Design Automation for Embedded Systems, 6, 243–275, 2002.

A Theory for Co-Scheduling Hardware and Software Pipelines in ASIPs and Embedded Processors*

R. GOVINDARAJAN govind@serc.iisc.ernet.in
Supercomputer Edn. and Res. Centre, Indian Institute of Science, Bangalore, 560 012, India

ERIK R. ALTMAN erik@watson.ibm.com
IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, U.S.A.

GUANG R. GAO ggao@capsl.udel.edu
Electrical & Computer Engineering, University of Delaware, Newark, DE 19716, U.S.A.

Abstract. Exploiting instruction-level parallelism (ILP) is extremely important for achieving high performance in application specific instruction set processors (ASIPs) and embedded processors. Unlike conventional general purpose processors, ASIPs and embedded processors typically run a single application and hence must be optimized extensively for this in order to extract maximum performance. Further, low power and low cost requirements of ASIPs may demand reuse of pipeline stages causing pipelines with complex structural hazards. In such architectures, exploiting higher ILP is a major challenge to the designer.

Existing techniques deal with either scheduling hardware pipelines to obtain higher throughput or software pipelining—an instruction scheduling technique for iterative computation—for exploiting greater ILP. We integrate these techniques to co-schedule hardware and software pipelines to achieve greater instruction throughput. In this paper, we develop the underlying theory of Co-Scheduling, called the *Modulo-Scheduled Pipeline* (or MS-Pipeline) theory. More specifically, we establish the necessary and sufficient condition for achieving the maximum throughput in a given pipeline operating under modulo scheduling. Further, we establish a sufficient condition to achieve a specified throughput, based on which we also develop a methodology for designing the hardware pipelines that achieve such a throughput. Further, we present initial experimental results which help to establish the usefulness of MS-pipeline theory in software pipelining. As the proposed theory helps to analyze and improve the throughput of *Modulo-Scheduled Pipelines (MS-pipelines)*, it is especially useful in designing ASIPs and embedded processors.

Keywords: Classical pipeline theory, co-scheduling, modulo-scheduled pipelines, software pipelining, synthesis of ASIPs and embedded processors.

1. Introduction

In order to achieve high performance, application specific instruction set processors (ASIPs) in general, and embedded processors in particular, must exploit instruction-level parallelism (ILP). Modern DSP processors such as TI's C6x [25], or Philip's TriMedia [20], and their embedded cores are very long instruction word (VLIW) architectures to exploit instruction-level parallelism. Unlike in conventional general purpose processors, ASIPs

* A shorter version of this work has appeared in the *Proceedings of the International Conference on Application Specific Array Processors*, Boston, MA, July 2000.

and embedded processors are customized for a specific class of applications to extract greater performance. Further, as ASIPs and embedded processors typically run a single application or single class of applications, both hardware and software are optimized extensively. Hence it is not uncommon in the embedded system domain that some of these synthesis/compiler optimizations are run for several hours to obtain highly efficient design. This is in sharp contrast with traditional compiler optimizations where the optimizations need to be run quickly and often compromise optimality and obtain efficient heuristic near-optimal solutions.

In addition, ASIPs and embedded processors are used in hand-held devices and hence generally have low-power requirements. Further, because of the wide spread deployment in video games and other devices, embedded processors have low-cost requirement too. As a result, ASIPs may result in pipelines with complex structural hazards. These further complicate the compiler/synthesis optimizations of ASIPs. In such architectures, exploiting higher ILP is a major challenge to the designer. Existing techniques either deal with scheduling hardware pipelines to obtain higher throughput [19], [14] or perform aggressive instruction scheduling and software pipelining [13], [15], [22]–[24] to exploit the ILP.

The theory of hardware pipelines, developed in [19], [14] discusses a method to analyze and improve the throughput and the utilization of pipelined and vector architectures [14], [19]. Their approach selects a latency cycle having the minimum average latency, and schedule operation in the hardware pipeline based on the period of the latency cycle. An important point to observe here is that the period of the latency cycle is solely determined by the hardware pipeline structure. We refer to the above scheduling as **cyclic scheduling of hardware pipelines**. They have also proposed an approach to introduce delays in the pipeline structure to improve its throughput.

Modulo scheduling or *software pipelining* [22], [15], [13], [5], [23], [24] is a compile-time instruction scheduling technique for iterative computation where instructions from successive iterations are overlapped to exploit higher ILP. In a modulo schedule, different instances of an instruction corresponding to different iterations are scheduled **II** (Initiation Interval) cycles apart. Thus the initiation rate of the loop is 1/**II**. To maximize the performance, i.e., to improve initiation rate, the objective of a modulo scheduler is to minimize **II**.

Existing techniques for exploiting ILP have dealt with scheduling of hardware and software pipelines independently. In order to achieve higher throughput and better utilization of the hardware resources, it is important to integrate the scheduling of hardware and software pipelines. Our earlier work [10], [11] adapts classical pipeline theory [19], [14], to co-schedule the hardware and software pipelines to obtain efficient modulo schedules. Similar efforts employing the use of classical pipeline theory in instruction scheduling have been studied in [18], [21], [2]. These work propose an FSA (finite state automata) based approach for instruction scheduling. However, as discussed in [10], [11] the FSA-based approach for instruction scheduling cannot be directly applied for software pipelining.

In this paper we develop the underlying theory called the *Modulo-Scheduled Pipeline theory* (or *MS-pipeline theory*), to co-schedule the hardware and software pipelines. The

MS-pipeline theory provides a mechanism for analyzing the throughput of MS-pipelines, and is also useful in designing pipelines to achieve a specified throughput. Thus, the MS-pipeline theory is in general useful in hardware-software co-design [12], and in particular, suitable for ASIP or embedded processor design. The major contributions of this paper are:

1. We establish the necessary and sufficient condition for achieving the maximum throughput in a given pipeline operating under modulo scheduling. The condition is powerful, yet surprisingly simple to check.

2. We establish a sufficient condition to achieve a specified number of instruction initiations in a hardware pipeline operating under modulo scheduling. Unlike the necessary and sufficient condition which is too restrictive, the sufficient condition established is more useful in establishing a method to obtain a given number of initiations in an MS-pipeline.

3. We demonstrate that the delay insertion method of classical pipeline theory can be adapted to MS-pipeline. The delay insertion method facilitates designing hardware pipelines that achieve the maximum throughput. Thus it is especially useful in the co-design of hardware and software pipelines in ASIPs and embedded processors.

