University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

An Enhanced Co-Scheduling Method Using Reduced MS-State Diagrams

R. Govindarajan†
N.S.S. Narasimha Rao‡
Erik R. Altman♯
Guang R. Gao

CAPSL Technical Memo 17
February 18, 1998

A shorter version of this memo appears in the Proceedings of the Merged 12th International Parallel Processing Symposium (IPPS-98) and 9th Symposium on Parallel and Distributed Processing, Orlando, Florida, March 1998

† Supercomputer Edn. & Res. Centre and Dept. of Computer Science & Automation
Indian Institute of Science, Bangalore, 560 012, India. `govind@{serc,csa}.iisc.ernet.in`
‡ Novell Software Development India Ltd., Garvephavipalya, Bangalore 560 068, India.
`nnarasimharao@novell.com`
♯ IBM T.J. Watson Research Centre,Yorktown Heights, NY 10598, U.S.A.
`erik@watson.ibm.com`

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Abstract

Instruction scheduling methods based on the construction of state diagrams (or automatons) have been used for architectures involving deeply pipelined function units. However, the size of the state diagram is prohibitively large, resulting in high execution time and space requirement, which in turn, restrict the use of these methods.

In this paper, we develop the underlying theory for reducing the size of state diagrams by identifying *primary* paths of a state diagram. We establish that a *reduced state diagram* consisting only primary paths is *complete*, *i.e.,* it retains all the useful information represented by the original state diagram as far as scheduling of operations in the software pipelining method 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.

Using the reduced MS-state diagrams, we develop an efficient software pipelining method. The proposed software pipelining algorithm produced efficient schedules when tested on a set of 1153 benchmark loops. Furthermore our software pipelining method performs significantly better than the original *Co-scheduling* method as well as Huff's *Slack Scheduling* method, in terms of both the initiation interval (II) and the time taken to construct a schedule.

Contents

List of Figures

List of Tables

1 Introduction

Software pipelining has been widely accepted as an efficient instruction scheduling method for loops [5, 14, 17, 18, 25, 24, 26, 29]. The constructed software-pipelined schedule, in addition to satisfying the loop-carried and loop-independent dependencies, must also satisfy the resource constraints imposed by the architecture. Function units in modern architectures involve increasingly more complex resource usage. Deeper pipelines and wide-word instructions are staging a resurgence. Processors capable of issuing 8 instructions per cycle are on the horizon, and structural hazard resolution 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 instruction scheduler must check and avoid any *structural hazard*, *e.g.,* contention for hardware resources by instructions. Conventional instruction scheduling methods [5, 7, 9, 10, 14, 17, 19, 25, 24, 26] accomplish this by maintaining a *global resource table* to model the resource usage. One drawback of this method 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 instruction scheduling technique — using ideas from the classical hardware pipeline theory [4, 16, 22] — has been proposed [2, 20, 23]. 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. The FSA method has been applied to the KSR (Kendall Square Research) compiler, and further extended to binary code translation for DEC ALPHA 21064 [2]. The *Co-scheduling* framework proposed by us in [12] is a software pipelining method that makes use of classical pipeline theory and state diagram construction. The constructed state diagram, called a Modulo-Scheduled (MS)-state diagram, represents valid latency sequences. 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.

Unfortunately, the practical use of state diagram based instruction scheduling methods [2, 12, 20, 23] is restricted due to the large size of the state diagram. In the context of Co-scheduling, the MS-state diagram may consist of a large number of paths especially for large values of the loop *initiation interval* **II**. As a consequence, the construction of the MS-state diagram can take a long time. For example, for a particular function unit FU-1 discussed in Section 5, there are 1.36 Million paths in the MS-state diagram for an **II** of 24.

One basic observation that we make in this paper is that a significant number of paths in a MS-state diagram are redundant. Thus, we can reduce the space requirement of the state diagram by identifying what are known as *primary paths.* A state diagram consisting only primary paths is termed as a *reduced state diagram.* We establish that the reduced state diagram is *complete*, *i.e.,* it retains all the useful information represented by the original state diagram as far as our scheduling goal is concerned. Our experiments reveal a significant reduction in the number of paths in the reduced MS-state diagram, The reduction is by a factor of several thousands or more in certain cases. In particular, in the above mentioned example, (for function unit FU-1) out of the 1.36 Million paths, only 60 are distinct — a reduction of more than 22,000 times!

In this paper, the original Co-scheduling method is also enhanced by considering multiple paths of the MS-state diagram and choosing the most appropriate one based on the data dependences in the loop. The use of multiple offset sets and the reduced state diagrams facilitate achieving better software pipelined schedules with smaller **II** and shortens the scheduling time. The enhanced Co-scheduling method based on reduced MS-state digram has been implemented and tested on a large number of loop kernels taken from several benchmark programs.

The major results observed in our experiments are:

1. The number of paths in the reduced state diagram has been greatly reduced; the reduction 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**, on the average. This, in turn, results in much lower storage requirement and time to construct the state diagram.

