multiple `FU` types, reconfiguration and the resulting **II** values are not unique, how can it be ensured that minimum **II** values are always tried? A more detailed and thorough investigation is required to answer these questions satisfactorily.

We also plan to extend our co-scheduling method for multi-function pipelines since, in practice, it is likely that a single FU can support multiple operations, with each possibly having a different reservation table. We also plan to investigate how to integrate co-scheduling with other modulo scheduling methods to efficiently handle both instruction and arithmetic pipelines. Also, as indicated in Section 5 we would like to see how co-scheduling performs in the presence inter-stage dependencies (in the pipeline stages).

8 Conclusions

In this paper we have proposed Co-Scheduling, a unified framework that performs the scheduling of hardware and software pipelines. The proposed method uses and extends classical pipeline theory to obtain better software pipelined schedules, as has been demonstrated through both examples and experimental results. As part of Co-Scheduling, we have introduced the Modulo Initiation Table (MIT) and Cyclic Reservation Table (CRT) as alternatives to the standard modulo reservation table.

We have implemented Co-scheduling and run experiments on a set of 1013 loops taken from various benchmark suites such as `SPEC92`, the `NAS kernels`, `linpack` and `livermore`. The median time for Co-Scheduling to handle one loop was 0.29 seconds. In 61% of the loops, our schedules achieved the Minimum Initiation Interval **MII**. In 92% of the loops, the achieved **II** was within *4* cycles of **MII**, and and 98% of the loops were successfully scheduled in all. The mean and median for (**II** − **MII**) were respectively 0 and 1.56, while the mean and median for (**II**/**MII**) were 1.0 and 1.08.

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] 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.

[7] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Pub., Inc., 1990.

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

[9] 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.

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

[11] 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.

[12] 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.

[13] J. H. Patel and E. S. Davidson. Improving the throughput of a pipeline by insertion of delays. In *Proc. of the 3rd Ann. Symp. on Computer Architecture*, pages 159–164, Clearwater, Flor., Jan. 19–21, 1976. IEEE Comp. Soc. and ACM SIGARCH.

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

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

[16] 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.

[17] 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.

[18] B. Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of the 27th Ann. Intl. Symp. on Microarchitecture*, pages 63–74, San Jose, Calif., Nov. 30–Dec.2, 1994. ACM SIGMICRO and IEEE-CS TC-MICRO.

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

[20] 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.

[21] 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.

[22] 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 Summary of Hardware Pipeline Terminology

The following table is extracted from [10].

Terminology	Meaning
Reservation Table	Two-dimensional representation of flow of data through for one function evaluation.
Initiation	Start of one instruction evaluation.
Latency	Number of time units between two initiations.
Latency Sequence	Sequence of latencies between successive initiations.
Latency Cycle	A latency sequence that repeats itself indefinitely.
Collision	Attempt by two different initiations to use same stage at same time.
Forbidden Latency	A latency that causes collision between two initiations.
Permissible Latency	A latency that does not cause collision between two initiations.
Initiation Rate	Average number of initiations per clock unit.
Average Latency	Average number of clock units between initiations. Equals reciprocal of initiation rate.
Minimum Average Latency (MAL)	Smallest average latency.
Collision Vector	Vector showing permitted latencies between two initiations from the same (reservation) table.

B Proof of Theorem 3.1

We state and prove Theorem 3.1 in this section.

Theorem 3.1 *The collision vector of every state S in the MS-state diagram derived according to* **Procedure 1** *represents all permissible (and forbidden) latencies in that state, taking into account all initiations made so far to reach the state S.*

Proof: At each state, there is an arc to the next state only if there is permissible latency p in the current collision vector. This follows from **Step 2**. Next we need to show that the collision vector in the next state correctly represents the permissible and forbidden latencies under the co-scheduling framework. Since by **Observation 3.2** the permissible and forbidden latency sets in cyclic scheduling are complements of each other, it is sufficient to consider either one of them. We consider the permissible set. The proof of the Theorem is by induction. By definition, the initial cyclic collision vector represents the permissible set correctly in the initial state. Assume the collision vector in state S_i represents the permissible sets correctly for all states S_i which have a maximum path length n from the initial state. If there is a state S_{i+1} from S_i with a permissible latency p, we have to prove that the collision vector of state S_{i+1}

is correct. To prove this we consider two parts of the collision vector of state S_i, namely those corresponding to latencies greater than or equal to p and those less than p. These two parts correspond respectively to the first $(\mathbf{II} - p)$ bits and the last p bits of the collision vector in S_{i+1}.

Part 1 First $\mathbf{II} - p$ bits of S_{i+1}: From the definition of the MS-state diagram, any latency $p' > p$ at state S_i corresponds to a latency $p' - p$ in state S_{i+1}. If the latency p' is forbidden in S_i, i.e. the p'-th bit of the collision vector is 1, then the $(p' - p)$-th bit of the collision vector in S_{i+1} must be 1. This is guaranteed by the `rotate-left` operation. (The logical `OR` operation performed subsequently does not affect this.) Now consider if p' was permissible in S_i. The corresponding latency value $p' - p$ in state S_{i+1} may or may not be permissible depending on the initial cyclic collision vector. If $p' - p$ is forbidden in the initial cyclic collision vector, it should be forbidden in state S_{i+1}. Otherwise, it should be permissible. It can be seen that after the `rotate-left` operation the $(p' - p)$-th bit will be 0. However the logical `OR`-ing with the initial cyclic collision vector will set the $(p' - p)$-th bit in the collision vector to 1 or 0 depending on whether $p' - p$ is forbidden or permissible in the initial state.

Part 2 Last p bits of S_{i+1}: These correspond to latency values from $(\mathbf{II} - p)$ to $(\mathbf{II} - 1)$. A latency f in this range is forbidden in state S_{i+1} if f is in the cyclic forbidden latency set. The logical `OR`-ing of the initial cyclic collision vector ensures this. If f is in the cyclic permissible latency set, then the corresponding bit in state S_{i+1} may be 1 or 0 depending on the initiations made up to state S_i. This is because in our co-scheduling framework, instructions (from different iterations) are initiated according to the latency sequence in each software pipeline cycle. Due to the inductive hypothesis, the collision vector in S_i correctly captures the permissible (and forbidden latencies) in state S_i, taking into account all the initiations made thus far. Further, the information required for latency values in the range $\mathbf{II} - p$ to $\mathbf{II} - 1$ is available in the first p bits of the collision vector in state S_i. The `rotate-left` operation, preserves these bits as the last p bits of the collision vector in state S_{i+1}. From there it follows that any latency $f \in [\mathbf{II} - p, \mathbf{II} - 1]$ is forbidden in state S_{i+1} if bit $f + p - \mathbf{II}$ is 1. Otherwise the latency f in state S_{i+1} is permissible. □

C 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