The rest of the paper is organized as follows. In the following section we motivate the need for the theory of MS-pipelines through a number of examples. Related work are compared in Section 3. Section 4 develops the basic framework of MS-pipelines. In Section 5, we establish the necessary and sufficient condition for achieving the maximum throughput in a hardware pipeline operating under modulo scheduling. Section 6 describes a method to introduce delays in the pipeline to achieve a specific initialization sequence. Experimental results on the usefulness of the MS-pipeline theory are reported in Section 7. Concluding remarks are presented in Section 8.

2. Background and Motivation

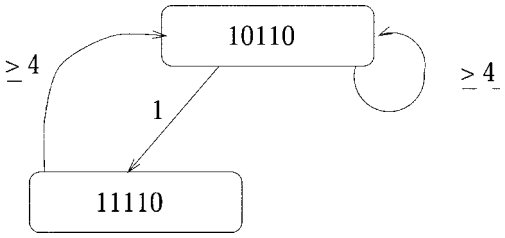
In this section, we motivate the need for *modulo-scheduled pipelines (MS-pipelines)* and the underlying theory for scheduling them. In Section 2.1, we briefly introduce the terminology used in classical pipeline theory. Section 2.2 deals with an introduction to software pipelining. Next, we demonstrate the need for MS-pipelines in Section 2.3 with the help of several motivating examples. Finally, in Section 2.4, we present the problem statement for MS-pipeline theory.

2.1. Terminology

First let us define a number of terms used in classical pipeline theory [14].

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

(a)



(b)

Figure 1. An example reservation table and its state diagram.

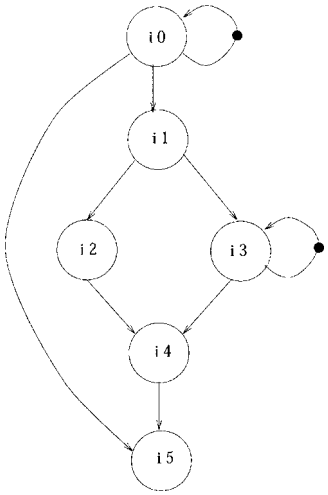
The resource usage of various stages of a hardware pipeline is represented by a two dimensional *Reservation Table*. An × mark in i-th row j-th column (henceforth referred to as (i, j)) indicates that the i-th stage is required j time steps after the operation was initiated. The time between the initiations of two operations is termed as *latency*. A latency is set to cause a *collision* if the two operations 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. If two operations entering a pipeline l cycles apart do not cause a collision, then l is termed a **permissible latency**; otherwise it is **forbidden**. For the reservation table shown in Figure 1(a), latencies 2 and 3 are forbidden; all other latencies, i.e., $1, 4, 5, \cdots$ are permissible.[1]

Classical pipeline theory identifies initiation sequences or *latency sequences* which maximize the throughput and the utilization of the pipeline using *state diagrams* [14]. Each state in the state diagram is represented by a *collision vector* which specifies the forbidden latencies (indicated by **1**) and permissible latencies (indicated by **0**) in the current state. Arcs in the state diagram indicate the state transition by allowing a new initiation from the current state at a specified (permissible) latency. Refer to [14] for the construction of the state diagram. From the state diagram shown in Figure 1(b), we can identify a *latency cycle* $\{1, 4\}$ that yields 2 initiations in 5 cycles. This latency cycle $\{1, 4\}$ yields the maximum throughput of $\frac{2}{5}$ initiations, with a **period** of 5, for the given reservation table. As mentioned in the Introduction, a latency cycle and its period are solely determined by the resource usage of the operation, i.e., the reservation table associated with the pipeline.

2.2. Software Pipelining

Modulo scheduling or software pipelining [22], [15], [13], [23], [5], [24] is a compiling technique to extract higher instruction-level parallelism in loops. We will briefly introduce the terminology used in software pipelining with the help of an example.

Consider the low level program code shown in Figure 2(a). The corresponding data dependency graph is shown in Figure 2(b). Assume the execution time of an integer add is 1 time unit, load/store is 3 time units, and floating point multiply is 5 units. Further,



i_0:	r8 = r8 + 4	% r8 is address of a[i] %
i_1:	r10 = load m(r8)	% r10 = a[i] %
i_2:	r11 = r10∗r10	% r11 = a[i]*w1 %
i_3:	r13 = r13∗r10	% s = s ∗ a[i] %
i_4:	r14 = r13∗r11	% temp = s ∗ a[i] ∗ w1 %
i_5:	store(r14, m(r8))	
	branch to $i0$ if $i \leq n$	
	enddo	

(a) Program Representation (b) DDG

Figure 2. An example loop and its data dependence graph.

assume that the load/store unit is fully pipelined (with a simple reservation table), while the reservation table in Figure 1(a) represents the resource usage of a multiply unit.

Loop-carried dependences put a lower bound, **RecMII**, on the initiation interval of the software pipelined schedule. The value of **RecMII** is determined by the critical (dependence) cycle(s) [26] in the Data Dependency Graph (DDG). Specifically

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

along the critical cycle(s). For the DDG in Figure 2(b) the critical cycle involving instruction i_3 enforces the lower bound due to recurrences. That is, $\mathbf{RecMII} = 5$.

Another lower bound **ResMII** on **II** is enforced by resource constraints. Suppose there are 1 integer ALUs, 1 Load/Store units, and 1 of each floating point add, multiply and divide units, There are three multiply instructions in the DDG. Each multiply operations requires stage 2 (and stage 3) of the pipeline for exactly 2 cycles. Thus, in order to process the 3 instructions of the DDG using 1 multiply unit, we need at least $(2 + 2 + 2=)6$ cycles. Mathematically,

$$\mathbf{ResMII} \geq \left\lceil \frac{N * d_{\max}}{F} \right\rceil$$

where $d_{\max}$ represents the maximum number of cycles for which any stage of the pipeline is used, N is the number of operations to be executed in the pipe, and F is the number of available pipes. If there is only one FP Multiply unit, then

Table 1. A Schedule for the Motivating Example with **II** = 10

	Time Steps																			
Iteration	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Iter. 0	i_0	i_1			i_2	i_3					i_4					i_5				
Iter. 1											i_0	i_1			i_2	i_3				

Table 2. Modulo Reservation Table for the Schedule with **II** = 10

	Time Steps									
Resource	0	1	2	3	4	5	6	7	8	9
Integer	$i_0(1)$									
Ld/St—Stage 1		$i_1(1)$				$i_5(0)$				
Ld/St—Stage 2			$i_1(1)$				$i_5(0)$			
Ld/St—Stage 3				$i_1(1)$				$i_5(0)$		
FP Mult—Stage 1	$i_4(0)$			$i_4(0)$	$i_2(1)$	$i_3(1)$		$i_2(1)$	$i_3(1)$	
FP Mult—Stage 2		$i_4(0)$		$i_4(0)$		$i_2(1)$	$i_3(1)$	$i_2(1)$	$i_3(1)$	
FP Mult—Stage 3			$i_4(0)$				$i_2(1)$	$i_3(1)$		
FP Mult—Stage 4					$i_4(0)$				$i_2(1)$	$i_3(1)$

$$\mathbf{ResMII} \geq \left\lceil \frac{3 * 2}{1} \right\rceil = 6.$$

Similar lower bounds on **ResMII** based on load/store unit and integer unit are $\lceil 2/1 \rceil$ and $\lceil 1/1 \rceil$ respectively. The maximum of these sets the lower bound **ResMII**. Hence, $\mathbf{ResMII} = 6$.