2. We have proposed an altnernative **direct** method to construct the scheduling information represented by the reduced state diagram. The direct method proposed in this paper is times faster, by a factor of 2 to 3, in terms of execution time compared to the construction of the reduced MS-state diagram.

3. Comparison with Huff's Slack Scheduling [14] highlights the improvement the enhanced Co-scheduling method can bring in terms of **II**, particularly, in loops where resource constraints dominate (inter-loop) dependency constraints.

4. Our enhanced Co-scheduling method requires lower execution time for scheduling loops. There is a 4-fold (or 400%) improvement in the scheduling time of our enhanced Co-Scheduling method compared to Huff's method.

5. Lastly, the enhanced Co-scheduling method also produces better schedules compared to the original Co-scheduling [12] in terms of both **II** and the time to construct the schedule.

In the following section we present the necessary background and motivate the need for reduced state diagram and enhanced Co-scheduling. In Section 3, we develop the underlying theory of reduced MS-state diagrams. Section 4 deals with the construction of reduced MS-state diagrams and the enhanced Co-scheduling algorithm. In Section 5, we report the results of our experiments. Section 6 deals with extension of our work to pipelines that share resources (stages). Related works are compared in Section 7 and concluding remarks are presented in Section 8.

2 Background and Motivation

In the first two subsections, we review our earlier work on Modulo-Scheduled (MS) state diagrams and Co-scheduling [12]. Co-scheduling is an efficient software pipelining method using the MS-state diagram. In subsection 2.3 we motivate the need for reduced state diagrams. Throughout this paper, we use the terms, "operations", "instructions", and "initiations" synonymously. We assume that our architecture consists of multiple function units (FUs), where each FU is capable of executing a specific instruction class. Further, each FU is a *static* pipeline whose resource usage pattern is described by a single reservation table [16]. It is possible to extend the ideas relating to reduced state diagram and the enhanced Co-scheduling method to architectures in which the pipelines share certain resources, such as the result write bus. A brief discussion on this extension is presented in Section 6.

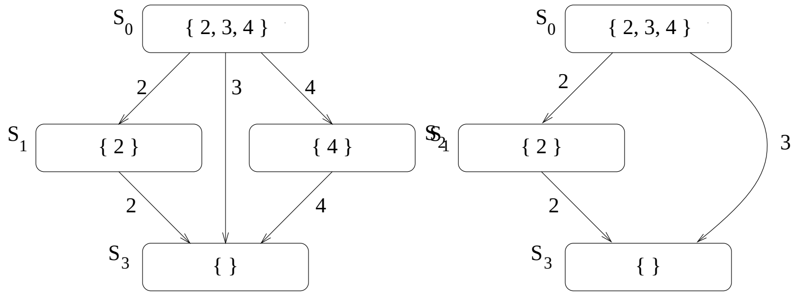
An important difference between the (cyclic) scheduling of operations in a hardware pipeline as discussed in [4, 3] and our work is that, the periodicity of the former is determined only by the resource usage; whereas, in software pipelining, the periodicity (or initiation interval (II)) depends not only on the resource usage (of this pipeline) but also on other resources, their usage, as well as the data dependences. To emphasize this difference, we refer to our pipelines operating under modulo scheduling (or software pipelining) as *Modulo-Scheduled (MS-) Pipelines.*

2.1 Modulo-Scheduled State Diagram

In order to analyze hardware pipelines operating under modulo scheduling and derive latency sequence that do no cause structural hazards, we construct the MS-state diagram [13]. A quick review of related concepts and the construction of MS-state diagram is given in Appendix A. In the following discussion, we present an example state diagram and its usage in the Co-Scheduling method.

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

(a) Cyclic Reservation Table



(b) (c)

Figure 1: Full and Reduced MS-State Diagrams for Example Reservation Table

For the reservation table of Figure 1(a), the MS-state diagram for an **II** $= 6$ is shown in Figure 1(b). 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 latency sequence** $\{p_1, p_2, \cdots, p_k\}$. The latency sequence represents $k+1$ initiations that are made at time steps 0, p_1, $(p_1 + p_2)$, $\cdots$, $(p_1 + p_2 + \cdots + p_k)$. In modulo scheduling, these time steps correspond to the **offset values**

$$0, p_1, (p_1 + p_2) \texttt{ mod } \mathbf{II}, \ \cdots \ , (p_1 + p_2 + \cdots + p_k) \texttt{ mod } \mathbf{II}$$

in a repetitive kernel.

