**In the International Journal of Parallel Programming, vol.28, no. 1, pages 1–46, Feb. 2000.**

# Enhanced Co-Scheduling: A Software Pipelining Method using Modulo-Scheduled Pipeline Theory

**R. Govindarajan**
Supercomputer Edn. & Res. Centre
Computer Science & Automation
Indian Institute of Science
Bangalore 560 012, India
govind@{serc,csa}.iisc.ernet.in

**N.S.S. Narasimha Rao**
Novell Software
Development India Ltd.
Garvephavipalya
Bangalore 560 068, India
nnarasimharao@novell.com

**E.R. Altman**
IBM T.J. Watson
Research Center
Yorktown Heights
NY 10598, U.S.A.
erik@watson.ibm.com

**Guang R. Gao**
Electrical & Computer Engg.
University of Delaware
Newark
DE 19716, U.S.A
ggao@eecis.udel.edu

**Abstract**

Instruction scheduling methods which use the concepts developed by the classical pipeline theory have been proposed for architectures involving deeply pipelined function units. These methods rely on the construction of state diagrams (or automatons) to (i) efficiently represent the complex resource usage pattern, and (ii) analyze *legal* initiation sequences, *i.e.,* those which do not cause a structural hazard. In this paper, we propose a state-diagram based approach for modulo scheduling or software pipelining, an instruction scheduling method for loops. Our approach adapts the classical pipeline theory for modulo scheduling, and, hence, the resulting theory is called Modulo-Scheduled pipeline (MS-pipeline) theory. The state diagram, called the Modulo-Scheduled (MS) state diagram is helpful in identifying *legal* initiation or latency sequences, that improve the number of instructions initiated in a pipeline. An efficient method, called Co-scheduling, which uses the legal initiation sequences as guidelines for constructing software pipelined schedules has been proposed in this paper. However, the complexity of the constructed MS-state diagram limits the usefulness of our Co-scheduling method.

Further analysis of the MS-pipeline theory, reveals that the space complexity of the MS-state diagram can be significantly reduced by identifying *primary* paths We develop the underlying theory to establish that the reduced MS-state diagram consisting only of primary paths is *complete*; *i.e.,* it retains all the useful information represented by the original state diagram as far as scheduling of operations is concerned. Our experiments show that the number of paths in the reduced state diagram is significantly lower — by 1 to 3 orders of magnitude — compared to the number of paths in the original state diagram.

The reduction in the state diagram facilitate the Co-scheduling method to consider multiple initiations sequences, and hence obtain more efficient schedules. We call the resulting method, enhanced Co-scheduling. The enhanced Co-scheduling method produced efficient schedules when tested on a set of 1153 benchmark loops. Further the schedules produced by this method are significantly better than those produced by Huff's

*Slack Scheduling* method, a competitive software pipelining method, in terms of both the initiation interval of the schedules and the time taken to construct them.

**Keywords:** Instruction-Level Parallelism, Software Pipelining, Classical Pipeline Theory, Co-scheduling, VLIW/Superscalar Architectures

# 1 Introduction

Pipelining is one of the most efficient means of improving performance in high-end processor architectures. Historically, design techniques for hardware pipelines with *structural hazards* 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 [1, 2].

In the compiling front, a technique — known as software pipelining — to exploit higher instruction-level parallelism in modern architectures has become increasingly popular for loop scheduling. A software pipelined schedule overlaps operations from different loop iterations in an attempt to fully exploit instruction-level parallelism. The interval between the initiation of two successive iterations of a loop is known as the *initiation interval* (**II**). A variety of software pipelining algorithms [3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16] have been proposed which operate under resource constraints. An excellent survey of these algorithms can be found in [17].

In the past decade, technology advance has made it feasible to design very aggressive arithmetic and instruction pipelines in commodity microprocessor architectures, *e.g.,* superpipelined architectures and superscalar architectures. Processors capable of issuing 8 instructions per cycle are on the horizon and Very Long Instruction Word (VLIW) architectures are staging a resurgence. With these, structural hazard resolution in modern processors is expected to be more complex. Furthermore, in certain emerging application areas, such as mobile computing or space vehicle on-board computing, the size, weight and power consumption may put tough requirements on the processor architecture design, which may result in more resource sharing, and, in turn, resulting in pipelines with more structural hazards.

With such complex resource usage, the scheduling method must check and avoid any *structural hazard*, *e.g.,* contention for hardware resources by instructions. This is accomplished by maintaining the *modulo reservation table* [8, 10, 12, 17] to model the resource usage. One drawback of this method, especially in the presence of complex resource usage, is its inefficiency. For scheduling each instruction, the method attempts to place the operation in successive cycles in the reservation table until it finds a cycle which does not cause a hazard (resource conflict). This type of greedy try-retry approach may not be very efficient especially when the pipelines involve arbitrary structural hazards — since each trial decision is made "locally" and greedily without any underlying guideline of what might be the best sequence of trial step to pursue for a given initiation interval.

Recently, a finite state automaton (FSA)-based scheduling technique — using ideas from the classical hardware pipeline theory [1, 2, 18] — has been proposed for general instruction scheduling [19, 20, 21]. In this method, the resource usage is modeled using forbidden/permissible latencies and a state diagram. This has effectively reduced the problem of checking structural hazards to a fast table-lookup, thereby getting a good speedup in the scheduling time. In an independent work, a software pipelining method, called *Co-scheduling*, that makes use of classical pipeline theory and state diagram construction, has been proposed by us in [22]. The constructed state diagram, called a Modulo-Scheduled (MS)-state diagram, represents valid initiation sequences

that do not cause any structural hazard. Each path in the MS-state diagram corresponds to a set of time steps at which different instructions can be initiated in the given pipeline under modulo scheduling without incurring any structural hazard. In the Co-Scheduling method, a single path in the MS-state diagram, and the time steps corresponding to it are used to guide the software pipelining method. The Co-Scheduling method is based on Huff's bidirectional slack scheduling [8].

In this paper, we first discuss the underlying theory, called Modulo-Scheduled (MS) Pipeline theory, for our Co-Scheduling method. We identify that the number of paths in a constructed MS-state diagram could be very large, in the order of several millions, prohibiting the practical use of our Co-Scheduling method. However, we observe that a significant number of these paths are redundant and can be eliminated by identifying what are called *primary paths*. A state diagram consisting only primary paths is termed as a *reduced state diagram*. We develop the underlying theory for the reduced state diagram and establish its *correctness* and *completeness*. The reduced state diagram consists of a few hundred paths, resulting in a reduction in number of paths by 2 to 3 orders of magnitude. Further, the theory of reduced state diagram also reveals an alternative and direct method for generating the information corresponding to the primary paths.

The theory of reduced MS-state diagram enables us to enhance the original Co-Scheduling method by using information from multiple (primary) paths and the corresponding initiation sequences to guide the software pipelining method. This eliminates a fundamental problem of our original Co-Scheduling method, that of being restricted to a single path and the use of the corresponding latency sequence in the software pipelining method. We evaluate the performance of the enhanced Co-Scheduling method and compare it with Huff's Slack Scheduling.

The major contributions of this paper are:

(1) The underlying theory of MS-pipelines and its application to software pipelining;

(2) The underlying theory for reduced MS-state diagram and the drastic reduction in the number of paths in the reduced state diagram;

(3) Alternative method for generating latency sequences corresponding to primary paths;

(4) Performance evaluation of enhanced Co-Scheduling; and

(5) Comparison of enhanced Co-Scheduling with Huff's slack scheduling method [8].

As mentioned earlier, the proposed enhanced Co-Scheduling method as well other finite state automaton based scheduling methods significantly reduce the time to construct the schedule compared to reservation table based approaches [8, 10, 12, 17]. However, this reduction in scheduling time comes at the cost of computational overhead for constructing the state diagram, if it is constructed on-line during the scheduling process, or space overhead to store the state diagram, if the construction of the state diagram was done off-line. While the

construction of the automaton is done *once* in the case of basic instruction scheduling [19, 20, 21], for software pipelining, the state diagram needs to be constructed for each initiation interval. Hence, it may be advantageous to construct these state diagrams for a given target architecture off-line and store them in a database, even though as just noted, this increases the storage overhead.

In the following section we provide the necessary background. In Section 3, we motivate the need for the Co-Scheduling framework with a number of examples. In Section 4, the theory of MS-pipeline is developed. Section 5 deals with the theory of reduced MS-state diagram. The enhanced Co-Scheduling method is discussed in Section 6. In Section 7, we present the experimental results of the enhanced Co-Scheduling method. Section 9 compares our approach to other related work. Discussion on future work is presented in Section 8 and concluding remarks in Section 10.

# 2 Background

In this section, we provide the necessary background material for software pipelining and a review of the classical pipeline theory for hardware pipelines. Readers familiar with this topic can proceed to the next section.

## 2.1 Software Pipelining

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 [17]. The minimum initiation interval (**MII**) is constrained by both *loop-carried* dependences (or recurrences) and available resources [3, 5, 8, 10, 17]. Loop-carried dependences put a lower bound, **RecMII**, on **MII**. The value of **RecMII** is determined by the critical (dependence) cycle(s) [23] in the Data Dependency Graph (DDG) of the loop. Specifically

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

along the critical cycle(s).

Another lower bound **ResMII** on **MII** is enforced by resource constraints. Let $d_{max,r}$ represent the maximum number of cycles for which an instruction uses any stage of a function unit (`FU`) type $r$ (*e.g.,* ADDER). 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 or equal to $\lceil N_r * d_{max,r}/F_r \rceil$. Thus **ResMII** is the maximum of this bound taken over all `FU` types:

$$\mathbf{ResMII} = \max_r \left\lceil \frac{N_r * d_{max,r}}{F_r} \right\rceil \tag{2}$$

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

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

Existing software pipelining methods keep track of the resources committed for the scheduled instructions through a Modulo Reservation Table (MRT) [5, 8, 10, 24]. The MRT contains **II** rows and a number of columns one 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.,* height or slackness) to pick which operation to schedule next. (3) Schedule the high-priority operation in a time slot 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.

In Section 3.1, we will see how the GSA is applied for an FU having structural hazard. Next, we review the classical pipeline theory of hardware pipelines.

## 2.2 Classical Pipeline Theory

In hardware pipelines, the resource usage of various pipeline stages are represented by a two dimensional *Reservation Table* [2]. 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.

Consider the reservation table for a pipelined FU is shown in Figure 1(a). Latencies 2 and 4 are forbidden while latencies 1, 3, and 5 are permissible. Classical pipeline theory identifies initiation sequences or *latency sequences* which maximize the throughput and the utilization of the pipeline using *state diagrams* [2]. Each state in the state diagram is represented by a *collision vector*. 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. The collision vector for the considered reservation table is `101010`.

The construction of the state diagram proceeds as follows.

**Step 1** Start with the initial Collision Vector.

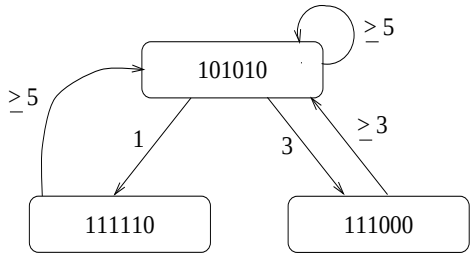
**Step 2** For each permissible latency $p$ in the current state, *i.e.,* for all bits $p$ in the collision vector whose value is 0, derive a subsequent state as follows.

(a) `shift-left` the current collision vector by $p$ bits.