Lastly, the Minimum Initiation Interval **MII** is the maximum of **RecMII** and **ResMII**. That is, $\mathbf{MII} = \texttt{max}\ (\mathbf{RecMII}, \mathbf{ResMII})$. For our example, $\mathbf{MII} = \texttt{max}\ (5, 6) = 6$.

Any resource constrained schedule for the loop will have an initiation interval $\mathbf{II} \geq \mathbf{MII}$. A software pipelined schedule for an $\mathbf{II} = 10$ is shown in Table 1. The resource usage in a software pipelined schedule is represented by a *Modulo Reservation Table (MRT)* [22, 23] which consists of m rows and **II** columns, where m is the total number of resources in the architecture. The resources include the different stages of a functional unit. In our architecture we shall assume that the Integer Unit has a single stage, and the Load/Store unit has 3 stages. An entry (s, t) in the MRT is 1 or 0 depending on whether or not the s th stage is used at time step t. The MRT for the above schedule is shown in Table 2 where, for illustrative purposes, the usage of a resource is represented by means of the instruction that uses the stage at time t. Further, the iteration number k shown in brackets next to instruction i represents that instruction i from $(j - k)$th iteration, for any j, uses the resource at the given time step.

	Time Steps					
Stage	0	1	2	3	4	5
1	$\widehat{i_2}$	i_3		i_2	$\widehat{i_3}$	
2		i_2, i_3		i_2		i_3
3	i_3		i_2			
4			i_3		i_2	

(a)

	Time Steps					
Stage	0	1	2	3	4	5
1	$\widehat{i_2}$		i_3	i_2		$\widehat{i_3}$
2	i_3	i_2	i_3	i_2		
3		i_3	i_2			
4				i_3	i_2	

(b)

Figure 3. MRTs corresponding to initiations with different latencies.

2.3. Modulo-Scheduled Pipelines

In a software pipelined schedule, the different instances of an instruction i corresponding to the different iterations are initiated with an initiation interval **II**. We presented a schedule for $\mathbf{II} = 10$ in the previous section. But what about for values of $\mathbf{II} \geq \mathbf{MII} = 6$? Let us consider $\mathbf{II} = 6$. Instructions i_2, i_3, and i_4 are to be scheduled on the FP multiply unit whose reservation table is shown in Figure 1(a). Classical pipeline theory analysis revealed that $\{1, 4\}$ is a permissible latency cycle, accommodating 2 initiations in 5 cycles. However, this latency cycle does not "fit" the current $\mathbf{II} = 6$. To illustrate this, assume instructions i_2 and i_3 are scheduled at time steps 0 and 4, with a latency 4 between them. The resource usage of these initiations is represented in a *Modulo Reservation Table* (MRT) in Figure 3. Only a part of the MRT corresponding to the FP Multi unit is shown in this figure. In the MRT, the resource usage beyond time step 6 wraps around, i.e., an usage in time step t is shown in column $t \bmod 6$. Additionally we use the symbol "$\widehat{\ }$" to indicate the time step at which an operation was initiated. We observe that both initiations require stage 2 at time step 1. Thus, even though 4 is a permissible latency for the given pipeline (according to the classical pipeline theory), it causes a collision under modulo scheduling with an $\mathbf{II} = 6$. The collision occurs in the software pipelined cycle due to the wrap-around resource usage. Classical pipeline theory does not address such collisions which are caused mainly due to the wrap-around resource usage.

The latency cycle $\{1, 5\}$ with a period 6 "matches" with given initiation interval $\mathbf{II} = 6$. Two initiations started at time steps, say 0 and 5, does not cause any collision as shown in Figure 3(b). Under this initiation, stages 1 and 2 of the pipeline are used in 4 out of the 6 cycles, i.e., have an utilization of only 66.67%. However, we have three instructions i_2, i_3, and i_4 in our loop. None of the latency cycles of this pipeline allow three initiations of 6 cycles. Thus, for the given reservation table, one has to resort to a higher **II** in order to accommodate the 3 initiations in a single pipeline. For the given reservation table, 3 initiations are possible in the pipeline only for values of $\mathbf{II} \geq 9$. This increase in **II** corresponds to a decrease in the computation rate of the software pipelined loop by 50%. This gives raise to the two questions which motivate our work on MS-pipeline theory.

	Time Steps					
Stage	0	1	2	3	4	5
1		×			×	
2		×		d	×	
3			×			
4						×

(a)

	Time Steps					
Stage	0	1	2	3	4	5
1	$\widehat{i_2}$	i_4	$\widehat{i_3}$	i_2	$\widehat{i_4}$	i_3
2	i_3	i_2	i_4	i_3	i_2	i_4
3	i_4		i_2		i_3	
4		i_3		i_4		i_2

(b)

Figure 4. Modified reservation table supporting 3 initiations.

2.4. Motivation

The previous subsection shows that latency cycles obtained from classical pipeline theory may or may not be permissible under modulo scheduling. Thus, how does one identify latency cycles that yield higher throughput and are permissible for a given **II** under modulo scheduling? Secondly, is it at all possible to "adjust" the reservation table of the pipeline to accommodate 3 initiations in 6 cycles? Patel and Davidson [19] have addressed a similar issue in the case of hardware pipelines. By inserting delays (non-compute stages) in the pipeline, they have proposed a method that reconfigures the pipeline to achieve higher throughput or utilization. Hence, the last question really is, can their method [19] be adapted to modulo scheduled pipelines (MS-pipelines). The first question was partly addressed in our earlier work on Co-Scheduling [10]. In this paper, we develop the theory of MS-pipelines which is useful to easily identify permissible latency sequences. Further, the MS-pipeline theory helps to address the last question, which can be formally stated as:

> *Given a pipeline structure (its reservation table) and an initiation interval* **II***, is it possible to improve the utilization of the pipe by modifying the usage pattern (keeping the resources usages intact)?*

It can be seen that this question directly relates to the question raised in the previous subsection, namely whether 3 operations can be initiated in the pipeline with an $\mathbf{II} = 6$ cycles. This is indeed possible by modifying the reservation table as shown in Figure 4(a), where a delay represented by "d" was introduced at time step 3 in stage 2. Figure 4(b) shows the MRT, with 3 initiations at time steps 0, 2, and 4.