As an example, the latency sequence $\{3\}$ corresponding to the path $S_0 \xrightarrow{3} S_3$ represents only 2 initiations made at time steps 0 and 3. However, the latency sequence {4,4}, corresponding to the path $S_0 \xrightarrow{4} S_2 \xrightarrow{4} S_3$, represents 3 initiations made at offsets 0, 4, and 2. Thus analyzing the MS-state diagram reveals valid latency sequences and the corresponding offset sets that do not cause collision. Further, in this state diagram paths $S_0 \xrightarrow{2} S_1 \xrightarrow{2} S_3$ and $S_0 \xrightarrow{4} S_2 \xrightarrow{4} S_3$ result in maximum number of initiations in the pipeline. We refer to $\{2, 2\}$ and $\{4, 4\}$ as maximum initiation latency sequences.

A software pipelining algorithm, called Co-scheduling, that uses (one of the) maximum initiation latency sequences from the MS-state diagram was reported in [12]. The details of this algorithm are presented in the following subsection.

2.2 Software Pipelining using MS-State Diagram

Co-scheduling was based on Huff's Slack Scheduling algorithm [14]. Co-scheduling starts with a Minimum Initiation Interval (**MII**) and attempts to schedule the loop for values of **II** $\geq$ **MII** until a schedule is found. For a given **II** it computes the MS-state diagram for all FUs that have structural hazards. To make the discussion concise and address only the relevant points, we will assume that there is only one type of pipeline[1]. From the MS-state diagram for the pipeline, a maximum initiation latency sequence is derived.

The basic notion of Huff's original Slack Scheduling [14] 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. These points remain in 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. In the Co-scheduling method, however, an instruction is scheduled only at pre-determined offset values given by the maximum initiation latency sequence.

To clarify matters, consider the motivating example (Figure 1). Suppose we have chosen the maximum initiation latency sequence $\{2, 2\}$ and the corresponding offset values are 0, 2, and 4. Further let us assume an instruction i_1 is scheduled at time step, say, 1 and two more instructions i_2, and i_3 are to be scheduled in the same pipeline. If the slack of i_2 is $(4, 6)$, *i.e.,* instruction i_2 can be placed at any time step from 4 to 6, the Co-scheduling method will choose time step 5, so that the offset between i_1 and i_2 is 4, as governed by the chosen latency sequence.

The Co-scheduling method goes by the chosen maximum initiation latency sequence, even if this means avoiding some other latency values that would yield a legal partial schedule. Note that, in the considered example, even though i_2 can be scheduled at time step 4, with a (permissible) latency 3 with i_1, the Co-scheduling method does not attempt this. This is because if i_1 and i_2 are scheduled at times 1 and 4 (offsets 0 and 3) respectively, with a latency 3 between them, then, as can been seen from the MS-state diagram, no further initiations are possible in the pipeline. That is, instruction i_3 cannot be scheduled in the same pipe. Thus the Co-scheduling method takes a global view in scheduling instructions and deals with structural hazards much more efficiently, by using a permissible latency sequence.

2.3 Motivation for Reduced State Diagram

The use of a single offset set, as done in Co-scheduling, may not be sufficient to obtain a schedule with the lowest value of **II**. We illustrate this with our example discussed in Section 2.2. Suppose there are only two instructions i_1 and i_2. Further assume that instructions i_1 and i_2 have a tight slack, requiring them to be scheduled at time steps 1 and 4 respectively. If we use a single set of *offset* values, *e.g.,* 0, 2, and 4, as suggested by the original Co-scheduling, then one of the two instructions cannot be scheduled in the pipeline, even though the chosen latency sequence can

[1]This assumption (only one type of pipeline) is made only to simplify the discussion. Neither the original Co-scheduling method nor the enhanced Co-scheduling method presented in this paper is restricted by this.

accommodate 3 operations in the pipeline. In this case, eventually the Co-scheduling algorithm will result in an **II** of 7. However, if we had used the latency sequence $\{3\}$, or the corresponding offset values 0 and 3, then we could have scheduled the loop with **II** $=$ 6. Thus, it is beneficial, if Co-scheduling can consider all latency sequences that allow more than n initiations, where n is the number of operations in the given loop that are to be executed in this pipeline. However, as discussed in the earlier paragraph, the number of offset sets for an MS-state diagram, can be quite large (greater than several 100,000s).

It should be noted that the MS-state diagram changes for different values of **II**. The number of paths in an MS-state diagram increases drastically for large values of **II**. For example, the number of distinct paths for the reservation table shown in Figure 1(a) increases from 12 to 200,496 when **II** value changes from 8 to 18 (Refer to Figure 2 in Section 5.1 to see the rate of increase of number of paths with respect to **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 1(b). Clearly, the states S_0, S_1, S_2, and S_3 are all distinct. However, the information represented by the paths $S_0 \xrightarrow{2} S_1 \xrightarrow{2} S_3$ and $S_0 \xrightarrow{4} S_2 \xrightarrow{4} S_3$ 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 $0, 2$, and 4. 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. The reduced state diagram is shown in Figure 1(c).

In this paper, we develop a method to construct the reduced MS-state diagram and show that the constructed state diagram is *complete*, *i.e.,* contains the set of all offset values at which initiations can be made in a pipeline. In Section 4, an enhanced Co-scheduling algorithm that uses many latency sequences, not just a maximum initiation latency sequence, to construct an efficient software pipelined schedule is presented.