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

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

(a) Reservation Table




(b)

Figure 1: An Example Reservation Table and its State Diagram

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

The state diagram for the example reservation table is shown in Figure 1(b). Each path in the state diagram represents a legal latency sequence and it is guaranteed that operations initiated according to the latency sequence do not cause any collision. A latency sequence that repeats itself is known as a **latency cycle**. For example, the following latency cycles can be identified in the state diagram: {3,3}, {1,5}, {5}. The sum of the latencies in a latency cycle is known as the **period** of the latency cycle. Lastly, the **throughput** of a latency cycle is the ratio of the number of operations initiated in the latency cycle to its period. Analysis of this state diagram reveals that latency sequences {1,5} and {3,3} give a maximum throughput of 1/3.

# 3 Motivation

To the best of our knowledge, none of the the software pipelining approaches 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.

## 3.1 Need for Pipeline Theory

Consider a loop with 3 operations $i_1$, $i_2$, and $i_3$, to be scheduled in the FU whose reservation table is shown in Figure 1(a). Assume the initiation interval **II** for the given loop is 9. Let the earliest (*Estart*) and latest start (*Lstart*) time steps [8] at which these 3 operations can be scheduled are *[2,3]*, *[3,5]*, and *[7,9]*. The GSA discussed in Section 2.1 will attempt to schedule the operations based on, say the *slack* — difference between the Estart and Lstart times [8]. Let operation $i_1$ be scheduled at its earliest time 2. The GSA uses the MRT to maintain the information on committed resources. The resource usage corresponding to this initiation is shown

in Figure 2(a). Now, operation $i_2$ can also be scheduled at its earliest time 3 since the latency, 1, between these two operations is permissible.

Once operations $i_1$ and $i_2$ are scheduled at time steps 2 and 3, it can be verified that the initiation of another operations at any time step will cause a structural hazard. Thus the greedy strategy of initiating $i_2$ at the earliest possible slot in the MRT inhibits the scheduling of operation $i_3$. However, it is possible to schedule the three instructions at time steps 2, 5, and 8, without causing any structural hazard, as shown in Figure 2(b). Note that the resource usage beyond cycle 8 wraps around, since **II**= 9. The GSA may recover from the earlier indicated wrong move in its backtracking step (Step 4) and could possibly schedule the operations at time 2, 5, and 8. However, this will results in a considerable number of retries. Also there is no guarantee that the GSA will recover from such wrong moves to eventually schedule all three operations with an **II** = 9.

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). The analysis from the state diagram reveals that initiations at time steps *2, 5, 8* are **permissible** and better utilize the pipeline—handling 3 operations every 9 cycles which gives the optimal throughput for the given FU. Thus knowing and using the optimal latency sequences in the software pipelining method facilitate producing better schedules and producing them faster (*i.e.,* in less compile time).

| Stage | Time Steps | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1 | | | $i_1$ | $i_2$ | $i_1$ | $i_2$ | $i_1$ | $i_2$ | |
| 2 | | | | $i_1$ | $i_2$ | | | $i_1$ | $i_2$ |
| 3 | | | | | | $i_1$ | $i_2$ | | |

(a) Initiation at time *2,3*

| Stage | Time Steps | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1 | $i_2$ | $i_3$ | $i_1$ | $i_3$ | $i_1$ | $i_2$ | $i_1$ | $i_2$ | $i_3$ |
| 2 | $i_3$ | $i_2$ | | $i_1$ | $i_3$ | | $i_2$ | $i_1$ | |
| 3 | | | $i_3$ | | | $i_1$ | | | $i_2$ |

(b) Initiation at time *2,5,8*

Figure 2: Modulo Reservation Tables

Secondly, the GSA model individual stages in an `FU` as separate resources in the *Modulo Reservation Tables (MRT)*. As a consequence, for a modern VLIW with less than 10 FUs, the number of stages could be as much as 50! Using the GSA on the resulting large MRT can increase the scheduling time. On the other hand, use of pipeline theory and the use of legal latency sequences allow each function unit to be modeled as a single resource instead of a number of pipeline stages.

Use of classical pipeline theory avoids several unnecessary tries made by the GSA at conflicting latencies. Even a simple extension, of using the fact that latencies 2 and 4 are forbidden, to earlier software pipelining approaches leads avoiding several attempts to initiate operations at the above forbidden latencies.

Thus, this section clearly brings out the advantage of using classical pipeline theory in existing software pipelining methods.

## 3.2 The Need for Modulo-Scheduled Pipeline Theory

Can one directly use classical pipeline theory in the context of software pipelining? We answer this question in this subsection which motivates our proposed Modulo-Scheduled (MS-) pipeline theory.

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**. 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 our motivating example.

Consider the reservation table and its state diagram shown in Figure 1. Assume **II** = 9. From the state diagram shown in Figure 1(b), we observe that the latency cycle {`1,5`} yields maximum throughput. However, under modulo scheduling with **II** = 9, scheduling two operations with a latency 5 will cause a collision, as shown in Table I, even though classical pipeline theory states that 5 is a permissible latency. It can be seen that the collision was caused by the "wrap-around" resource usage in the MRT. This is not unexpected since the state diagram is obtained for a reservation table with 6 columns and was derived without a wrap-around resource usage in mind.

| Stage | Time Steps | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1 | 0,5 | | 0 | | 0 | 5 | | 5 | |
| 2 | | 0,5 | | | | 0 | 5 | | |
| 3 | | | | 0 | | | | | 5 |

Table I: Initiation of Instructions at *(0, 5)* in the Modulo Reservation Table

The classical pipeline theory [1, 2] **does** indicate that *5* is an impermissible latency for any cycle with period 9, since *5* is the complement of the forbidden latency *4* in the modulo space with **II** = 9. However, the focus in

these works [1, 2] is on how to reconfigure the hardware pipelines for a *given latency cycle*.[1] Whereas here we are interested in *finding the "best" latency cycle* for a given initiation interval **II**.

As the above example shows, the state diagram constructed using the classical pipeline theory does not account for the software pipelining **II**. As a consequence, some latency cycles identified as legal by the state diagram may lead to structural hazards under modulo scheduling. In the following section we show how to extend the classical pipeline theory to achieve the simultaneous scheduling of hardware and software pipelines.

# 4 Modulo Scheduled Pipelines

In this section we revisit the classical pipeline theory in the context of software pipelining. To differentiate our approach from the classical pipeline theory, we refer to our pipelines as *Modulo-Scheduled (MS-)* pipelines. We define the terms *reservation table*, *forbidden latency*, *collision vector*, and *state diagram* as they apply to MS-pipelines. We then develop the theory of MS-pipelines which in turn forms the basis for our Co-scheduling method.

## 4.1 Preliminaries

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 $d_{max,r}$ to denote the maximum number of cycles for which any stage of the pipeline is used.

Modulo scheduling constraint [4, 12, 17] prohibits the use of any resource (stage of a functional unit) at time steps separated by multiples of **II** by operations belonging to the same iteration. Since each operation needs at least one stage for $d_{max,r}$ time steps, modulo scheduling requires **II** to be greater than or equal to $d_{max,r}$. Formally,

**Lemma 4.1** *If* `FU` *type* $r$ *is used by the schedule, the initiation interval of a software pipelined schedule* $\mathbf{II} \geq d_{max,r}$.

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

[1] 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 8.

With MS-pipelines, each instruction must be initiated in the pipeline once every **II** cycles. Therefore it would be appropriate to use a reservation table with **II** (rather than $l_r$) columns. Notice that Lemma 4.1 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) **II** $> l_r$, (2) $l_r >$ **II**, or (3) $l_r =$ **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 $CRT_r[s,t]$. The CRT for the reservation table in Section 2.2 is shown in Figure 3(a).

With the folding required in case (2), multiple X marks separated by **II** may be placed in the same column of the CRT. However, fortunately, the modulo scheduling constraint already prohibits such occurrences. Thus if the reservation table satisfies the modulo scheduling constraint, the cyclic reservation table will not have two x marks on the same column of the CRT[2].

Next we define several terms.

**Definition 4.1 (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 $CRT[s,(t+f)$ mod **II**$]$, contain an X mark.*

It can be easily seen that in a MS-pipeline latency values $f$ greater than **II** are equivalent to $f$ mod **II**. Hence, for MS-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 Figure 3(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 $\{0, 2, 4, 5, 7\}$.

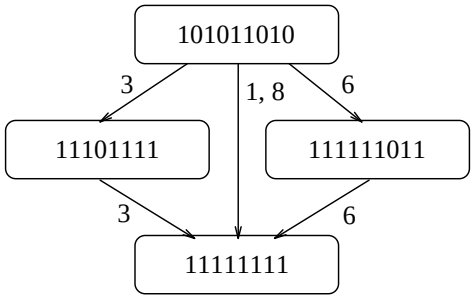
**Definition 4.2 (Cyclic Permissible Latency)** *A latency $f \leq$ **II** is said to be a* **cyclic permissible latency** *if $f$ is not in the cyclic forbidden latency set.*

For the CRT in Fig. 3(a), the cyclic permissible latencies are 1, 3, 6, and 8. From the above definitions it can be easily observed that: From the definition of cyclic forbidden latency, it can be seen that if $f$ is a forbidden latency, then the latency **II** $- f$ is also forbidden. A similar property holds for all cyclic permissible latencies also.

[2]However, if this is not the case, scheduling constraint, it is possible to satisfy the modulo scheduling constraint by either incrementing **II** by 1 or modifying the hardware so as to delay all but one of the operations mapping to the same time $t$ as in [1]. A discussion on the introduction of such delays and their impact on their hardware architecture is beyond the scope of this paper.

| 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**

Figure 3: A Cyclic Reservation Table and its MS-State Diagram

## 4.2 State Diagram for Cyclic Reservation Tables

Our interest 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 same way as is done in classical pipeline theory. We use the term **Modulo-Scheduled State Diagram (MS-state diagram)** to distinguish it from the state diagram of the classical pipeline theory. 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 4.3 (Cyclic Collision Vector)** *A Cyclic Collision vector is a binary vector of length* **II**, *with the bits numbered from* 0 *to* **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 1(a) with the forbidden latency set as $\{0, 2, 4, 5, 7\}$, the initial cyclic collision vector is `101011010`.

The construction of the MS-state diagram proceeds as follows.

**Procedure 4.1 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.,* for each bit $p$ in the collision vector whose value is 0, derive a new state as follows.

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

(b) Logically `OR` the resulting vector with the initial cyclic collision vector to get the collision vector of the new state.

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

The MS-state diagram for the CRT in Figure 3(a) is shown in Figure 3(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(b), the state `111111111` can be reached from the initial state with a latency value of either 1 or 8, or from states $S_1$ or $S_2$.

Observe that there is a very close resemblance of **Procedure 4.1** to the state diagram construction in the classical pipeline theory. The main difference is that in Step 2(a) of **Procedure 4.1** a `Rotate-left` operation is performed rather than a `shift-left` operation. For example, the cyclic collision vector `101011010` when rotated left by 3-bits gives `011010101`. Compare this with the result of a `shift-left` operation by 3-bit which would have given `101011000`. Notice that the rightmost 3-bits in rotate left is 010 indicating that a latency 8 (apart from latencies 0, 2, 4, and 5) is forbidden in the new state. More precisely, after two initiations at time steps 0 and $(0 + 3)$, a latency of 8 at time step $0 + 3 + 8 = 11$ will cause a collision. Why? Because, another instance (from the following iteration) of the instruction which was initiated at time step 0 will be initiated at time step $0 + \mathbf{II} = 0 + 9$. This operation will have a latency 2 with the operation initiated at time step 11. Since 2 is in the cyclic forbidden latency set, there will be a collision.