It is easy to see how this relates to ASIP or embedded processor design. Suppose there is a time critical loop where it is important to achieve the lowest possible **II**, and the resource usage of some pipeline is not "amenable" to that **II**. Then the ASIP designer can modify the resource usage of the pipeline so as to achieve a given number of initiations (or a required utilization) of the pipeline through the proposed delay insertion method. This enables the ASIP designer to co-schedule the hardware and software pipelines together to achieve the maximum performance.

3. Related Work

Classical pipeline theory was developed almost 2 decades ago and was successfully employed to improve the throughput and the utilization of pipelined and vector architectures [14], [19]. Such approaches select a latency cycle having the minimum average latency, and schedule operations in the pipeline repetitively based on the period of the latency cycle. Patel and Davidson proposed delay insertion to improve the performance of hardware pipelines [19]. Their approach introduced delays in the reservation table to support latency cycles that achieve the *optimal* minimum average latency. An important point to observe here is that the period of the latency cycle is solely determined by the pipeline structure. We refer to the above scheduling as **cyclic scheduling of pipelines**. The theory of cyclic scheduling of pipelines has been applied to cyclic job-shop scheduling in manufacturing systems [3]. In contrast to cyclic scheduling of pipelines [19], [14], [3], the initiation interval (**II**) of **Modulo-Scheduled Pipelines** depends on the recurrences in the loop and the resource availability.

Recently, ideas from hardware pipeline theory have been used to develop an FSA-based approach to perform instruction scheduling for pipelined architectures involving structural hazards [18], [21], [2]. However, these instruction scheduling methods are for straight-line code and do not consider software pipelining. Further, these FSA-based methods follow a *greedy* approach, attempting to schedule an operation in the current cycle if it does not cause a structural hazard. In contrast, our approach in this paper and the work reported in [10] considers *all* latency sequences and chooses the one that maximizes the throughput of the MS-pipeline. Lastly, several examples are given in [10], [11] to illustrate that neither classical pipeline theory nor the FSA-based approaches can directly be used to analyze pipeline structures operating under modulo scheduling.

A comprehensive survey of software pipelining methods can be found in [23]. A number of heuristic methods for software pipelining have been proposed, starting with initial work of Rau and Glaeser [22] and its application in the FPS compiler and Cydra 5 compiler [4], [5]. Lam proposed a resource-constrained software pipelining method using list scheduling and hierarchical reduction of cyclic components [15]. Huff's Slack Scheduling [13] is also an iterative solution which gives priority to scheduling nodes with minimum *slack* in the time at which they can be scheduled, and tries to schedule a node at a time which minimizes register pressure. Other heuristic-based scheduling methods have been proposed by Gasperoni and Schwiegelshohn [7], Wang et al. [27], and Rau [24]. We have proposed an integer linear programming formulation based approach for optimal software pipelining in [8], [9], [1]. Unlike the heuristic approaches, the integer linear programming based formulation obtains an optimal solution (i.e., constructs a software pipelined schedule at the lowest possible **II**), although at the expense of a large computation time. Some of the software pipelining methods [13], [6], [17] also concentrate on minimizing register pressure.

The theory of MS-pipelines has direct application in software pipelining, as established in our Co-Scheduling framework [10], [11]. A salient feature of the MS-pipeline theory is that it integrates scheduling constraints posed by the pipeline structure and the modulo initiation interval. To the best of our knowledge this is the first attempt to tune the

classical pipeline theory in a form suitable for software pipelining. The proposed theory of MS-pipelines and the methods developed in this paper to improve their performance are directly useful to any modulo scheduling algorithm.

In this paper we have developed the MS-pipeline theory for architectures in which the processors do not share any resources. However, the MS-pipeline theory and Co-Scheduling have been extended in [30] for pipelines that share resources; e.g., the normalization stage being shared by the floating point add and floating point multiply pipelines. In such cases the size of the MS-state diagram could grow extremely large even for moderate values of **II**. In [11], we have established a simple condition for identifying distinct paths in the MS-State diagram. Further, our earlier work in [30] presents two alternative efficient approaches to construct the MS-state diagram. These results make Co-Scheduling a viable approach for analyzing and designing pipelines for ASIPs and embedded processors. The theory developed in this paper is also useful in the context of reconfigurable architectures [28], [29] where the pipeline resource usage can be changed from one loop to another to improve the utilization of hardware pipelines.

Lastly, the work proposed in this paper is significantly different from those reported in [10], [11]. The work in [10], [11] deals with the application of MS pipeline theory in a software pipelining method. More specifically, they deal with, given a pipeline architecture with structural hazards, how the use of state diagram and the latency sequences can improve the software pipelining method. They do not deal with issues relating to improving the number of initiations in a pipeline for a given **II**, whereas this paper proposes a method to achieve that by modifying the resource usage pattern through delay insertion. Further, although the underlying theoretical base is common (part of Section 4), the earlier work [10], [11] did not deal with establishing the MS-pipeline theory and its analysis, specifically Theorems 5.1, 5.2, and 6.1.

4. Theory of Modulo-Scheduled Pipelines

In this section, we begin with a brief review of the basic concepts of MS-pipelines [10]. Then we develop a method for constructing the state diagram of MS-pipelines and establish its correctness.

Throughout this paper, we consider only *static* [14] pipelines, whose resource usage pattern can be described by a single reservation table. Thus each (static) pipeline of an architecture (e.g. FP Add, FP Multiply, FP Divide) needs to be analyzed independently and the results of these analyses can be used in constructing the software pipeline schedule involving these different operations.