3 Reduced MS-State Diagram

In this section we develop the necessary theory behind the construction of Reduced MS-state diagrams. Section 4 deals with the construction of Reduced MS-state diagrams.

3.1 Definitions

We begin with a number of definitions. Throughout this paper, we use the notation S_0 and S_f to represent respectively the *start* and the *final* states of the MS-state diagram[2]. Further, the term "path" always refers to a path from the start state to the final state, unless specified otherwise.

Definition 3.1 *Given 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, the* **offset set** *for the path are*

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

where $\oplus$ *stands for addition modulo* **II**.

In the above definition, and throughout the paper, without loss of generality, we assume that the first initiation is made at an offset 0. Thus all offset values are relative to the first initiation. Further, each of the offset values, except 0, must equal one of the initial permissible latencies. Otherwise, we have two initiations, one at offset 0, and another at, say f, with a forbidden latency between the two.

Definition 3.2 *The set of* **permissible offsets** $\mathcal{O}$ *is defined as*

$$\mathcal{O} = \{S_0 \cup \{0\}\ \}. \tag{1}$$

where S_0 *represents the initial permissible latency set.*

That is, the set of permissible offsets includes all initial permissible latencies and 0 (zero). For our motivating example (CRT shown in Figure 4(c), the permissible offsets are $\{0, 2, 3, 4\}$.

Definition 3.3 *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,*

$$p_1 + p_2 + \cdots + p_k < \mathbf{II}$$

A path is called **secondary** *if*

$$p_1 + p_2 + \cdots + p_k > \mathbf{II}$$

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 [16, 22].

[2]The final state is one in which no further initiations are possible. For example, the final state for the MS-state diagram shown in Figure 1(a) is state S_3.

Definition 3.4 *Two offsets o_1 and o_2 belonging to $\mathcal{O}$ are* **compatible** *if $(o_1 - o_2)$ $\texttt{mod}$ $\mathbf{II}$ is in $\mathcal{O}$.*

Definition 3.5 *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 3.6 *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.

3.2 Properties of 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 3.1 *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

$$o_0 = 0; \quad o_1 = p_1; \quad o_2 = p_1 \oplus p_2; \quad \cdots \quad o_k = p_1 \oplus p_2 \oplus \cdots \oplus p_k; \tag{2}$$

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 $\mathbf{II}$, 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[3]. 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

[3]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 $\mathbf{II}$ cycles, the difference between the offsets suffices.

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,

$$\{p_1, (p_1 \oplus p_2), \cdots, (p_1 \oplus p_2 \oplus \cdots \oplus p_k), c\} \subseteq S_0 \tag{3}$$

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

$$\{(p_1 \oplus p_2) \ominus p_1, \cdots, (p_1 \oplus p_2 \oplus \cdots \oplus p_k) \ominus p_1, c \ominus p_1\} \subseteq S_1 \tag{4}$$

This can be rewritten, using Equation 2 as.

$$\{o_2 \ominus o_1, o_3 \ominus o_1, \cdots, o_k \ominus o_1, c \ominus o_1\} \subseteq S_1 \tag{5}$$

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

$$\{(p_1 \oplus p_2 \oplus p_3) \ominus p_1 \ominus p_2, \cdots, (p_1 \oplus p_2 \oplus \cdots \oplus p_k) \ominus p_1 \ominus p_2, c \ominus p_1 \ominus p_2\} \subseteq S_2 \tag{6}$$

To see how each element on the L.H.S. of Equation 6 belong to S_2, rearrange Equation 6 as follows, and apply the arguments that each offset o_i (as well as c) is compatible with $o_2 (= p_1 \oplus p_2)$.

$$\{(p_1 \oplus p_2 \oplus p_3) \ominus (p_1 \oplus p_2), \cdots, (p_1 \oplus p_2 \oplus \cdots \oplus p_k) \ominus (p_1 \oplus p_2), c \ominus (p_1 \oplus p_2)\} \subseteq S_2 \tag{7}$$

$$i.e., \quad \{(o_3 \ominus o_2), \cdots, (o_k \ominus o_2), c \ominus o_2\} \subseteq S_2 \tag{8}$$

Proceeding this way, we can show that

$$\{(p_1 \oplus p_2 \oplus \cdots \oplus p_k) \ominus (p_1 \oplus p_2 \oplus \cdots \oplus p_{k-1}), c \ominus (p_1 \oplus p_2 \oplus \cdots \oplus p_{k-1})\} \subseteq S_{k-1} \tag{9}$$

and

$$\{c \ominus (p_1 \oplus p_2 \oplus \cdots \oplus p_k)\} \subseteq S_f \tag{10}$$

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 3.2 *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,

$$S_0 \xrightarrow{p_1} S_1 \xrightarrow{p_2} S_2 \cdots \xrightarrow{p_k} S_f. \tag{11}$$

where

$$p_i = c_i - c_{i-1} \tag{12}$$

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 3.3 and 3.4 establish this. Thus the path in Equation 11 is a valid path representing a legal latency sequence. By Theorem A.1, every legal path must be in the MS-state diagram which completes the proof.