If there were no software pipelining, *i.e.,* we only have the problem of scheduling hardware pipelines, then of course, the collision vector (obtained by a shift-left operation) will have a 0 in bit position 8 indicating that a new operation can indeed be initiated at time 11, and there will be no collision. Thus, the `rotate-left` operation in Step 2(a) accounts for the initiation of instructions (from different iterations) at time steps that differ by **II**. Thus, in modulo scheduling, an instruction scheduled at time step $p$ in the repetitive pattern 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 4.1** *The collision vector of every state $S$ in the MS-state diagram derived according to* **Procedure 4.1** *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** in Procedure 4.1. Next we need to show that the collision vector in the next state correctly represents the permissible and forbidden latencies under modulo scheduling. 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. □

## 4.3 An Alternative Representation of MS-State Diagram

Instead of representing each state in the MS-state diagram by its collision vector, the set of permissible latencies in the current state can be used to represent the state. The latter is a more direct representation than the former. In addition, we find the latter to be more useful in establishing certain properties of the MS-state diagram. Therefore, we present a direct construction method involving this alternative representation for the MS-state diagram.

**Procedure 4.2: Alternative Construction Procedure for MS-State Diagram:**

**Step 1** The initial state $S_0$ of the MS-State diagram contains the (initial) permissible latency set $S_0 = \{p_1, p_2, \cdots, p_k\}$. We will use the state name, *e.g.,* $S_0$, itself to represent the permissible latencies in the given state.

**Step 2** For each permissible latency $p_i$ in the current state $S$, there is an arc from $S$ to a new state $S'$. $S'$ represents the state with a new initiation $p_i$ cycles after state $S$. Also, the set $S'$, computed as below, represents the set of latencies at which a further initiation can be started from state $S'$. The permissible latencies in the new state is given by $S' = S_{-p_i} \cap S_0$ where $S_{-p_i}$ is defined as

$$S_{-p_i} = \{(p_j - p_i) \bmod \mathbf{II} \mid p_j \in S\}.$$

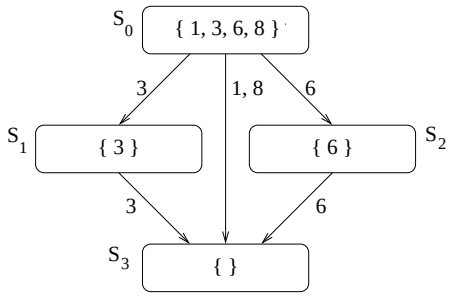



Figure 4: An Alternative Representation of the MS-State Diagram

Some explanation of Step 2 in the construction of the MS-state diagram may be required to have a clear understanding of the state diagram. The set $S_{-p_i}$ is obtained by subtracting $p_i$, the chosen latency, from each permissible latency $p_j$ in $S$. The subtractions are performed modulo **II**. This step corresponds to the `rotate-left` operation in Procedure 4.1. Intuitively, the set $S_{-p_i}$ is the set of latencies, that **may be** permissible from the new state $S'$. However, for a latency $l$ to be permissible in the new state $S'$, $l$ must be in the (initial) permissible latency set $S_0$ . Thus the set of permissible latencies in the new state $S'$ is the intersection of $S_0$ and $S_{-p_i}$. It can be observed that this step corresponds to the logical `OR`ing step in Procedure 4.1.

Next, we establish the equivalence between the representations by showing the equivalence between Procedures 4.1 and 4.2.

**Theorem 4.2** *Procedure 4.1 and Procedure 4.2 are equivalent. That is, the MS-state diagrams constructed by them are equivalent and contain the same information.*

**Proof:** The proof this theorem follows from the fact that (i) the definition of $S{-}p_i$ corresponds to the rotate-left operation in Step 2(a) of Procedure 2.1, (ii) the intersection with $S_0$ is equivalent to the logical-`OR` operation[3],

[3] Note that logical-`OR` actually corresponds to set union (of 1's); but in the alternative representation, the states are represented by permissible latencies which correspond to 0's. Thus to get the set of permissible latencies in a state, we take the union of forbidden latencies and complement it. This corresponds to taking set intersection.

and (iii) the collision vector (of a state) represents the same information as the set of permissible latencies in that state. □

Henceforth we will use the alternative representation of the MS-state diagram. The alternative representation of the MS-state diagram for our example reservation table is shown in Figure 4. Lastly,

**Definition 4.4** *The* **final state** *of an MS-state diagram is one which contains an empty permissible latency set.*

## 4.4 Analyzing the MS-State Diagram

A path $S_0 \xrightarrow{p_1} S_1 \xrightarrow{p_2} S_2 \cdots \xrightarrow{p_k} S_k$ in the MS-state diagram corresponds to a sequence of initiations which are permissible. The latencies, $p_1, p_2, \cdots, p_k$ associated with the path correspond to the latencies between successive initiations.

**Definition 4.5** *Given the above path* $S_0 \xrightarrow{p_1} S_1 \xrightarrow{p_2} S_2 \cdots \xrightarrow{p_k} S_k$, *successive initiations are made at time steps* $0, p_1, (p_1 \oplus p_2), \cdots, (p_1 \oplus p_2 \cdots \oplus p_k)$, *where* $\oplus$ *refers to addition modulo* **II**. *These values are referred to as* **offset values** *from the first initiation made at time 0. The set* $\{0, p_1, (p_1 \oplus p_2), \cdots, (p_1 \oplus p_2 \cdots \oplus p_k)\}$ *is referred to as the* **permissible offset set**.

For example, the path $S_0 \xrightarrow{3} S_1 \xrightarrow{3} S_3$ corresponds to initiations at offset values 0, $(0 + 3)$ `mod` **II**, and $(0 + 3 + 3)$ `mod` **II**. Henceforth, without loss of generality we always assume: *(i)* the first initiation is made at time 0 and *(ii)* offset values are specified in modulo **II**.

**Lemma 4.2** *Each offset value (except offset 0) of any path in the MS-state diagram corresponds to a permissible latency.*

**Proof:** The proof is by contradiction. If the offset value corresponding to an initiation is not a permissible latency, then the latency between that initiation and the one corresponding to offset 0, is forbidden, and hence the initiation is not legal. This contradicts the fact that all initiations corresponding to a path are legal. Hence the Lemma. □.

The number of initiations made corresponding to a path in the MS-state diagram equals $\mathcal{L}(P) + 1$, where $\mathcal{L}(P)$ represents the length of the path $P$ in terms of the number of arcs. There can be several paths from $S_0$ to $S_k$. As we are interested in maximizing the number of initiations in a pipeline, we consider the longest path from the initial state $S_0$. The maximum number of initiations *Max_Init* possible in an MS-pipeline is given by the longest path from the start state to the final state. For example, for the state diagram shown in Figure 4, the *Max_Init* is 3 corresponding to the paths $S_0 \xrightarrow{3} S_1 \xrightarrow{3} S_3$ and $S_0 \xrightarrow{6} S_2 \xrightarrow{6} S_3$

The *Max_Init* of an MS-pipelined is bounded by an upper bound (*UB_Init*) of possible initiations in the MS-pipeline.

**Theorem 4.3** *The upper bound on the number of operations (UB_Init) that can be initiated in an MS-pipeline during* **II** *cycles is*

$$UB\_Init = \min\left((k+1), \left\lfloor \frac{\mathbf{II}}{d_{max}} \right\rfloor\right)$$

*where* $k$ *is the cardinality of the permissible latency set and* $d_{max}$ *is the maximum number of* `X` *marks in any row in the reservation table.*

**Proof:** The upper bound on the number of initiations is bounded by two factors: The first is due to the number of permissible latencies. By our assumption, the first initiation is always made at time step 0. Further, by Lemma 4.2, initiations are always made only on a permissible latency. Furthermore, at most one operation can be initiated a particular cycle. This is because of the fact that, under modulo scheduling, we will be initiating the successive instances of an operation, belonging to different iterations of the loop, once every **II** cycle. The second bound is due to resource usage. If a particular stage of the pipeline is needed for $d_{max}$ cycles, then, obviously, no more than $\lfloor \mathbf{II}/d_{max} \rfloor$ initiations can be made. Hence the Lemma □.

Note that *Max_Init* specifies the maximum number of initiations **actually** possible in the given MS-pipeline, while *UB_Init* provides an upper bound for *Max_Init* That is,

$$Max\_Init \leq UB\_Init$$

In an MS-state diagram, as we go from the start state to the final state, the number of permissible latencies monotonically decreases. Formally,

**Lemma 4.3** *If there is an arc from* $S$ *to* $S'$ *in the MS-state diagram, then* $|S| > |S'|$*, where* $|S|$ *represents the cardinality of the permissible latency set associated with* $S$*.*

**Proof:** Let $p_i$ be the latency associated with the arc from $S$ to $S'$. From Step 2 of Procedure 4.2, and the definition of $S_{-p_i}$,

$$|S'| = |S_{-p_i} \cap S_0| \leq |S_{-p_i}| = |S|$$

That is, $|S| \geq |S'|$. But we need to show strict inequality. For this, consider the latency $p_i$ in $S$. This latency translates to $p_i - p_i = 0$ in $S_{-p_i}$. Further 0 is not a permissible latency for any single function pipeline. Thus, clearly, the latency corresponding to $p_i \in S$, does not belong to $S'$. Hence $|S| > |S'|$. □

Unlike the state diagram of classical pipeline theory which involve cycles, MS-state diagrams do not contain any directed cycles.

**Lemma 4.4** *There are no directed cycles in the MS-State diagram.*

**Proof:** The proof of this lemma is by contradiction. Assume that there is a directed cycle in the MS-state diagram involving $S_1, S_2, \cdots S_k, S_1$. By Lemma 4.3,

$$|S_1| > |S_2| > \cdots > |S_k| > |S_1|$$

which is impossible. □

The following lemmas show the existence of a final state and the termination of Procedure 4.2.

**Lemma 4.5** *Every MS-state diagram contains a final state.*

**Proof:** The proof of this lemma follows from the fact that the cardinality of the permissible latency set associated with successive states along a directed path decreases. □

**Lemma 4.6** *The construction of the MS-State diagram (***Procedure 3.1***) terminates after a finite number of steps.*

**Proof:** The proof of this lemma follows from Lemma 4.3, 4.4 and 4.5.

One can verify that Lemma 4.3 to 4.5 hold for the MS-state diagram shown in Fig. 4.

# 5 Reduced MS-State Diagram: Motivation and Theory

In this section we motivate the idea for reduced MS-state diagram and develop the necessary theory behind the construction of Reduced MS-state diagrams. Section 5.4 discusses two alternative construction methods for Reduced MS-state diagrams.

## 5.1 Motivation

Each path in a MS-state diagram represents a legal latency sequence and a corresponding set of offset values at which initiations can be made in the MS-pipeline. The latency sequence corresponding to a path in the MS-state diagram is used to guide modulo scheduling in the original Co-Scheduling method [22]. More than the latency sequence of a path, we found the corresponding offset set to be a better representation for guiding the modulo scheduling. But, the number of paths, and hence the number of offset sets in an MS-state diagram, can be quite large (greater than several 100,000s), especially for large values of **II**. For example, for a particular function unit FU-1 discussed in Section 7, there are 1.36 Million paths in the MS-state diagram for an **II** of 24. Further, it should be noted that the MS-state diagram, and, in particular, the number of paths increase drastically for large values of **II**. As a consequence, the construction of the state diagram is expensive in terms of both space and time complexity, especially for large **II** values. The large number of paths in an MS-state diagram also makes precomputing the state diagram and storing the set of all offset sets in a database an expensive proposition.