4.1. Review of Basic Concepts

As noted earlier, the reservation table of a hardware pipeline is represented by an $m \times l$ reservation table where m is the number of stages in the pipeline and l is the execution time (latency) of an operation executing on the FU. Let $d_{\max}$ denote the maximum

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

(a) Cyclic Reservation Table for **II** = 4

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

(b) Cyclic Reservation Table for **II** = 6

Figure 5. A cyclic reservation tables for different **II**s.

number of cycles for which any stage of the pipeline is needed. To represent the resource usage of MS-pipelines under the modulo scheduling framework, we extend the reservation table to a *cyclic reservation table (CRT)*. The CRT is obtained as follows. (1) If $l < \mathbf{II}$, the reservation table is extended to **II** columns (with the additional columns all empty). (2) If $\mathbf{II} < l$, the reservation table is folded. An × mark at (s, t) in the original reservation table appears at time step $t \bmod \mathbf{II}$ in the s-th row of the folded reservation table.[2] (3) If $\mathbf{II} = l$, nothing need be changed. As an example, the reservation table in Figure 4(a) yields the cyclic reservation tables shown in Figure 5(a) and Figure 5(b) for $\mathbf{II} = 4$ and $\mathbf{II} = 6$.

Next we define forbidden and permissible latencies for MS-pipelines.

DEFINITION 4.1 *A latency $f < \mathbf{II}$ is said to be a **forbidden latency** if there exists a row s in the CRT such that both (s, t) and $(s, (t+f) \bmod \mathbf{II})$ of the CRT contain an × mark. A latency f that is not forbidden is termed **permissible**.*

It can be easily seen that in an MS-pipeline, a latency value f greater than **II** is equivalent to $f \bmod \mathbf{II}$. Further, if f is a forbidden latency, then $\mathbf{II} - f$ is also forbidden. The latency value 3 is forbidden in the CRT in Figure 5(b). All other latencies are permissible. Hence the permissible latency set is $\{1, 2, 4, 5\}$.

The following property is satisfied by the permissible latency set.

LEMMA 4.1 *Let $S_0 = \{p_1, \cdots, p_k\}$ be the permissible latency set for an MS-pipeline with initiation interval **II**. If $p_1, p_2, \cdots, p_k$ are in (strictly) ascending order, then*

$$\mathbf{II} = p_1 + p_k = p_2 + p_{k-1} = \cdots = p_{\lceil k/2 \rceil} + p_{\lceil (k+1)/2 \rceil}. \tag{1}$$

Proof. Since $p_1, p_2, \cdots, p_k$ are in the ascending order, $\mathbf{II} - p_1, \mathbf{II} - p_2, \cdots, \mathbf{II} - p_k$ are in the descending order. Also, $p_k, p_{k-1}, \cdots, p, p_1$ is a descending sequence. By the definition of permissible latencies, if p_i is permissible then $\mathbf{II} - p_i$ is also permissible. Further, as p_1 is the smallest permissible latency, $\mathbf{II} - p_1$ must correspond to the largest permissible latency, which must be p_k. That is,

$$\mathbf{II} - p_1 = p_k \text{ or } \mathbf{II} = p_1 + p_k.$$

Likewise, $\mathbf{II} - p_2$ is the second largest permissible latency and hence corresponds to p_{k-1}. Therefore,

$$\mathbf{II} - p_2 = p_{k-1} \text{ or } \mathbf{II} = p_2 + p_{k-1}.$$

Likewise the other equalities, represented by $\cdots$ in Equation 1 can be established. Lastly, if k is odd, $p_{\lceil k/2 \rceil} = p_{\lceil (k+1)/2 \rceil}$ is the middle element. Otherwise $p_{\lceil k/2 \rceil}$ and $p_{\lceil (k+1)/2 \rceil}$, are the two middle elements. Hence,

$$\mathbf{II} = p_{\lceil (k/2) \rceil} + p_{\lceil ((k+1)/2) \rceil}. \qquad \square$$

As an example, the permissible latency set of the CRT of Figure 5(b) is $\{1, 2, 4, 5\}$ and $\mathbf{II} = 6$. We have $1 + 5 = 2 + 4 = 6 = \mathbf{II}$. From now on, the initial permissible latency set is always assumed to be in the ascending order.

The number of initiations possible in a pipeline is bounded by two factors, *UB_perm* and *UB_res* [10].

THEOREM 4.1 [10] *The upper bound (henceforth referred to as UB_Init) on the number of initiations made in an MS-pipeline during* $\mathbf{II}$ *cycles is*

$$UB\_Init = \max(UB\_res,\ UB\_perm), \quad \text{where } UB\_res = \left\lfloor \frac{\mathbf{II}}{d_{\max}} \right\rfloor,\ UB\_perm = (k+1),$$

k is the cardinality of the permissible latency set and $d_{\max}$ is the maximum number of $\times$ marks in any row in the reservation table.

Proof. By definition, instructions can be initiated only on a permissible latency. Further, at most one instruction can be initiated at each permissible latency; another instruction initiated at this time step violates the modulo scheduling constraint and thereby causes structural hazards at every stage. Hence the value of *UB_perm* should be one more than the number of permissible latencies (accounting one extra initiation at time step 0). The second bound *UB_res* 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. $\square$

4.2. Modulo-Scheduled (MS) State Diagram

The MS-state diagram is somewhat similar to, but different from, the state diagram representation in classical pipeline theory [14]. The difference between the two will be discussed subsequently.

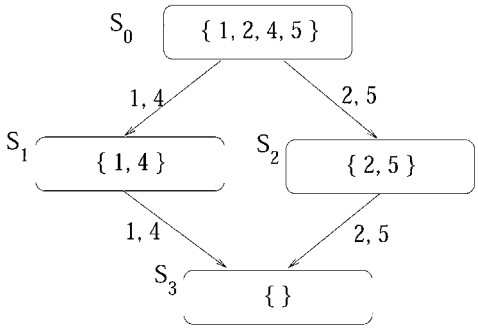


Figure 6. MS-state diagram.

Procedure 4.1. Construction of 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 latency set of the new state is given by $S' = S_{-p_i} \cap S_0$ where S_{-p_i} is defined as

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

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**. 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}.

As an example, consider the CRT shown in Figure 5(b) with $\mathbf{II} = 6$. The initial set of permissible latencies is $S_0 = \{1, 2, 4, 5\}$. The MS-state diagram is shown in Figure 6. 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 a path correspond to the latencies between successive initiations. Successive initiations are made at time steps 0, p_1, $(p_1 + p_2) \bmod \mathbf{II}$, $\cdots$, $(p_1 + p_2 \cdots, p_{(k)}) \bmod \mathbf{II}$; we refer to these values as the **offset values** from the first initiation made at time 0. For example, the path

$S_0 \xrightarrow{1} S_1 \xrightarrow{4} S_3$ corresponds to initiations at offset values 0, $(0+1) \bmod \mathbf{II}$, and $(0+1+4) \bmod \mathbf{II}$, that is, 0, 1, and 5. 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**.