Lemma 3.3 *The latencies p_i of path shown in Equation 11 are permissible.*

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 3.4 *For the path shown in Equation 11, the latency p_i is a permissible latency (an element) in S_{i-1}.*

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 12, and $c_0 = 0$, we get

$$p_1 = (c_1 - c_0) = c_1. \tag{13}$$

Next, using Equation 12 in the second offset value.

$$p_1 \oplus p_2 = (c_1 - c_0) + (c_2 - c_1) = c_2 \tag{14}$$

Proceeding this way, we get

$$(p_1 \oplus p_2 \oplus \cdots \oplus p_k) = c_k \tag{15}$$

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 3.1. 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 (by Equation 1). 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 13 to 15, we get

$$\{p_1, (p_1 \oplus p_2), \cdots, (p_1 \oplus p_2 \oplus \cdots \oplus p_k)\} \subseteq S_0 \tag{16}$$

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 16, 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,

$$\{(p_2), (p_2 \oplus p_3) \cdots, (p_2 \oplus p_3 \oplus \cdots \oplus p_k)\} \subseteq S_1 \tag{17}$$

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

$$\{(p_3), (p_3 \oplus p_4), \cdots, (p_3 \oplus p_4 \oplus \cdots \oplus p_k)\} \subseteq S_2 \tag{18}$$

Proceeding further, we get

$$\{(p_k)\} \subseteq S_{k-1} \tag{19}$$

Hence the Lemma. □

Next we will show that the path shown in Equation 11 is primary.

Lemma 3.5 *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 3.2 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 12, 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 3.1 *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 3.1, the secondary path under consideration, results in an offset set that equals a maximal compatibility class of $\mathcal{O}$. But by Lemma 3.5, for this maximal compatibility class there exists a **primary** path that supports the same offset set. Hence the theorem. □

Theorem 3.2 *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 3.1. □

In the following section we present a method to construct the MS-state diagram with only primary paths. The following section also introduces the improved Co-scheduling method that uses the reduced MS-state diagram.

4 Enhanced Co-Scheduling using Reduced MS-State Diagram

In this section, we present two methods to obtain the offset sets corresponding to all primary paths in the MS-state diagram. In section 4.3, we present our enhanced Co-scheduling algorithm which uses the reduced MS-state diagram, and is based on Huff's bidirectional Slack Scheduling algorithm [14] and the original Co-scheduling algorithm [12].

4.1 Generation of Offset Sets using Reduced MS-State diagram

As demonstrated in Section 3 (Theorem 3.2), 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. Identifying secondary paths can be done during the time of creation of a new state in the MS-state diagram (Step 2 in the construction procedure, Procedure 2.1). But the elimination of secondary paths requires a subsequent traversal on the state diagram.

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.

4.2 Generation of Offset Sets using Maximal Compatible Classes

Lemmas 3.1 and 3.2 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 [16] (page 99). This approach is a direct way of obtaining the set of all offset sets of the reduced MS-state diagram. We compare the performance of the offset set generation using the two methods, *viz.,* reduced MS-state diagrams and maximal compatible classes, in Section 5.2. 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.

4.3 Enhanced Co-Scheduling Algorithm

As mentioned earlier, our Co-scheduling algorithm is similar to Huff's bidirectional Slack Scheduling algorithm. In both approaches, instructions are picked from the set of unscheduled instructions based on a priority function. An instruction with the smallest slack, gets highest priority in the scheduling order. Further, like in Huff's scheduling, priority is a dynamic measure and the slacks of unscheduled instructions are updated after the placement of every instruction in the partial schedule. 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 [14]. 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. In these aspects the two scheduling methods are alike.

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[4]. Sets with smaller cardinalities, need not be considered as they do not support the required number of instructions. 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 B.

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[5] *(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

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

[5] 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.

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[6], 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.

4.4 Remarks

Our method differs from Huff's approach in three aspects:

Resource Usage Representation: While the Slack Scheduling method uses a Modulo Reservation Table (MRT) [14, 24] to represent both the schedule of operations and their resource usage, our enhanced Co-scheduling algorithm uses a Modulo Initiation Table (MIT) [12] to represent the schedule of operations. Resource usage need not be explicitly represented in the MIT as we attempt to schedule operations only in time steps (offset values) that do not cause a collision. Hence, each FU can be represented in the MIT by a single row. In contrast, each stage of an FU corresponds to a row in the MRT.

Cycles Considered for Scheduling: 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.

Decisions on Ejecting Operations: 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

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

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. It has been observed that these differences, affect the schedules of loops in which the resource constraints dominate over the recurrence constraints. A detailed discussion on the comparison between the two approaches is presented in Section 5.