Consider the state diagram shown in Figure 4. Clearly, the states $S_0, S_1, S_2$, and $S_3$ are all distinct. However, the information represented by the paths $S_0 \xrightarrow{3} S_1 \xrightarrow{3} S_3$ and $S_0 \xrightarrow{6} S_2 \xrightarrow{6} S_3$ are not. Though the latency

sequences corresponding to the above paths are different, it can be seen that the *offset* values for both of them are $0, 3$, and $6$. Thus the latter path (actually any one of the two paths) is redundant. This raises the question, that even though state $S_2$ is distinct, can we avoid generating the state if all the paths that go through $S_2$ are redundant. Equivalently, can we list only the paths that lead to *distinct* offset sets? In our example, removing state $S_2$ and the arcs that are connected to it, does not result in any loss of information. This motivates us to study the theory of reduced MS-state diagrams which involve only paths corresponding to distinct offset sets. In [25] we proposed the construction of reduced MS-state diagrams which involves distinct paths. We establish the necessary theory for reduced MS-state diagrams in this paper.

## 5.2 Definitions

We begin with the following definitions.

**Definition 5.1** *A path* $S_0 \xrightarrow{p_1} S_1 \xrightarrow{p_2} S_2 \cdots \xrightarrow{p_k} S_f$ *in the MS-state diagram is called* **primary** *if the sum of the latency values does not exceed* **II***; that is,* $p_1+p_2+\cdots+p_k <$ **II** *A path is called* **secondary** *if* $p_1+p_2+\cdots+p_k >$ **II**

Note that the operation used in the above definition is simple addition (+), and not $\oplus$. The sum of the latencies along any path in the MS-state diagram will not be equal to **II**. Otherwise, the initiation representing state $S_f$ corresponds to the offset value 0 which causes a collision with the initiation at $S_0$ (the initial state).

Next we adapt the following definitions from [1, 2].

**Definition 5.2** *Two offsets* $o_1$ *and* $o_2$ *belonging to* $\mathcal{O}$ *are* **compatible** *if* $(o_1 - o_2)$ `mod` **II** *is in* $\mathcal{O}$.

**Definition 5.3** *A* **compatibility class** *with respect to* $\mathcal{O}$ *is a set in which all pairs of elements are compatible.*

Two compatible classes for $\{0, 2, 3, 4\}$ are $\{0, 2, 4\}$ and $\{0, 3\}$. Lastly,

**Definition 5.4** *A* **maximal compatibility class** *is a compatibility class that is not a proper subset of any other compatible class.*

The compatibility class $\{0, 2, 4\}$ is maximal, while $\{0, 2\}$ is not. Note that any maximal class of $\mathcal{O}$ will include the element 0.

## 5.3 Theory of Reduced MS-State Diagrams

The compatibility classes of $\mathcal{O}$ are related to the offset sets of different paths in the MS-state diagram. The following lemmas establish that.

**Lemma 5.1** *The offset set of any path from the start state $S_0$ to the final state $S_f$ in the MS-state diagram forms a maximal compatibility class of $\mathcal{O}$.*

**Proof:** Consider the path $S_0 \xrightarrow{p_1} S_1 \xrightarrow{p_2} S_2 \cdots \xrightarrow{p_k} S_f$ where $S_f$ is the final state in the MS-state diagram. Let the offset set for this path be $O = \{o_0, o_1, o_2, \cdots, o_k\}$, where

$$o_0 = 0; \quad o_1 = p_1; \quad o_2 = p_1 \oplus p_2; \quad \cdots \quad o_k = p_1 \oplus p_2 \oplus \cdots \oplus p_k; \tag{4}$$

There are two parts to the proof of this lemma: to prove (i) $O$ is a compatibility class and (ii) $O$ is maximal.

**Part 1:** Consider any pair of offsets $o_i$ and $o_j$. Clearly $(o_i \ominus o_j)$, where $\ominus$ stands for subtraction modulo **II**, must be a permissible latency. Otherwise, the MS-state diagram would consist of a path in which there are two initiations separated by a forbidden latency[4]. This in turn would violate the fact that the above path (and the corresponding latency sequence) is legal, *i.e.,* does not cause any collision. Thus $O$ is a compatibility class of $\mathcal{O}$.

**Part 2:** To prove $O$ is a maximal compatibility class, we use proof by contradiction. Assume that an offset $c$ is compatible with each $o_i \in O$, but is not represented by the path. By definition, all offsets in $O$, except 0, are permissible latencies; *i.e.,* $O - \{0\} \subseteq S_O$. Further, by our assumption $c$ is also in $S_0$. Thus,

$$\{p_1, (p_1 \oplus p_2), \cdots, (p_1 \oplus p_2 \oplus \cdots \oplus p_k), c\} \subseteq S_0 \tag{5}$$

Now, since there is an arc from $S_0$ to $S_1$ with a latency $p_1$, from the construction of the state diagram, and the fact $o_1 = p_1$ is compatible with each $o_i = (p_1 \oplus p_2 \oplus \cdots \oplus p_i)$ and $c$, it can be seen that

$$\{(p_1 \oplus p_2) \ominus p_1, \cdots, (p_1 \oplus p_2 \oplus \cdots \oplus p_k) \ominus p_1, c \ominus p_1\} \subseteq S_1 \tag{6}$$

This can be rewritten, using Equation 4 as.

$$\{o_2 \ominus o_1, o_3 \ominus o_1, \cdots, o_k \ominus o_1, c \ominus o_1\} \subseteq S_1 \tag{7}$$

Similarly, for the arc $S_1 \xrightarrow{p_2} S_2$, we get

$$\{(p_1 \oplus p_2 \oplus p_3) \ominus p_1 \ominus p_2, \cdots, (p_1 \oplus p_2 \oplus \cdots \oplus p_k) \ominus p_1 \ominus p_2, c \ominus p_1 \ominus p_2\} \subseteq S_2 \tag{8}$$

To see how each element on the L.H.S. of Equation 8 belong to $S_2$, rearrange Equation 8 as follows, and apply the arguments that each offset $o_i$ (as well as $c$) is compatible with $o_2 (= p_1 \oplus p_2)$.

$$\{(p_1 \oplus p_2 \oplus p_3) \ominus (p_1 \oplus p_2), \cdots, (p_1 \oplus p_2 \oplus \cdots \oplus p_k) \ominus (p_1 \oplus p_2), c \ominus (p_1 \oplus p_2)\} \subseteq S_2 \tag{9}$$

$$i.e., \quad \{(o_3 \ominus o_2), \cdots, (o_k \ominus o_2), c \ominus o_2\} \subseteq S_2 \tag{10}$$

[4] Note that, we consider the latencies between the offsets of the two initiations, rather than the latencies between actual time of initiations. Under modulo scheduling, since all initiations are repeated once every **II** cycles, the difference between the offsets suffices.

Proceeding this way, we can show that

$$\{(p_1 \oplus p_2 \oplus \cdots \oplus p_k) \ominus (p_1 \oplus p_2 \oplus \cdots \oplus p_{k-1}), c \ominus (p_1 \oplus p_2 \oplus \cdots \oplus p_{k-1})\} \subseteq S_{k-1} \tag{11}$$

and

$$\{c \ominus (p_1 \oplus p_2 \oplus \cdots \oplus p_k)\} \subseteq S_f \tag{12}$$

This means that $S_f$ is non-empty which contradicts the definition of the final state. Hence our assumption $c$ is compatible with all elements of $O$ must be wrong. Thus, $O$ is a maximal compatible class. □

Next we state and prove the converse of the above lemma.

**Lemma 5.2** *For each maximal compatibility class $C$ of permissible offsets, there exists a path in the MS-state diagrams whose offset set $O$ is equal to $C$.*

**Proof:** Consider a compatible class $C = \{c_0, c_1, c_2, \cdots c_k\}$ of $\mathcal{O}$. Without loss of generality, let the offsets be in the ascending order. Further, since $C$ is maximal, it includes 0. Therefore $c_0 = 0$. We prove this Lemma by constructing a path,

$$S_0 \xrightarrow{p_1} S_1 \xrightarrow{p_2} S_2 \cdots \xrightarrow{p_k} S_f. \tag{13}$$

where

$$p_i = c_i - c_{i-1} \tag{14}$$

We will prove that this path exists in the MS-state diagram. To show this, we need to prove (i) each latency $p_i$ is a permissible and (ii) $p_i$ is an element in $S_{i-1}$. Lemma 5.3 and 5.4 establish this. Thus the path in Equation 13 is a valid path representing a legal latency sequence. By Theorem 4.1, every legal path must be in the MS-state diagram which completes the proof.

**Lemma 5.3** *The latencies $p_i$ of path shown in Equation 13 are permissible.*

**Proof:** Since the difference between any pair of elements of $C$, in particular, $c_i - c_{i-1} = p_i$ lies in $\mathcal{O}$. Further $c_i - c_{i-1} \neq 0$ as $c_i \neq c_{i-1}$. Thus each latency $p_i$ is a non-zero offset which is a permissible offset. Hence it is also a permissible latency. □

**Lemma 5.4** *For the path shown in Equation 13, the latency $p_i$ is a permissible latency (an element) in $S_{i-1}$.*

**Proof:** It can be seen that the offset set for the above path is

$$\{0, p_1, (p_1 \oplus p_2), \cdots, (p_1 \oplus p_2 \oplus \cdots \oplus p_k)\}$$

First we will show that these offset values correspond to $c_0, c_1, \cdots, c_k$ respectively. Using Equation 14, and $c_0 = 0$, we get

$$p_1 = (c_1 - c_0) = c_1. \tag{15}$$

Next, using Equation 14 in the second offset value.

$$p_1 \oplus p_2 = (c_1 - c_0) + (c_2 - c_1) = c_2 \tag{16}$$

Proceeding this way, we get

$$(p_1 \oplus p_2 \oplus \cdots \oplus p_k) = c_k \tag{17}$$

Next we will show that $p_i$ is in state $S_{i-1}$. The proof of this part is similar to the proof given in Part 2 of Lemma 5.1. Since $C$ is a subset of the permissible offsets $\mathcal{O}$, except for $c_0 = 0$ all elements of $C$ will be in the initial permissible latency set $S_0$. Thus, the initial state $S_0$ consists of $c_1$, $c_2, \cdots, c_k$. Mathematically,

$$C - c_0 \subseteq S_0 \quad i.e., \quad \{c_1, c_2, \cdots, c_k\} \subseteq S_0$$

Now substituting $c_1, c_2, \cdots, c_k$ from Equations 15 to 17, we get

$$\{p_1, (p_1 \oplus p_2), \cdots, (p_1 \oplus p_2 \oplus \cdots \oplus p_k)\} \subseteq S_0 \tag{18}$$

Thus $p_1$ is in $S_0$. Further, since $(p_1 + p_2)$ is in $S_0$ and $S_0 \xrightarrow{p_1} S_1$, from the construction of the MS-state diagram it is clear that $(p_1 + p_2) - p_1 = p_2 \in S_1$. Similarly, from Equation 18, one can say that $(p_2 \oplus p_3) \in S_1, \cdots, (p_2 \oplus p_3 \oplus \cdots \oplus p_k) \in S_1$. That is,

$$\{(p_2), (p_2 \oplus p_3) \cdots, (p_2 \oplus p_3 \oplus \cdots \oplus p_k)\} \subseteq S_1 \tag{19}$$

Now, consider the arc $S_1 \xrightarrow{p_2} S_2$. Using the above argument, we can show that

$$\{(p_3), (p_3 \oplus p_4), \cdots, (p_3 \oplus p_4 \oplus \cdots \oplus p_k)\} \subseteq S_2 \tag{20}$$

Proceeding further, we get

$$\{(p_k)\} \subseteq S_{k-1} \tag{21}$$

Hence the Lemma. □

Next we will show that the path shown in Equation 13 is primary.