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 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 in the longest path from the initial state S_0. Lastly, we say that a node S is **at a distance** d if the length of the (longest) path from S_0 is $(d-1)$.

DEFINITION 4.2 *The **final state** of an MS-state diagram is one which contains an empty permissible latency set.*

The longest path from the initial to the final state in the MS-state diagram corresponds to the maximum number of initiations (*Max_Init*) achievable in the MS-pipeline. It is important to observe here that *Max_Init* computed from the longest path in the MS-state diagram is what is **actually achievable** in the MS-pipeline. The bound on *UB_Init* due to Theorem 4.1 is such that $Max\_Init \leq UB\_Init$. That is, in some MS-state diagrams the *Max_Init* achievable may be less than *UB_Init*. Lastly, from this longest path (to the final state) one can identify the offset values at which these initiations can be made. This information on the offset values can be used in a modulo scheduling algorithm to construct software-pipelined schedules. It was demonstrated in [10], [11] that the above analysis can be successfully applied in modulo scheduling. Such an approach facilitated obtaining schedules with lower initiation interval and also constructing them in shorter compilation time.

Next we will establish certain properties of MS-state diagram.

4.3. Properties of MS-State Diagram

The state diagrams constructed using classical pipeline theory [14] uses *collision vectors* to represent the set of permissible latencies in the current state. In our representation, instead, the set of permissible latencies itself is used directly.[3]

First we will establish the correctness of MS-state diagram.

THEOREM 4.2 *The latency set associated with any state S in the MS-state diagram represents all permissible latencies in that state, taking into account all initiations made to reach the state S.*

Proof. This theorem is proved using induction on the length of the (longest) path of state S. For the initial state S_0, the theorem is obviously true. Assume it holds for any state having

a path of length less than or equal to k. Let S be a state at a distance k. Let there be an arc from S to S' labeled p_i. That is, S' can be reached from S with an initiation p_i cycles after S. According to the inductive hypothesis, p_i is a permissible latency in S. Thus, the arc from S to S' with a latency p_i represents a valid initiation. After this initiation, corresponding to each permissible latency $p_j \in S$, $p'_j = (p_j - p_i) \bmod \mathbf{II}$ becomes a **may be** permissible latency in the new state S'. The role of mod operation in the above equation is obvious for all $p_j < p_i$. The latency p'_j in S' is permissible only if p'_j is in the initial permissible latency set S_0. Thus, the intersection (with S_0) in Step 2 of the construction procedure of the MS-state diagram (Procedure 4.1) guarantees that p'_j is a permissible latency in S'.

To complete the proof, we need to show that every latency that is permissible in state S' is included in the permissible latency set S'. Assume p is a permissible latency in state S', but is not included in the latency set S'. Either (i) $(p + p_i) \bmod \mathbf{II} \notin S$ or (ii) $p \notin S_0$. But, since p is a permissible in the state S', p must be in S_0. Hence (i) must hold. The latency value p in state S' corresponds to $(p + p_i) \bmod \mathbf{II}$ in state S. However, since p is permissible in S', $(p + p_i) \bmod \mathbf{II}$ must be permissible in S. But, by (i), $(p + p_i) \bmod \mathbf{II} \notin S$. Thus, there is a latency that is permissible but not included in the permissible latency set in a state at a distance k. This contradicts the inductive hypothesis. □

In an MS-state diagram, as we go from the start state to the final state, the number of permissible latencies monotonically decreases, i.e., only fewer initiations are possible.

LEMMA 4.2 *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 4.1, 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 $|S| > |S'|$. □

LEMMA 4.3 *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 4.2,

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

which is impossible. □

LEMMA 4.4 *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 4.5 *The construction of the MS-State diagram (**Procedure 4.1**) terminates after a finite number of steps.*

Proof. The proof of this lemma follows from Lemmas 4.2, 4.3, and 4.4. □

One can verify that Lemmas 4.2 to 4.4 hold for the MS-state diagram shown in Figure 6.

5. Achieving Maximum Initiations in MS-Pipelines

The construction method for MS-state diagram described in the last section is useful in identifying *Max_Init*, while Theorem 4.1 (in Section 4.1) gives an upper bound *UB_Init* for the number of initiations in an MS-pipeline. The natural questions then are: (1) Under what condition(s) does *Max_Init* equals *UB_Init*? (2) When $Max\_Init < UB\_Init$, is it possible to improve the *Max_Init*? We answer the first question in this section by analyzing the MS-state diagram. As will be seen later this analysis provides useful guidance to pipeline designers. Section 6 deals with the second question.

5.1. Cardinality of the Permissible Latency Set

First, we define a special form arithmetic progression.

DEFINITION 5.1 *An arithmetic progression (A.P.) of the form $p, 2p, \cdots, k.p$ is called a **special form A.P.***

In a special form A.P. the difference between two successive elements is same as the first element of the A.P. For example the sequence (3, 6, 9) is a special form A.P. Special form A.P.s help to characterize the performance of MS-pipelines as shown in the following theorem. The following theorem establishes the necessary and sufficient condition for achieving the upper bound on the number of initiations governed by cardinality of the permissible latency set. In doing so, the theorem also provides practical guidance to hardware designers in getting maximum utilization from their pipelines. In the following discussion we assume that the cardinality of the latency set is less than or equal to $\lfloor(\mathbf{II}/d_{\max})\rfloor$.

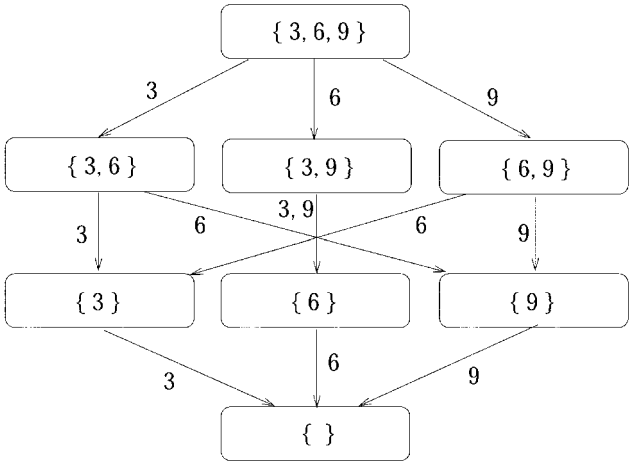


Figure 7. MS-state diagram for permissible latency set $= \{2, 4, 6, 8\}$.