Our method differs from the original Co-scheduling approach [12] mainly in terms of the number of offset sets considered for scheduling. The Co-scheduling method considered only a *single* latency sequence, and the corresponding offset set. Our approach extends the original Co-scheduling method to consider multiple offset sets, and thus providing more opportunities for scheduling the loop.

5 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.

5.1 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 1 in Section A) and the reduced MS-state diagram (described in Section 4.1). 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[7] of **II** from 8 to 24. Table 1 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 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.

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

[7] 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 3(b).

Avg. Reduction in No. of Paths	$8 \leq$ **II** ≤ 15						$16 \leq$ **II** ≤ 24					
	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 1: Average Reduction in the Number of **Paths**.

of the number of paths in Figure 2. 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.

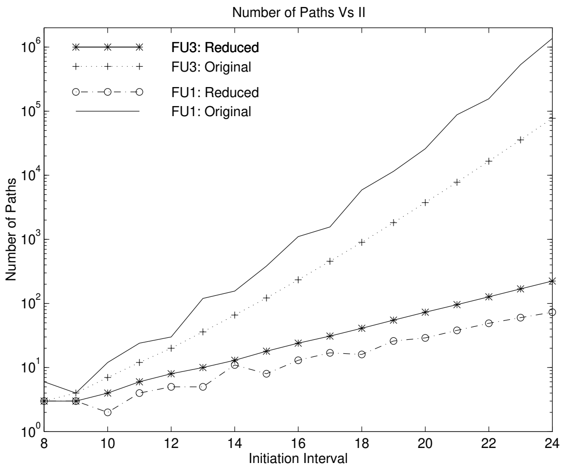


Figure 2: Increase in Number of Paths for Different **II**s

5.2 Maximal Compatible Set Generation

As discussed in Section 4.2, 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 [16], 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 2. 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 2: Execution Time of Two Methods to Generate Offset Sets

5.3 Performance of Enhanced Co-Scheduling Method

We have implemented the enhanced Co-scheduling method presented in Section 4.3, 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 C. 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 5.1 and 5.2. The results of our experiments are shown in Table 3.

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
≥ 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 3: Performance of Enhanced Co-Scheduling

Table 3(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 3(b)).

The (arithmetic) mean time to compute a schedule is 2.9 milliseconds, while the median for this is 1.1 milliseconds. Table 3(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 3(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. Lastly, even though enhanced Co-scheduling uses the set of all offset sets, typically several hundreds in number, it still was successful in finding a schedule within a few milliseconds.

5.4 Comparison with Huff's Scheduling Method

The enhanced Co-scheduling method is compared with our implementation of Huff's Slack Scheduling method[8]. Among the several heuristic methods for software pipelining, 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 4. As seen from Table 4 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 minor (1.6%). This could be due to the order in which we eject the instructions in the enhanced Co-scheduling method.

The results shown in Table 4 exhibit a significant reduction in execution time (time to construct schedules). In 988 of the benchmarks programs, the execution time of our Co-scheduling

[8]To the best of our knowledge, our implementation of Huff's Slack Scheduling method faithfully follows the implementation details presented in [14].

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 4: Comparison of Enhanced Co-Scheduling with Slack Scheduling

method is lower than that of Slack Scheduling. The average improvement in execution time is roughly by a factor of 4.5 on the average. As mentioned in the earlier subsection, the execution time of our enhanced Co-scheduling method did not include the time to construct the MS-state diagram.

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 4.4, 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 4, 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 metthod 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.

5.5 Comparison with Original Co-Scheduling Method

Lastly, we compare the performance of the enhanced Co-scheduling method with the original method [12]. Since the original Co-scheduling method considered only a single offset set, in more than 23% of benchmarks, the enhanced method obtained lower **II**. The average improvement in **II** is 16.2%. Once again, the improvement in **II** is observed mainly in resource-critical loops. Lastly, both methods show improvement, in terms of, the average number of trials, average number of ejected operations, and execution time, in more or less equal number of benchmarks.

Measure	Enhanced Co-Scheduling Better			Original Co-Scheduling Better			Both Same	
	No. of Benchmarks	% Cases	% Improvement	No. of Benchmarks	% Cases	% Improment	No. of Benchmarks	% Cases
II	255	23	16.2	4	0.03	1.4	871	77
Avg. Ejections per Instrn.	88	8	526.5	149	13	525.2	893	79
Avg. Trials per Instrn.	144	13	24.6	97	9	24.7	889	79
Exec. Time	651	58	50.3	476	42	50.3	5	0

Table 5: Comparison of Enhanced Co-scheduling with Original Co-scheduling

5.6 Summary of Results

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

- 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 1).

- 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 2).

- Comparison with Huff's Slack Scheduling [14] 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%).

- The enhanced Co-scheduling method performs better than the original Co-scheduling [12] in terms of **II**, in 255 loops. However, its scheduling time is comparable to that of original Co-scheduling.