**Lemma 5.5** *For each maximal compatibility class $C$ of permissible offsets, there exists a* **primary** *path in the MS-state diagrams whose offset set $O$ is equal to $C$.*

**Proof:** Lemma 5.2 establishes that there exists a path $S_0 \xrightarrow{p_1} S_1 \xrightarrow{p_2} S_2 \cdots \xrightarrow{p_k} S_k$, where $p_i = c_i - c_{i-1}$, in the MS-state diagram that supports the offsets given by the maximal compatibility class $C$. Now, to prove that this path is primary, consider $(p_1 + p_2 + \cdots + p_k)$. Substituting for each $p_i$ from Equation 14, we get

$$(p_1 + p_2 + \cdots + p_k) = c_k.$$

Since $c_k$ is permissible offset, $c_k \in \mathcal{O}$, and by the definition of offset values, $c_k < \mathbf{II}$. Hence the Lemma. □

**Theorem 5.1** *For each secondary path from $S_0$ to $S_f$ in the MS-state diagram there exists primary path such that their offset sets are equal.*

**Proof:** From Lemma 5.1, the secondary path under consideration, results in an offset set that equals a maximal compatibility class of $\mathcal{O}$. But by Lemma 5.5, for this maximal compatibility class there exists a **primary** path that supports the same offset set. Hence the theorem. □

**Theorem 5.2** *A reduced MS-state diagram consisting only of primary paths contains the set of all valid offset sets that are permissible in the original state diagram.*

Proof: Follows from Theorem 5.1. □

## 5.4 Alternative Construction Methods for Reduced MS-State Diagrams

As demonstrated in the previous subsection, it is sufficient to obtain a reduced MS-state diagram consisting only of primary paths. One method to obtain such a state diagram is by identifying secondary paths and eliminating them. This can be accomplished in two passes.

In the first pass, at the time of creation of a new state, the state diagram construction method checks whether the path leading to the newly created state is secondary. If so, it marks such a state as *redundant* and the construction of the subtrees of the state is stopped. In the second pass, the construction algorithm checks each state for redundant children and removes them. If all the children of a state are redundant, then the state itself is marked redundant. Subsequently, this state gets eliminated in the *recursive ascend*, that is, when its parent is checked for redundant children.

An alternative construction method for generating the offset sets corresponding to primary paths is based on the enumeration of maximal compatible classes. Lemmas 5.1 and 5.2 establish that each path in the MS-state diagram corresponds to a maximal compatibility class and for every maximal compatibility class there is a primary path. Hence obtaining the maximal compatible classes is an alternative way of obtaining the offset sets. An approach to obtain the compatible classes is presented in [2] (page 99). This approach is a direct way of obtaining the set of all offset sets of the reduced MS-state diagram. It should, however, be noted that the software pipelining algorithm presented in the following section is independent of the method used to obtain the set of offsets.

The following section deals with the improved Co-scheduling method that uses the reduced MS-state diagram.

# 6 Enhanced Co-Scheduling Method

In this section we detail how the enhanced Co-Scheduling[5] approach generates schedules for `FUs` with structural hazards. In Section 6.2, we compare the enhanced Co-Scheduling method with Huff's Slack scheduling method.

## 6.1 Co-Scheduling Algorithm

Enhanced Co-Scheduling was based on Huff's Slack Scheduling algorithm [8]. Both methods start with the Minimum Initiation Interval (**MII**) and attempts to schedule the loop for values of **II** $\geq$ **MII** until a schedule is found. The basic notion of Huff's original Slack Scheduling was to schedule instructions in increasing order of their *slackness*: the difference between the earliest time and the latest time at which an instruction may be scheduled. *Slack* is a dynamic measure and is updated after each instruction is scheduled. Given the earliest time (*Estart*) and the latest start time (*Lstart*) of an operation, the decision on the *search direction*, i.e., whether to attempt the placement of the operation from *Estart* or *Lstart*, is determined by what is called the *Stretchability* of the instruction [8]. Stretchability of an instruction is a measure that is similar to slack, but indicates whether the instruction stretches the lifetime of the results produced by (1) its predecessors or (2) itself. This measure helps in obtaining schedules with lower register pressure. These points remain in our enhanced Co-Scheduling.

The difference lies in how a time is chosen within the slack range. The original Slack Scheduling permitted instructions to be scheduled anywhere in their slack range, whereas in our Co-Scheduling method, an instruction is scheduled only at pre-determined offset values given by the offset sets of the reduced state diagram. Second, while the Slack Scheduling method uses the Modulo Reservation Table (MRT) to keep track of the resources committed for the scheduled operation, our Co-Scheduling method uses a *Modulo Initiation Table (MIT)*. The MIT consists of **II** columns and one row for each function unit in the architecture. Notice that the MIT does not explicitly store resource commitment for each stage of the pipeline, as is required in the case of MRT. The MIT represents only the modulo initiation time of different instructions; resource usage and conflicts are maintained through the MS-state diagram.

For a given **II**, our enhanced Co-Scheduled method first computes the offset sets corresponding to the reduced MS-state diagram for all FUs that have structural hazards. Once an instruction is picked and the search direction (say, from *Estart* to *Lstart* ) is determined, the next step is to find a cycle that is closest to *Estart* which does not cause a structural hazard. In our method, the resource constraints are represented by the set of permissible offset sets, derived from the reduced MS-state diagram. We consider only those offset sets, that have a cardinality greater than or equal to the number of operations mapped on to that function unit[6]. Sets with smaller cardinalities, need not be considered as they do not support the required number of instructions.

[5] The original Co-Scheduling discussed in [22] uses a single path and the corresponding offset set information to guide the software pipelining method. In contrast, the reduced Co-scheduling method considers all offset set of the reduced MS-state diagram.

[6] For simplicity, we consider only a single instance of FU in each FU type.

The way scheduling proceeds is better explained with the help of an example. Though the example concentrates on how resource constraints are met in a single FU, the method is general enough to handle multiple FU types. The detailed algorithm is presented in Appendix A.

Consider, a function unit with four instructions $i_1, i_2, i_3$, and $i_4$ mapped on to it. Let the offset sets for the function unit be

$$O_1 = \{0, 5, 8\}, \quad O_2 = \{0, 5, 10\}, \quad O_3 = \{0, 5, 12\};$$

$$O_4 = \{0, 1, 7, 10\}; \quad O_5 = \{0, 1, 6, 7\}; \quad O_6 = \{0, 3, 6, 9, 12\}; \quad \cdots$$

As mentioned above, we need to consider only those offset sets that support at least 4 instructions. Thus only offset sets $O_4, O_5$, and $O_6$ are considered for the given loop by the software pipelining method. These sets ($O_4, O_5$, and $O_6$) are initially the *active* offset sets.

Let us start with the scheduling of instruction $i_1$ with a slack[7] *(3,5)*. Since this is the first instruction to be scheduled in the pipe, it has no structural hazards and can be placed in any cycle in its slack. To simplify the discussion, it is assumed that the search direction for all instructions is from *Estart* to *Lstart*. Hence $i_1$ is placed at its *Estart*, 3.

Note that the offset values are relative. Thus, the first instruction is always assumed to be scheduled at an offset 0. In our example, instruction $i_1$ which is scheduled at time step 3 corresponds to an offset 0. All future initiations in this pipeline, and their offset values will be with respect to $i_1$. Now, suppose $i_2$ has a slack *(10,15)*. The *Estart* time of $i_2$ corresponds to an offset $10 - 3 = 7$ with respect to $i_1$. A look at the offset sets reveals that $7, 9, 10$, and 12 are permissible offsets. Hence $i_2$ is scheduled at time step 10 with an offset 7 with respect to $i_1$. Since $O_6$ does not support an offset 7, it is marked *inactive*; the offset sets $O_4$ and $O_5$ are currently *active*.

Now, if instruction $i_3$ has a tight slack *(12,12)* with an offset 9, neither of the offset sets $O_4$ and $O_5$ can support the scheduling of $i_3$ at time 12. In such a case, the most recently placed operation is ejected. This may increase the number of active offset sets and hence the possibility of a placement. In our example, when $i_2$ is unscheduled, the offset set $O_6$ becomes active, and $i_3$ is scheduled at time 12 (and offset 9) in $O_6$. Scheduling $i_3$ at time 12 makes offset sets $O_4$ and $O_5$ inactive, leaving $O_6$ as the only active set. Subsequently, when $i_2$ is chosen for scheduling[8], it is scheduled at time step 15, with offset 12. In a similar manner, if instruction $i_4$ has a slack *(19,26)*, it can be placed at one of the remaining offsets, 3 or 6. If no valid schedule is found even after ejecting a number of operations (greater than a *threshold* value), the current (partial) schedule is aborted and successive values of **II** are tried until a valid schedule is obtained.

[7] Throughout this paper we assume that all slack ranges are inclusive of both extreme points. Thus, in this case, the slack includes both 3 and 5.

[8] For simplicity, assume that $i_2$'s slack does not change.

### 6.2 Remarks

Our method differs from Huff's approach in three aspects: As discussed earlier, Huff's method uses MRT to represent the resource usage while Co-Scheduling uses MIT to represent the initiation time of scheduled instructions. Second, the Slack Scheduling method attempts to schedule an instruction in every cycle in the slack range. Whereas, with enhanced Co-scheduling, only time steps that correspond to chosen offset values in the slack range are tried. This results in fewer number of trials per operation. Also we check only those offset sets, which support the required number of initiations. This avoids getting caught in or trying a wrong offset set which cannot support the required number of initiations in the pipe. Lastly, in forcing the placement of an operation, Huff's method ejects only conflicting operations; whereas in our approach, the operations scheduled in a pipe are ejected in the reverse order in which they are scheduled. Though it is possible in our method to eject only the conflicting operation, we chose the reverse scheduling order for ejection to avoid getting trapped in a specific offset set.

## 7 Experimental Results

In this section, first we present a quantitative comparison of the reduced MS-state diagram and the original MS-state diagram. In the subsequent subsections we present the performance of the enhanced Co-scheduling algorithm. In Section 7.4 we compare enhanced Co-scheduling with Huff's Slack Scheduling method. Section 7.6 provides a summary of the experimental results.

### 7.1 Reduced MS-state Diagram

We compare the reduced MS-state diagram with the original state diagram in terms of the number of paths. We have implemented the construction of the original state diagram (Procedure 4.2 in Section 4.3) and the reduced MS-state diagram (described in Section 5.4). Using these implementations, the reduced and original state diagrams have been constructed for a set of 6 function units, typical of a modern day processor, for a range[9] of **II** from 8 to 24. The reservation tables for the FUs are shown in Appendix B. Table II shows the average reduction in the number of paths achieved by considering only the primary paths, the average being taken over different values of **II** considered in the given range. For small values of **II**, i.e., less than 16, the the reduction in number of paths varies from 2 to 26 in the case of geometric mean, and 2 to 60 in the case of arithmetic mean. However, for the range of **II** between 16 to 24, the reduction in number of paths varies from 32 to 9,084 in the case of geometric mean, and 55 to 27,543 for arithmetic mean.

[9] For values of **II** greater than 24, the number of paths in the original state diagram exceeds 10 Millions and all paths could not be enumerated within 30 minutes of CPU time. Hence, in this study we limited **II** to a maximum of 24. However, the reduced MS-state diagram can be constructed even for large values of **II** as can be seen from Table IV(b).

| **Avg. Reduction** | $8 \leq$ **II** $\leq 15$ | | | | | | $16 \leq$ **II** $\leq 24$ | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **in No. of Paths** | FU-1 | FU-2 | FU-3 | FU-4 | FU-5 | FU-6 | FU-1 | FU-2 | FU-3 | FU-4 | FU-5 | FU-6 |
| **Geo. Mean** | 7.5 | 20.8 | 2.5 | 2.6 | 26.1 | 2.2 | 1037.3 | 9084.3 | 54.3 | 52.3 | 2273.9 | 32.7 |
| **Arith. Mean** | 13.5 | 60.7 | 3.0 | 3.4 | 64.7 | 2.5 | 3882.5 | 27543.7 | 99.5 | 95.3 | 22490.7 | 55.5 |

Table II: Average Reduction in the Number of **Paths**.