THEOREM 5.1 *Assume an MS-pipeline with a modulo initiation interval* **II** *and the cardinality of the permissible latency set* k. *Then* $(k + 1)$ *initiations are possible in this pipe* ***if and only if*** *the initial permissible latency set forms a special form A.P.*

Let us consider an example. Let $S_0 = \{3, 6, 9\}$ and $\mathbf{II} = 12$. Here $k = 3$. The MS-state diagram shown in Figure 7 has the longest path with path length $\mathcal{L}(P)$ equal to 3, and $k + 1 = \mathcal{L}(P) + 1 = 4$ initiations are possible.

Proof. **[If-part]** It is given that the initial permissible latency set forms a special form A.P., say $S_0 = \{p, 2p, \cdots, k.p\}$. We need to show that there exists a path of length k. The proof is by construction of the path $S_0 \xrightarrow{p} S_1 \xrightarrow{p} S_2 \cdots \xrightarrow{p} S_k$. First, choose p as the latency from S_0 to reach S_1. One can easily verify that

$$S_1 = S_{0_{-p}} \cap S_0 = \{0, p, 2p, \cdots, (k-1)p\} \cap S_0 = \{p, 2p, \cdots, (k-1)p\}.$$

Again we choose p as the latency to reach state S_2 from S_1. Then $S_2 = \{p, 2p, \cdots, (k-2)p\}$. Proceeding this way, $S_{k-1} = \{p\}$ and $S_k = \{\}$. □

We prove the **[only-if]** part in the following way. First we show that when $(k + 1)$ initiations are possible, each one of these initiations start at a different time step in the MRT; further each offset value of these initiations equal to a unique permissible latency. Using this result we will show that two permissible latencies p_i and p_{i+1} are separated by p_1; i.e., for all p_i, $p_{(i+1)} - p_i = p_1$. From this, it follows that the permissible latencies form a special form A.P.

Next, we define a CL-path as follows.

DEFINITION 5.2 *A path in the MS-state diagram with path length equal to the cardinality of the (initial) permissible latency set is said to be a* ***Cardinal Length (CL) path****.*

Note that an MS-state diagram may or may not have a CL-path. Theorem 5.1 states that an MS-state diagram has a CL-path if and only if the initial permissible latency set is in a special form A.P. From the **if**-part of Theorem 5.1, it is also possible to show that every path in the MS-state diagram whose initial permissible latency set is a special form A.P., is a CL-path; but we will skip the proof.

The following lemma shows that the offset values corresponding to any path in the MS-state diagram form a subset of the permissible latency set. Further, it shows that no two offset values are equal. As an example, consider the path $S_0 \xrightarrow{6} S_3 \xrightarrow{2} S_5$ with $l_1 = 6$ and $l_2 = 2$. Here $\mathbf{II} = 7$. Clearly, $l_1 = 6 \in S_0$ and

$$(l_1 + l_2) \bmod \mathbf{II} = 8 \bmod 7 = 1 \in S_0.$$

Further, $l_1 = 6$ is distinct from $(l_1 + l_2) \bmod \mathbf{II} = 1$.

LEMMA 5.1 *Let* $S_0 = \{p_1, p_2, \cdots, p_k\}$ *and there be a path* $S_0 \xrightarrow{l_1} S_1 \xrightarrow{l_2} \cdots \xrightarrow{l_r} S_r$ *in the MS-state diagram. Then*

$$l_1 = p_{i_1}; \quad (l_1 + l_2) \bmod \mathbf{II} = p_{i_2}; \quad \cdots \quad (l_1 + l_2 + \cdots + l_r) \bmod \mathbf{II} = p_{i_r}$$

where $i_1, i_2, \cdots, i_r$ *are distinct index values taken from* $[1, k]$.

Proof. For each initiation S_i, the corresponding offset value is $O_i = (l_1 + l_2 + \cdots + l_i) \bmod \mathbf{II}$; Now O_i must equal some permissible latency p_j. Otherwise, the latency between S_i and S_0 is forbidden, which contradicts the fact that $S_0 \xrightarrow{l_1} S_1 \xrightarrow{l_2} \cdots \xrightarrow{l_r} S_r$ is a path in the MS-state diagram. Further, the offset values O_i and O_j corresponding two initiations i and j must be distinct; otherwise it violates the modulo scheduling constraint—each initiation is repeated once every **II** cycle. Hence the Lemma. □

COROLLARY 5.1 *If there exists a CL-path* $S_0 \xrightarrow{l_1} S_1 \xrightarrow{l_2} \cdots \xrightarrow{l_k} S_k$ *in the MS-state diagram, then the set containing the* offset *values of the above initiations is a permutation of* S_0.

This is a special case of the previous lemma with $r = k$. □

LEMMA 5.2 *If* $S_0 = \{p_1, p_2, \cdots, p_k\}$ *is the initial permissible latency set, and there exists a CL-path in the MS-state diagram, then for any* $i, j \in [1, k]$ *and* $i \neq j$, $(p_i - p_j) \bmod \mathbf{II} \in S_0$.

Proof. Since there exists CL-path in the MS-state diagram, from Corollary 5.1, the offsets of the last k initiations (from the first initiation at time 0) is a permutation of the

permissible latency set. That is, there is one initiation at an offset value equal to *each* of the permissible latencies. Thus for any two initiations, with offsets p_i and p_j, $i \neq j$, to be valid, i.e., does not cause a collision, the latencies between these two initiations must be permissible. That is, $(p_i - p_j)$ mod **II** must be in the initial permissible latency set. □

Note that the above lemma holds only for MS-state diagram having a CL-path. For example, if $S_0 = \{2, 3, 5, 7, 8\}$ then the difference between the permissible latencies 3 and 7 is not a permissible latency. That is, $7 - 3 = 4$ does not belong to S_0. As the last step in the preparation for proving the **[only if]**-part of Theorem 5.1, we establish the following lemma.

LEMMA 5.3 *Let* $S_0 = \{p_1, p_2, \cdots, p_k\}$. If $(p_j - p_i)$ mod **II** $\in S_0$, *for any* $p_i \in S_0$, *and for all* $p_j \in S_0$ *such that* $p_j \neq p_i$, *then* $p_{(i+1)} - p_i = p_1$.

Proof. It is given that all $(p_j - p_i)$ mod **II** $\in S_0$. In particular, $(p_{(i+1)} - p_i)$ mod **II** $\in S_0$. Since the elements of S_0 are in the ascending order, the elements $(p_{(i+1)} - p_i)$, $(p_{(i+2)} - p_i), \cdots, (p_k - p_i)$ are also in the ascending order. The modulo operation with **II** is omitted here since, for all $j \geq i$, $0 \leq (p_j - p_i) \leq$ **II**. Further, since each $p_j \in S_0$ is unique, the above elements are also unique.