6 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 3: 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 3. 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 [16, 23, 2]. We extend the initial permissible latency set to permissible latency matrices[9] 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 similarlity between these definitions and the definition of *collision matrices* in [16, 23, 2].

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

[9] 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 [16] will have exactly **II** bits in each row. Hence we continue to call this a permissible matrix.

the ith row of the current state (represented by permissible matrix)), then the new state S' is obtained by subtracting p (subtraction modulo **II**) from each element of the permissible matrix S and taking the intersection, on a row-by-row basis[10], with the initial permissible matrix for instruction class 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.

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

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

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

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–FP-Mult}. 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? We leave these questions for future work.

7 Related Work

Several software pipelining methods have been proposed in the literature [5, 8, 11, 1, 14, 17, 25, 26, 28]. These methods are based on a global modulo reservation table. A comprehensive survey of these works is provided by Rau and Fisher in [24]. 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 [4, 16, 22].

[10] 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.

The approach proposed in [2, 20, 23] 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 [12] 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 [22, 16, 3]. 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.

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 [6]. 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.

8 Conclusions

In this paper we present an enhanced Co-scheduling method that makes use of the state diagram based approach for software pipelining. This work identifies and eliminates redundant information in the original MS-state diagram. We demonstrate 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**, on the average. The reduction in the number of paths results in space efficiency. We establish the necessary theory behind the reduced MS-state diagram.

We extend the original Co-scheduling method by considering multiple latency sequences from the MS-state diagram. The enhanced Co-scheduling method addresses a major problem of the original Co-scheduling method, *viz.,* the original Co-scheduling method can do worse than Huff's slack scheduling, if the chosen latency sequence does not "fit" well with the slacks of the instructions. 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. Our experiments demonstrate that the performance of the enhanced Co-scheduling is better than Huff's scheduling method as well as the original Co-scheduling in terms of both **II** and the time to construct the schedule. Lastly, we show how to extend our enhanced Co-scheduling method for pipelines that share resources.

Acknowledgments

We wish to thank Sean Ryan, Chihong Zhang, members of CAPSL, University of Delaware, and the anonymous referees for their helpful suggestions. The last author acknowledges the support of National Science Foundation, U.S.A.

References

[1] 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, California, June 18–21, 1995.

[2] 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, Michigan, November 29–December1, 1995.

[3] 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.

[4] 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 Intl. Conf., COMPCON Spring '75*. February 1975.

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

[6] 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*, Philadelphia, Pennsylvania, May 21–24, 1996.

[7] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 7(30):478–490, July 1981.

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

[9] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proc. of the SIGPLAN '86 Symp. on Compiler Construction*, pages 11–16, Palo Alto, California, June 25–27, 1986.

[10] J. R. Goodman and W-C. Hsu. Code scheduling and register allocation in large basic blocks. In *Conf. Proc., 1988 Intl. Conf. on Supercomputing*, pages 442–452, St. Malo, France, July 4–8, 1988.

[11] 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, Calif., Nov. 30–Dec.2, 1994.

[12] 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, California, February 3–7, 1996.

[13] R. Govindarajan, E. R. Altman, and G. R. Gao. Modulo-Scheduled Pipelines. ACAPS technical memo, School of Computer Science, McGill University, Montréal, Québec, June 1996.

[14] 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, New Mexico, June 23–25, 1993.

[15] W. M. Hwu, et. al. The superblock: An effective technique for VLIW and superblock compilation. *Journal of Supercomputing*, 7:229–248, January 1993.

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

[17] 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.

[18] 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, Michigan, November 29–December1, 1995.

[19] S-M. Moon and K. Ebcioğlu. An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. In *Proc. of the 25th Ann. Intl. Symp. on Microarchitecture*, pages 55–71, Portland, Oregon, December 1–4, 1992.

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

[21] B. Natarajan and M. Schlansker. Spill-free parallel scheduling of basic blocks. In *Proc. of the 28th Ann. Intl. Symp. on Microarchitecture*, pages 119–124, Ann Arbor, Michigan, November 29–December1, 1995.

[22] 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, Florida, January 19–21, 1976.

[23] 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, Oregon, January 17–21, 1994.

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

[25] 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, Massachusetts, October 12–15, 1981.

[26] 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, California, November 30–December2, 1994.

[27] B. R. Rau, M. S. Schlansker, and P. P. Tirumalai. Code generation schema for modulo scheduled loops. In *Proc. of the 25th Ann. Intl. Symp. on Microarchitecture*, pages 158–169, Portland, Oregon, December 1–4, 1992.

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

[29] N. J. Warter-Perez and N. Partamian. Modulo scheduling with multiple initiation intervals. In *Proc. of the 28th Ann. Intl. Symp. on Microarchitecture*, pages 111–118, Ann Arbor, Michigan, November 29–December1, 1995.