For function units FU-3, FU-4, and FU-6, the reduction in the number of paths is not so significant for the values of **II** considered in our experiments. This is because, at small **II** values, many of the latencies were forbidden for these FUs. It is expected that for larger values of **II**, the reduction in number of paths will be significant. This can be seen from the rate of increase of the number of paths in Figure 5. We plot the number of paths in the original and reduced state diagrams for two function units (FU-1 and FU-3) for values of **II** from 8 to 24. Note that the y-axis (number of paths) is plotted on a log-scale. We observe that the number of paths in the original state diagram increases exponentially with **II**; whereas, the rate of increase in the reduced state diagram is rather steady. The number of offset sets in the reduced state diagram (or equivalently the number of primary paths) is only few hundreds.

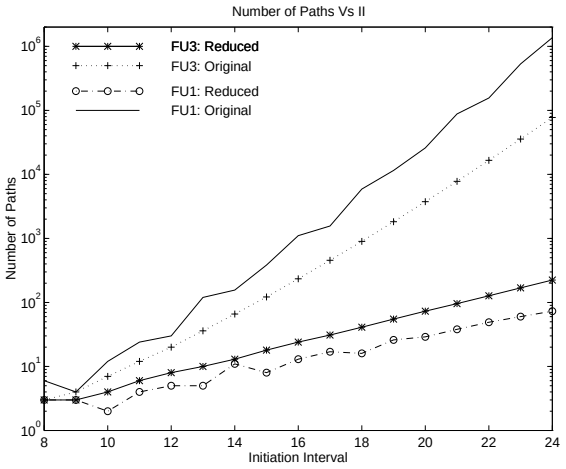



Figure 5: Increase in Number of Paths for Different **II**s

## 7.2 Maximal Compatible Set Generation

As discussed in Section 5.4, generating the maximal compatible classes of the permissible offsets $\mathcal{O}$ is an alternative way of obtaining the set of offset sets corresponding of a reduced MS-state diagram. Using the algorithm given in [2], we have implemented a method to generate all maximal compatible classes of the permissible offsets. For the same set of reservation tables used in the previous section, and for the same **II** values, we generated

the offset sets using the maximal compatible classes. The execution time, on an UltraSparc 170E, were compared with those of the reduced MS-state diagram approach. The average execution time, averaged over runs for different values of **II**, for each FU results are shown in Table III. Obtaining the offset sets from the maximal compatible classes results in a 3-fold improvement in execution time for the values of **II** considered in our experiments.

| Approach | **Avg. Exec. Time** | | | | | |
|---|---|---|---|---|---|---|
| | FU-1 | FU-2 | FU-3 | FU-4 | FU-5 | FU-6 |
| | (in milliseconds) | | | | | |
| Reduced State Diagram (Arith. Mean) | 277.7 | 1025.2 | 80.8 | 2933.8 | 138.2 | 17.6 |
| Maximal Compatible Class (Arith. Mean) | 70.7 | 146.1 | 93.1 | 519.4 | 74.5 | 7.0 |
| Improvement in Exec. Time (Geom. Mean) | 6.9 | 5.3 | 1.9 | 3.8 | 3.1 | 3.7 |

Table III: Execution Time of Two Methods to Generate Offset Sets

## 7.3 Performance of Enhanced Co-Scheduling Method

We have implemented the enhanced Co-scheduling method presented in Section 6, and tested it on 1153 loops taken from a variety of benchmarks : `specfp92, specint92, livermore, linpack, and NAS kernels`. We assumed a target architecture with 7 function units: 2 Integer Units, 1 Load Unit, 1 Store Unit, 1 FP Add Unit, 1 FP Multiply Unit, and 1 FP Divide Unit. Except for the Integer and Store Units, the resource usage of other instruction classes were assumed to involve structural hazards. Their reservation tables are shown in Appendix B. Of these, the reservation tables for the FP Multiply and FP Add Units correspond to those of FU-5 and FU-6, respectively, used in Sections 7.1 and 7.2. The results of our experiments are shown in Table IV.

Table IV(a) gives a break up of the total benchmark programs in terms of how far the **II** of the constructed schedule is from the minimum initiation interval (**MII**). Our enhanced Co-scheduling found schedules at the minimum initiation interval in 880 cases. In the remaining cases, **II** of the resulting schedules were 2.58 time steps away, on an average, from the minimum initiation interval (refer to Table IV(b)).

The (arithmetic) mean time to compute a schedule is 2.9 milliseconds, while the median for this is 1.1 milliseconds. Table IV(b) also gives other statistics on the performance of our enhanced Co-scheduling. Execution times, on an UltraSparc-170E, are reported in the last row of the table shown in Table IV(b). In reporting the execution time, the time to construct reduced MS-state diagrams for different function units is not included. This is because, the generation of the offset sets can be done off-line, and stored in a database. As mentioned in the Introduction, this comes at the cost of increased space requirements. The average time to construct the state diagram for a function unit is in the order of a few hundred milliseconds as shown in Table III. Lastly,

| **II – MII** | No. of Bench-marks | %–age Cases |
|---|---|---|
| 0 | 880 | 76.3 |
| 1 | 144 | 12.5 |
| 2 | 20 | 1.7 |
| 3 | 17 | 1.5 |
| 4 | 15 | 1.3 |
| 5 | 61 | 5.3 |
| $\geq 6$ | 16 | 1.4 |

**(a)**

| Measure | Min. | Max. | Arith.Mean | Geo.Mean | Median |
|---|---|---|---|---|---|
| No. of Nodes | 1 | 52 | 6.4 | 5.7 | 6.0 |
| **II** | 1 | 85 | 9.1 | 7.0 | 6.0 |
| **II - MII** | 0 | 15 | 2.58 | 2.0 | 1.0 |
| **II/MII** | 1 | 3 | 1.1 | 1.0 | 1.0 |
| Time(msec) | 0.24 | 304 | 2.9 | 2.0 | 1.1 |

**(b)**

Table IV: Performance of Enhanced Co-Scheduling

even though enhanced Co-scheduling uses the set of all offset sets, typically several hundred in number, it still was successful in finding a schedule within a few milliseconds.

## 7.4 Comparison with Huff's Scheduling Method

The enhanced Co-scheduling method is compared with our implementation of Huff's Slack Scheduling method[10]. Though comparison with other modulo scheduling methods could have been made, we chose Huff's method for comparison for the following reasons: (i) Enhanced Co-scheduling is based on Huff's method, and therefore a comparison with it would directly reveal the impact of the Co-scheduling approach and the use of reduced state diagrams in the software pipelining method; (ii) Huff's Slack Scheduling method has widely been accepted to result in better schedules in shorter execution time; (iii) Lastly, Huff's method is life-time sensitive and hence attempts to reduce the register pressure of the software pipelined schedule.

The results of our comparison are presented in Table V. As seen from Table V our enhanced Co-scheduling results in better **II** in 114 benchmarks; the average improvement in **II** is 13.5%. In a large number of cases (993 benchmarks) both methods achieved the same **II**. This is because, for the target architecture considered for the scheduling, only one-fourth (24%) of the loops are *resource-critical* — i.e., resource MII (**ResMII**) dominates recurrence MII (**RecMII**). Since Co-scheduling is basically Slack scheduling, fine-tuned for better selection of offset values, it is not surprising that the improvement, in terms of **II**, happens only in resource-critical loops. Also, it is possible that in some of the resource-critical loops both methods have achieved the **MII**. In a small number of benchmarks, Huff's Slack Scheduling achieves a better **II**, though the percentage improvement is only

[10] To the best of our knowledge, our implementation of Huff's Slack Scheduling method faithfully follows the implementation details presented in [8].

minor (1.6%). This could be due to the order in which we eject the instructions in the enhanced Co-scheduling method.

| Measure | Enhanced Co-Scheduling Better | | | Huff Better | | | Both Same | |
|---|---|---|---|---|---|---|---|---|
| | No. of Benchmarks | % Cases | % Improvement | No. of Benchmarks | % Cases | % Improvement | No. of Benchmarks | % Cases |
| **II** | 114 | 10 | 13.5 | 40 | 3 | 1.6 | 993 | 87 |
| Avg. Trials per instrn. | 662 | 58 | 560.1 | 5 | 0.1 | 322.4 | 480 | 42 |
| Avg. Ejections per instrn. | 338 | 29 | 858.7 | 79 | 7 | 858.0 | 730 | 63 |
| Exec. Time | 988 | 86 | 457.9 | 158 | 14 | 427.6 | 1 | 0.0 |

Table V: Comparison of Enhanced Co-Scheduling with Slack Scheduling

The results shown in Table V exhibit a significant reduction in execution time (time to construct schedules). In 988 of the benchmarks programs, the execution time of our Co-scheduling method is lower than that of Slack Scheduling. The average improvement in execution time is roughly 4.5 fold. It should be noted here that in the above comparison the execution time of our enhanced Co-scheduling method did not include the time to construct the MS-state diagram. If the construction time had been included, Huff's Slack Scheduling would have lower scheduling time in many of the loops. Nevertheless, the enhanced Co-Scheduling method enjoys a performance advantage in terms of **II** in nearly 114 loops.

Apart from **II** and execution times, we compare the two methods in terms of two other measures, namely (i) average number of trials per operation, and (ii) average number of ejections per instruction. In Huff's Slack Scheduling, as explained in Section 6.2, all time steps in the slack range of an instruction are tried successively, until a (resource) conflict-free time slot is found. Whereas, in our method, we need to consider only those time steps in the slack range that correspond to the offset values of the *active* offset sets. Hence the number of such trails is expected to be much less in our case compared to Slack Scheduling. From Table V, we observe that our enhanced Co-scheduling method performs better in more than 58% of the benchmarks, with an average improvement of 560%; both methods perform equally in 480 cases. Fewer ejections per instructions was observed in Slack Scheduling only in 5 benchmarks.

Second, in forcing an operation, our approach differs from Slack Scheduling in the order in which it ejects the operations. Hence, we compared the average number of ejections per instruction for these two methods. Our method performed better in more than 30% of the benchmarks yielding an 8-fold improvement. The reduction in the average number of ejected operations and the average number of trials achieved by our enhanced Co-scheduling method, in turn, results in an improved execution time as well. Lastly, even though Huff's Slack Scheduling achieves comparable improvement in performance, in terms of average ejections, trials, and scheduling

time, the number of benchmarks where the improvement is observed is significantly lower (respectively 79, 5, and 158) benchmark programs.

## 7.5 Heuristic Methods for Reducing Number of Offset Sets

In the enhanced Co-scheduling, the initial number of offset sets in the active set is a few hundreds (Refer to Figure 5). Thus it is an expensive proposition to consider all offset sets in the enhanced Co-scheduling. In this section, we study the effect of reducing the initial number of offset sets used in the enhanced Co-scheduling method.

We have experimented with the following three heuristics, that selects as few as 5 offset sets from the set of all offset sets.

**Rand:** Pick any five offset sets randomly, including the one with largest cardinality.

**Pick:** Starting with the offset set with the maximum cardinality, five offset sets are picked, one at a time, such that the offset set considered at a step differs from the union of picked offset sets by at least certain percentage. The aim of this heuristic is to consider 5 offset sets, the union of which is close to the set of all offset values.