A Review of MS-Pipeline Theory

In classical pipeline theory, the resource usage of a hardware pipeline is represented by an $m \times l$ matrix where m is the number of stages in the pipeline and the l is the execution latency of the pipeline. However, the resource usage of this pipeline under modulo scheduling is represented by a *Cyclic Reservation Table (CRT)* of m rows and **II** columns. If the execution latency l of the hardware pipeline is less than **II**, then blank columns are appended at the end to get **II** columns. Otherwise, the reservation table is *folded* to **II** columns; a resource usage at (i, j) (ith row and jth column) is represented at $(i, (j \texttt{ mod } \mathbf{II}))$ in the CRT [11]. As an example, the reservation table in Figure 4(a) yields the CRT shown in Figure 4(b) and Figure 4(c) for $\mathbf{II} = 3$ and $\mathbf{II} = 6$ respectively.

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

(a) Reservation Table

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

(b) CRT for $\mathbf{II} = 3$.

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

(c) CRT for $\mathbf{II} = 6$.

Figure 4: Cyclic Reservation Tables for Different **II**s

It should be noted that the CRT is not same as a *Modulo Reservation Table (MRT)* used in existing software pipelining methods [5, 14, 17, 25, 24, 26]. The former represents the resource usage of a single initiation whereas the latter represents the schedule of all operations in the loop and their resource usage. Besides, as will be seen in Section 4.4, the Co-scheduling method

[11] With this folding, `X` marks separated by **II** columns may be placed on the same column in the CRT. However, under modulo scheduling it is not possible to have the same stage of a pipeline used (by one or more operations from the same iterations) at time steps separated by **II** cycles. For simplicity, we assume that the folding of the reservation does not violate the above constraint, known as the *modulo scheduling constraint.* It can be noted that modulo scheduling constraint can be satisfied by using the delay insertion method proposed in [22].

will use the *Modulo Initiation Table (MIT)*, in which the different stages of an FU correspond to a single row [12].

The time elapsed between two initiations in a pipeline is termed *latency*. A latency is said to cause a *collision* if the two instructions require the same stage of the pipeline at the same time. Multiple operations can simultaneously be processed in the pipeline as long as there is no collision. A latency that results in a collision is called *forbidden latency*. The distance between pairs of `X` marks in the rows of the CRT determine the *forbidden latencies* in an MS-pipeline. If there exists a row s in the CRT such that both (s, t) and $(s, (t + f) \texttt{ mod } \mathbf{II})$ contain an `X` mark, then f is a forbidden latency. In an MS-pipeline, if f is forbidden, then $\mathbf{II} - f$ is also forbidden. The latencies that are not forbidden are termed as *permissible*. The forbidden and permissible latency sets for the reservation table shown in Figure 4(c) are $\{0, 1, 5\}$ and $\{2, 3, 4\}$ respectively.

To analyze the MS-pipeline and to identify valid latency sequences that do not cause a collision, we construct the MS-state diagram [13]. The initial state in the MS-state diagram represents an initiation at time 0. Each state in the state diagram is represented by the set of permissible latencies in that state. Thus, the initial state consists of the initial permissible latencies. The construction of the MS-state diagram proceeds as follows.

Procedure 1: Construction of MS-State Diagram:

Step 1 Start with the initial state having the initial permissible latency set S_0.

Step 2 For each permissible latency p_i in the current state S, derive a subsequent state S' with the following permissible latency set.

$$S' = S_{-p_i} \cap S_0 \qquad \text{where} \qquad S_{-p_i} = \{(p_j - p_i)\texttt{mod } \mathbf{II} | \forall p_j \in S.\}$$

(See explanation below.)

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 $\mathbf{II}$. The resulting set is the set of *possible* permissible latencies at the current state. Of these latencies, those that are members of the initial forbidden latency set will be forbidden; other will be permissible. Thus the intersection with the initial permissible latency yields the set of permissible latencies in the new state S'.

In constructing the MS-state diagram, duplicate states are eliminated. For example, if the permissible latency set for a new state S' computed by Step 2 is such that there exists another state S'' with same permissible latencies, then states S' and S'' are represented by a single state S'' , and an arc $S \xrightarrow{p_i} S''$ is created instead. Thus each state in the MS-state diagram is distinct. Because of this, it may happen that there could be multiple arcs between a pair of states with different latencies associated with them. Diagrammatically, these multiple arcs are represented by a single arc and multiple latencies associated with it. It was shown in [13] that the state S' produced by Step 2 in the construction procedure consists of all, and only, permissible latencies in the new state. Formally,

Theorem A.1 *Every state S in the MS-state diagram, derived according to* **Procedure 1**, *represents all permissible latencies in that state, taking into account all initiations made so far to reach the state S.*

The above MS-state diagram representation and the construction procedure were also shown to be equivalent to their counterparts discussed in [12].

B 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** Construct the Reduced MS-state diagram;
  - **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 paths 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_OPS`), 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 */

C 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