**PickCard:** Similar to **Pick** but each of the offset sets, if possible, has a different cardinality.

The enhanced Co-scheduling was modified to use only 5 offset sets, chosen using one of the above three heuristics. These variants of enhanced Co-scheduling (**ECS**) are called **Rand-ECS**, **Pick-ECS** and **PickCard-ECS** respectively. The performance of these are compared with both the **ECS** (that uses all the offset sets) and Huff's slack scheduling method. All three methods (**Rand-ECS**, **Pick-ECS** and **PickCard-ECS**) match the performance of enhanced Co-scheduling in 92% of the cases. In the remaining 8% of the cases, they perform worse than enhanced Co-scheduling as well as Huff's slack scheduling. This degradation in 8% of the benchmark loops indicates that some of the offset sets, which would have produced a valid schedule at a lower value of **II**, were left out in the pick. Also, in the 10% of the cases, where enhanced Co-scheduling was doing better than Huff's method, the heuristic methods continue to perform better. The summary of results for the three methods is reported in Table VI.

Lastly, we investigate the effect of the number of offset sets considered by the heuristic methods. The variation in performance as the number of offset sets considered is plotted in Figure 6 for **RandECS**.

## 7.6 Summary of Results

To summarize, the major results observed in our experiments are:

| Heuristic | With ECS | | | | With Huff | | | |
|---|---|---|---|---|---|---|---|---|
| | ECS better | | Heuristic ECS better | | Huff better | | Heuristic ECS better | |
| | % Cases | % Improvement | % Cases | % Improvement | % Cases | % Improvement | % Cases | % Improvement |
| **Rand** | 7 | 2 | 0 | 0 | 10 | 3.9 | 10 | 11.6 |
| **Pick** | 8 | 1.7 | 0 | 0 | 11 | 3.53 | 10 | 11.63 |
| **PickCard** | 8 | 2 | 0 | 0 | 11 | 3.73 | 10 | 11.7 |

Table VI: Comparison of Heuristic Methods with ECS and Slack Scheduling

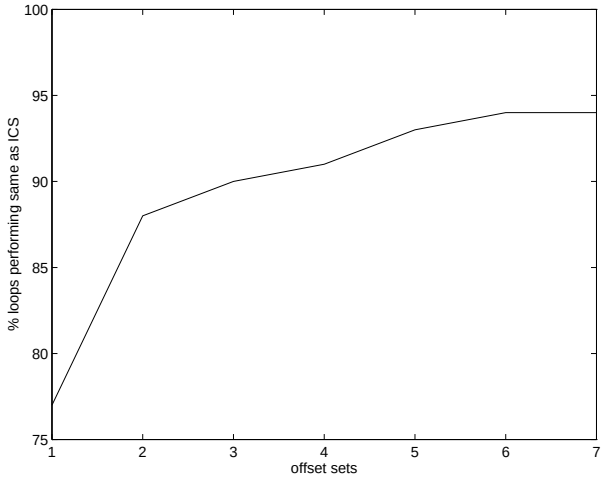



Figure 6: Performance Variation With the Number of Offset Sets

- The number of paths in the reduced state diagram is significantly lower; the reduction in number of paths is by a factor of 2 to 20 for values of **II** less than 16, and 32 to 9,084 for larger values of **II**, on the average (refer to Table II).
- Obtaining the set of offsets using the maximal compatible classes is faster, in terms of execution time, at least for smaller values of **II** (refer to Table III).
- Comparison with Huff's Slack Scheduling [8] reveals that our enhanced Co-scheduling performs better, in terms of **II**, in 114 loops, a majority of which are resource-critical loops; in terms of scheduling time, it performs better in as many as 988 loops with a 457% average improvement. The improvement in **II** achieved by Slack Scheduling is in fewer cases (40 loops); average improvement is also lower (1.6%). In terms of scheduling time, Slack scheduling has better performance only in 158 cases (14%).

# 8 Extension to Pipelines with Shared Resources

The theory of reduced MS-state diagram and the proposed enhanced Co-scheduling method presented in this paper concentrated only on architectures where two function units or instruction classes do not share any resource. In this section we briefly outline how it is possible to extend our approach to function units sharing resources.

| Stage | Time Steps | | | |
|---|---|---|---|---|
| | 0 | 1 | 2 | 3 |
| Decode | x | | | |
| FP-Add | | x | | |
| FP-Mult | | | | |
| Write-Back | | | x | |

(a) Reservation Table for **FP Add**

| Stage | Time Steps | | | |
|---|---|---|---|---|
| | 0 | 1 | 2 | 3 |
| Decode | x | | | |
| FP-Add | | | | |
| FP-Mult | | x | x | |
| Write-Back | | | | x |

(b) Reservation Table for **FP Multiply**

Figure 7: Reservation Tables of Pipelines with Shared Resources

To make the discussion simple, consider that an **FP Add** unit and an **FP Multiply** unit share the *Decode* and *Write-back* stages of the pipeline as shown in the reservation tables of Figure 7. It is possible to extend the construction of MS-state diagram for the above pipelines in a manner similar to the construction of state diagram for multi-function pipelines as discussed in [2, 20, 21]. We extend the initial permissible latency set to permissible latency matrices[11] which define the permissible latencies for different instruction classes. For example, for the above reservation tables and for an **II**= 4, we define the two permissible matrices

$$P_{FP\_Add} = \begin{bmatrix} 1, 2, 3 \\ 1, 2 \end{bmatrix} \quad \text{and} \quad P_{FP\_Mult} = \begin{bmatrix} 2, 3 \\ 2 \end{bmatrix}.$$

The latencies in the first row of $P_{FP\_Add}$ indicate the permissible latencies between two **FP-Add** instructions while the latencies in the second row indicate the permissible latencies between and **FP-Add** and a subsequent **FP-Mult**. Similarly the latencies shown in the first and second row of $P_{FP\_Mult}$ indicate permissible latencies between an **FP-Mult** and a subsequent **FP-Add** and between two **FP-Mult** instructions. One can find a similarity between these definitions and the definition of *collision matrices* in [2, 20, 21].

The construction of the MS-state diagram proceeds as follows. If an instruction type $i$ (*e.g.,* **FP-Mult**) with a latency $p$ is permissible from the current state $S$ (*i.e.,* $p$ is present in the $i$th row of the current state (represented by permissible matrix)), then the new state $S'$ is obtained by subtracting $p$ (subtraction modulo

[11] We use the term matrices in a *loose* sense here even though the number of elements in different rows may not be equal. However, a collision matrix representation [2] will have exactly **II** bits in each row. Hence we continue to call this a permissible matrix.

**II**) from each element of the permissible matrix $\boldsymbol{S}$ and taking the intersection, on a row-by-row basis[12], with the initial permissible matrix for instruction class $\boldsymbol{i}$. Due to lack of space and for simplicity sake, we only show a few paths in the MS-state diagram. As an example, following is a path in the MS-state diagram for shared resources.

$$\begin{bmatrix} 1,2,3 \\ 1,2 \end{bmatrix} \xrightarrow{\text{1–FP-Add}} \begin{bmatrix} 1,2 \\ 1 \end{bmatrix} \xrightarrow{\text{1–FP-Mult}} \begin{bmatrix} \quad \\ \quad \end{bmatrix}$$

Note that in the above notation, we indicate the type of instruction initiated, in addition to the latency, as in 1–FP-Add. The above path corresponds to an offset set {0–FP-Add, 1–FP-Add, 2–FP-Mult}. One can verify that a path

$$\begin{bmatrix} 1,2,3 \\ 1,2 \end{bmatrix} \xrightarrow{\text{2–FP-Mult}} \begin{bmatrix} 3 \\ \quad \end{bmatrix} \xrightarrow{\text{3–FP-Add}} \begin{bmatrix} \quad \\ \quad \end{bmatrix}$$

is a secondary path (sum of the latency values exceeds **II**= 4) which also corresponds to the offset set {0–FP-Add, 1–FP-Add, 2–FI
It can be shown that for every secondary path in the above MS-state diagram there is a primary path with the same offset set. This shows that the theory of reduced MS-state diagram is still applicable to pipelines with shared resources.

It is now straightforward to see how one can use these offset sets in our enhanced Co-scheduling method to schedule a **FP-Add** or **FP-Mult** instructions. However, a number of issues are still open. Though secondary paths in the above MS-state diagram are redundant, it can be shown, by examples, that there can be more than one primary path that correspond to the same offset set. How does one generate the *distinct* offset sets? Further, the number of *distinct* paths in the above MS-state diagram can be quite large, even for moderate values of **II**. How does one handle this complexity and use (possibly, a subset of) the offset sets for constructing the software pipelined schedule?

Some of these questions are addressed in a recent work [26] where efficient state diagram construction methods have been proposed for pipelines with shared resources. This work proposes two approaches, one based on a heuristic and the other based on graph theoretic methods, for the construction of offset sets. In particular, the heuristic method prevents all redundant paths in the MS-state diagram for pipelines with shared resources. But, the proposed heuristic may result in missing some offset sets. However, as the number of offset sets in these state diagrams are more than a few million, it is claimed that the loss of a few offset sets may not degrade the performance of the constructed software pipelined schedule. The proposed heuristic and the graph theoretic methods are applied for generating offset sets for two real processors, namely the DEC Alpha 21064 superscalar processor and the Cydra VLIW processor [26].

[12] A detailed explanation of the construction procedure is beyond the scope of this paper. The purpose of this discussion is to quickly show how our ideas can be extended to pipelines with shared resources.

# 9 Related Work

Several software pipelining methods have been proposed in the literature [3, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]. These methods are based on a global modulo reservation table. A comprehensive survey of these works is provided by Rau and Fisher in [17]. As explained in the Introduction, one major drawback of these methods is their inefficiency. These methods make several trials before successfully placing an operation in the modulo reservation table. They do not make effective use of the well-developed classical pipeline theory [1, 2, 18].

The approach proposed in [19, 20, 21] uses a finite state automaton (FSA)-based instruction scheduling technique. These methods use ideas from the classical pipeline theory, especially the notion of forbidden and permissible latency sequences. However these methods deal with general instruction scheduling, and do not handle software pipelining, where each scheduled instruction is initiated once every **II** cycles. The Co-scheduling framework proposed in [22] is a state diagram-based software pipelining method. In the Co-scheduling method, a single permissible latency sequence chosen from the MS-state diagram and the corresponding offset values were used to guide the software pipelining method. Co-scheduling is different from pipelines scheduled at (fixed) latency cycles [1, 2, 27]. The periodicity of the latter depends only on the resource usage of the pipeline, while the periodicity of MS-pipelines discussed in this paper is governed by both the resource usage and the recurrences in the loop considered for scheduling.

In this paper, we have extended the original Co-scheduling method, by considering multiple latency sequences from the MS-state diagram. We develop the necessary theory to consider only the primary paths in the state diagram that yields distinct set of offset values. Considering only the primary path reduces the number of paths in the reduced MS-state diagram significantly. Using the reduced MS-state diagram, and the set of offset values corresponding to primary paths, we have enhanced the Co-scheduling algorithm. The enhanced Co-scheduling method attempts to address a major problem of the original Co-scheduling method, viz., the original Co-scheduling method, restricted to a single latency sequence (and a corresponding offset set), can do worse than Huff's slack scheduling. This can happen if the latency sequence chosen by the original Co-scheduling method does not "fit" well with the slacks of the instructions. The enhanced Co-scheduling method alleviates this problem by considering multiple latency sequences. Our experiments have shown that except in a small number of loops (less than 3% of the benchmarks), the performance of the enhanced Co-scheduling is equal or better than Huff's Slack Scheduling. In this paper we have compared the performance of the enhanced Co-Scheduling method only with Huff's Slack Scheduling. We chose Slack Scheduling both because it is known to perform well and because we felt it was representative of other approaches — particularly in that it was originally designed primarily for use with clean pipelines. Our method could have been compared with a number of other modulo scheduling methods based on modulo reservation table [5, 7, 9, 10, 12, 15, 16].

Lastly, Eichenberger and Davidson proposed a method, which still relies on the use of global resource table, but reduces the cost of structural-hazard checking by reducing the machine description [28]. Their approach uses

forbidden latency information to obtain a minimal representation for individual reservation tables of different function units. Their method is applicable to both general instruction scheduling and modulo scheduling.

# 10 Conclusions

In this paper we have proposed Modulo-Scheduled (MS) pipeline theory, a theory for hardware pipeline structures operating under modulo scheduling. The proposed theory extends classical pipeline theory and identifies distinct paths in the MS-state diagram through the use of primary paths. We have established the relationship between offset sets corresponding to primary paths and the maximal compatible classes of permissible offset sets. Further, it was shown that the reduced MS-state diagram consisting only of *primary paths* reduces the number of paths drastically; the improvement is by a factor of 2 to 26 for values of **II** less than 16, and 32 to 9,084 for larger values of **II**.

We have presented an enhanced Co-scheduling method that makes use of the reduced MS-state diagram. The enhanced Co-scheduling method considers multiple latency sequences from the reduced MS-state diagram to guide the software pipelining method. The use of reduced state diagrams and multiple offset sets is expected to result in schedules with smaller **II** as well as arriving at them with shorter execution (scheduling) time.

Our implementation of the enhanced Co-scheduling was tested on a set of of 1153 loops taken from various benchmark suites. The enhanced Co-scheduling was successful in scheduling 86% of the loops at their minimum initiation interval. In the remaining loops, the schedules were 2.58 time steps away, on an average, from the the minimum initiation interval. Comparison with Huff's Slack Scheduling [8] reveals that our enhanced Co-scheduling performs better, in terms of **II**, in 114 loops, a majority of which are resource-critical loops; in terms of scheduling time, it performs better in as many as 988 loops with a 457% average improvement. Lastly, we have extended our method to consider a small number of (as few as 5) offset sets; yet the method achieves 92% of the performance potentials of the enhanced Co-scheduling method (which considers *all* offset sets). Extension of our work to function units which share resources is also discussed.

## Acknowledgments

We wish to thank Sean Ryan, Chihong Zhang, members of CAPSL, University of Delaware for their helpful suggestions. The last author acknowledges the support of National Science Foundation, U.S.A. Lastly, the authors are thankful to the anonymous reviewers for their constructive comments.

# References

[1] 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, FL, Jan. 19–21, 1976.

[2] P. M. Kogge. *The Architecture of Pipelined Computers*. McGraw-Hill Book Co., New York, NY, 1981.

[3] 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 Workshop*, pages 183–198, Chatham, MA, Oct. 12–15, 1981.

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

[5] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proc. of the SIGPLAN '88 Conf. on Programming Language Design and Implementation*, pages 318–328, Atlanta, Georgia, June 22–24, 1988.

[6] A. Aiken and A. Nicolau. A realistic resource-constrained software pipelining algorithm. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, Research Monographs in Parallel and Dist. Computing, chapter 14, pages 274–290. Pitman Publishing and the MIT Press, London, England, and Cambridge, MA, 1991.

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

[8] R. A. Huff. Lifetime-sensitive modulo scheduling. In *Proc. of the ACM SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pages 258–267, Albuquerque, NM, June 23–25, 1993.

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

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

[11] A. E. Eichenberger, E. S. Davidson, and S. G. Abraham. Minimum register requirements for a modulo schedule. In *Proc. of the 27th Ann. Intl. Symp. on Microarchitecture*, pages 75–84, San Jose, CA, Nov. 30–Dec.2, 1994.

[12] B. R. 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, CA, Nov. 30–Dec.2, 1994.

[13] R. Govindarajan, E. R. Altman, and G. 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, CA, Nov. 30–Dec.2, 1994.

[14] E. R. Altman, R. Govindarajan, and G. R. Gao. Scheduling and mapping: Software pipelining in the presence of structural hazards. In *Proc. of the ACM SIGPLAN '95 Conf. on Programming Language Design and Implementation*, pages 139–150, La Jolla, CA, June 18–21, 1995.

[15] J. Llosa, M. Valero, E. Ayguadé, and A. González. Hypernode reduction modulo scheduling. In *Proc. of the 28th Ann. Intl. Symp. on Microarchitecture*, pages 350–360, Ann Arbor, MI, Nov. 29–Dec.1, 1995.

[16] J. Llosa, A. González, M. Valero, and E. Ayguadé. Swing modulo scheduling: A lifetime sensitive approach. In *Proc. of the Intl. Conf. on Parallel Architectures and Compilation Technique Microarchitecture*, pages 80–86, Boston, MA, Oct. 1996.

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

[18] E.S. Davidson, L.E. Shar, A.T. Thomas, and J.H. Patel. Effective control for pipelined computers. In *Digest of Papers, 15th IEEE Computer Society International Conference, COMPCON Spring '75*. February 1975.

[19] T. Muller. Employing finite state automata for resource scheduling. In *Proceedings of the 26th Ann. Intl. Symp. on Microarchitecture*, Austin, TX, December 1–3, 1993.

[20] T. A. Proebsting and C. W. Fraser. Detecting pipeline structural hazards quickly. In *Conf. Record of the 21st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 280–286, Portland, OR, Jan. 17–21, 1994.

[21] V. Bala and N. Rubin. Efficient instruction scheduling using finite state automata. In *Proc. of the 28th Ann. Intl. Symp. on Microarchitecture*, pages 46–56, Ann Arbor, MI, Nov. 29–Dec.1, 1995.

[22] R. Govindarajan, E. R. Altman, and G. R. Gao. Co-scheduling hardware and software pipelines. In *Proc. of the Second Intl. Symp. on High-Performance Computer Architecture*, pages 52–61, San Jose, CA, Feb. 3–7, 1996.

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

[24] S. Ramakrishnan. Software pipelining in PA-RISC compilers. *Hewlett-Packard Jl.*, pages 39–45, June 1992.

[25] R. Govindarajan, N.S.S. Narasimha Rao, E. R. Altman, and G. R. Gao. An enhanced Co-scheduling method using reduced MS-state diagrams. In *Proc. of the Merged 12th Intl. Parallel Processing Symp. and 9th Intl. Symp. on Parallel and Distributed Processing*, Orlando, FL, Apr. 1998.

[26] C. Zhang, R. Govindarajan, S. Ryan, and G. R. Gao. Efficient state-diagram construction methods for software pipelining. In *Proc. of the 8th Intl. Conf. on Compiler Construction* Amsterdam, The Netherlands, Mar. 1999.

[27] J.K. Chaar and E.S. Davidson. Cyclic job shop scheduling using collision vectors. Technical Report CSE-TR-169-93, University of Michigan, Ann Arbor, MI., Aug. 1993.

[28] A.E. Eichenberger and E.S. Davidson. A reduced multipipeline machine description that preserves scheduling constraints. In *Proc. of the ACM SIGPLAN '96 Conf. on Programming Language Design and Implementation*, Phil., PA, May 21–24, 1996.

# A Enhanced Co-Scheduling Algorithm

In this section we present the enhanced Co-scheduling algorithm in a more formal way. Though we use the reduced state diagram approach for constructing the set of *offset* sets, it can be easily replaced by the maximal compatibility classes method.

**Procedure A.1: The Enhanced Co-scheduling Method**

**Step 1** Set **II= MII**

**Step 2** While (not a valid schedule found)

**Step 2.1** For each instruction class $\mathcal{I}$

**Step 2.1.1** Generate the offset sets of the primary paths.

**Step 2.1.2** Determine the set of offset sets $\mathcal{O}(\mathcal{I})$ that support at least $N(\mathcal{I})$ instructions, where $N(\mathcal{I})$ represents the number of instructions in the given loop that are executed in this FU type. $\mathcal{A}(\mathcal{I}) = \mathcal{O}(\mathcal{I})$ is the set of active offset sets for instruction class $\mathcal{I}$.

**Step 2.1.3** If there are no offset sets supporting $N(\mathcal{I})$ initiations, increment **II** by 1; Goto Step 2.1.

**Step 2.2** While there exists an unscheduled instructions, repeat Steps 2.2.1 to Step 2.2.5

**Step 2.2.1** If the total number of ejected instructions exceed some threshold value (say `THRESHOLD_ON_TOTAL_EJECTED`) then increase **II** by 1; Discard the partial schedule and go back to Step 2.1.

**Step 2.2.2** Compute the slack and priority of the unscheduled instructions.

**Step 2.2.3** Choose the instruction $i$ with the highest priority. Let $i$ be in the instruction class $\mathcal{I}$.

**Step 2.2.4** Attempt to schedule the instruction at a time step in its slack range. The chosen time step must correspond to an offset value supported by at least one of the offset sets in $\mathcal{A}(\mathcal{I})$. Remove the offset sets that do not support the chosen offset value from $\mathcal{A}(\mathcal{I})$.

**Step 2.2.5** If there are no paths in $\mathcal{A}(\mathcal{I})$ that support any of the offset value in the slack range,

**Step 2.2.5.1** Unschedule the last instruction of this instruction class. Suppose the unscheduled operation was scheduled at an offset $o'$.

**Step 2.2.5.2** Add the offset sets that got excluded from the active set, because they could not support $o'$. (This somewhat corresponds to backtracking on the MS-state diagram to a previous level. )

**Step 2.2.5.3** Increment the number of ejected operations by 1.

**Step 2.2.5.4** Go back to Step 3.4; i.e. attempt to schedule the current instruction. (This will eventually succeed, because when the algorithm backtracks to the root of the MS-state diagram, it should be possible to schedule the instruction, as it is the first one initiated in this pipe. )

**Step 2.2.6** End /* end of If */

**Step 2.3** End /* end of while */

**Step 3** End /* end of while */

# B Reservation Tables used in Software Pipelining Methods

| | Time Steps | | | | |
|---|---|---|---|---|---|
| | *0* | *1* | *2* | *3* | *4* |
| **Stage 1** | x | | | | |
| **Stage 2** | | x | | | |
| **Stage 3** | | | | x | |
| **Stage 4** | | | | | x |

**Integer** ALU's

| | Time Steps | | | | |
|---|---|---|---|---|---|
| | *0* | *1* | *2* | *3* | *4* |
| **Stage 1** | x | | | | |
| **Stage 2** | | x | | | |
| **Stage 3** | | | x | | x |
| **Stage 4** | | | | x | |

**Load Units**

| | Time Steps | | | |
|---|---|---|---|---|
| | *0* | *1* | *2* | *3* |
| **Stage 1** | x | | | |
| **Stage 2** | | x | | |
| **Stage 3** | | | x | |
| **Stage 4** | | | | x |

**Store Units**

| | Time Steps | | | | |
|---|---|---|---|---|---|
| | *0* | *1* | *2* | *3* | *4* |
| **Stage 1** | x | | | | x |
| **Stage 2** | | x | | | |
| **Stage 3** | | | x | | |
| **Stage 4** | | x | | x | |

**FP Add Units**

| | Time Steps | | | | | | |
|---|---|---|---|---|---|---|---|
| | *0* | *1* | *2* | *3* | *4* | *5* | *6* |
| **Stage 1** | x | | | | x | | |
| **Stage 2** | | x | | | | x | |
| **Stage 3** | | x | x | x | | | x |

**FP Multiply Units**

| | Time Steps | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *0* | *1* | *2* | *3* | *4* | *5* | *6* | *7* | *8* | *9* | *10* | *11* | *12* | *13* | *14* | *15* | *16* | *17* | *18* | *19* | *20* | *21* |
| Stage 1 | x | | | | | | | | | | | | | | | | | | | | | |
| Stage 2 | | x | | | | | | | | | | | | | | | | | | | | |
| Stage 3 | | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x |

**Floating Point Div Units**