We know that $p_k <$ **II**. Subtracting p_i from both sides we have,

$$p_k - p_i < \mathbf{II} - p_i.$$

But from Lemma 4.1, $p_{(k-i+1)} + p_i =$ **II**. Thus substituting $p_{(k-i+1)}$ for **II** $- p_i$, we have,

$$p_k - p_i < p_{(k-i+1)}.$$

Now, since $p_k - p_i$ is a permissible latency, and $p_{(k-i)} < p_{(k-i+1)}$,

$$p_k - p_i \leq p_{(k-i)}.$$

Since $p_{(k-1)} < p_k$, using $p_{(k-1)}$ in the place of p_k we get,

$$p_{(k-1)} - p_i < p_{(k-i)}.$$

As $p_{(k-1)} - p_i$ is a permissible latency, and, $p_{(k-i-1)} < p_{(k-i)}$, we get

$$p_{(k-1)} - p_i \leq p_{(k-i-1)}.$$

Proceeding further this way,

$$p_{(i+1)} - p_i \leq p_{(i+1-i)}; \quad \text{i.e. } p_{(i+1)} - p_i \leq p_1.$$

But p_1 is the smallest permissible latency in S_0 and therefore $p_{(i+1)} - p_i = p_1$. □

Now we are ready to prove the **[only-if]**-part of Theorem 5.1.

Proof. [**Only-if** part of Theorem 5.1] Since there are $k+1$ initiations possible, there must exist a CL-path k in the MS-state diagram. Then by Lemma 5.2, the difference (modulo **II**) between every pair of elements in S_0, is also in S_0. Now choosing p_i to be one of $p_1, p_2, \cdots, p_{k-1}$ and applying Lemma 5.3, we get

$$p_2 - p_1 = p_1; \quad p_3 - p_2 = p_1; \quad \cdots; \quad p_k - p_{(k-1)} = p_1;$$

which completes the proof. □

5.2. *A Sufficient Condition*

Theorem 5.1 states that in order to achieve *UB_Init*, when it is governed by the first bound in Theorem 4.1, the initial permissible latency set must be a special form A.P. This is a strong result; however its applicability is limited as it requires **all** permissible latencies to be in a special form A.P. In this subsection we present a weaker, but more useful, result which requires only a subset of the initial permissible latencies to be a special form A.P. Having such a subset is a sufficient condition for achieving a given number of initiations.

Consider the initial permissible latency set $S_0 = \{3, 4, 6, 7\}$ and $\mathbf{II} = 10$. S_0 contains subsequence $\{3, 6\}$, of cardinality 2, which forms a special form A.P. Hence there exists a path $S_0 \xrightarrow{3} S_1 \xrightarrow{3} S_2$ which corresponds to at least $2 + 1 = 3$ initiations. The following theorem formalizes this idea.

THEOREM 5.2 *If the initial permissible latency set S_0 contains a special form arithmetic progression of length l, then at least $(l+1)$ initiations are possible in the MS-pipe.*

Proof. The proof of this lemma is similar to the proof of the **[if-part]** of Theorem 5.1. By considering only those permissible latencies which form the special form A.P. it is possible to construct a path of length l in the MS-state diagram. Hence the Theorem. □

Theorem 5.2 provides only a sufficient but not necessary condition. The following example illustrates this. Assume $\mathbf{II} = 10$, $d_{\max} = 2$ and the initial permissible latency set $S_O = \{2, 3, 5, 7, 8\}$. It can be seen that there exists a path

$$S_0 \xrightarrow{3} S_1 \xrightarrow{2} S_2 \xrightarrow{3} S_3$$

in the MS-state diagram and thus *Max_Init* $= 4$. However the longest special form A.P. contained in the permissible latency set is of length 1. (Trivially every element forms a special form A.P. sequence of one element.)

6. Delay Insertion in MS-Pipelines

In this section we address the questions relating to improving *Max_Init* towards the upper bound. We adapt the delay insertion method of Patel and Davidson to MS-pipelines to

improve *Max_Init*. We show that, with the delay insertion method, it is always possible to realize the upper bound due to *UB_res*, namely $\lfloor \mathbf{II}/d_{\max} \rfloor$. It is important to note here that *UB_perm* no longer plays a role in determining *UB_Init*, as the permissible latency set itself changes with delay insertion.

6.1. *Delay Insertion to Improve the Number of Initiations*

In our motivating example, we saw that at most two initiations can be made for the reservation table shown in Figure 1 for an **II** of 6; i.e., $Max\_Init = 2$. However, *UB_Init* for that reservation table is 3. The question is can we somehow modify the reservation table to accommodate 3 initiations in the same pipe in 6 cycles. It was also seen that the modified reservation table shown in Figure 4(a) can, indeed, realize a *Max_Init* of 3. We develop a systematic approach to this problem in this section.

By Theorem 5.2, $(m + 1)$ initiations can be guaranteed if the permissible latency set S_0 contains a special form A.P. of length m. Thus if we can adjust the permissible latency set to contain an m-length special form A.P. then we are done. It must be noted that the existence of a special form A.P. is only a sufficient but not necessary condition. Thus it may be possible to achieve $(m + 1)$ initiations without S_0 containing an m-length special form A.P. For example, the MS-pipeline discussed in Section 5.2 with an initial permissible latency $S_0 = \{2, 3, 5, 7, 8\}$ supports 4 initiations, though its permissible latency set contains only a special form A.P. of size 1. In this paper we do not consider such alternative possibilities.

Theorem 5.2 suggests that if the permissible latency contains a special form A.P. of length 2, then 3 initiations can be guaranteed. Let the A.P. be $\{p, 2p\}$. Clearly,

$$2.p < \mathbf{II}; \quad \text{i.e., } p \leq \lfloor ((\mathbf{II} - 1)/2) \rfloor = \lfloor 5/2 \rfloor = 2.$$

In other words, we need to have an A.P. $\{p, 2p\}$ in the permissible latency set for any value of p in the range 1 to 2. For example when $p = 1$, let us try to include the A.P. $\{1, 2\}$ in permissible latency set. Let $P_i = \{1, 2\}$ denote the A.P. If the permissible latency set (denoted by P) has to include the elements 1 and 2 then it will also include their complements $\mathbf{II} - 1 = 5$ and $\mathbf{II} - 2 = 4$. Thus $P = \{1, 2, 4, 5\}$ is a permissible latency set that will guarantee at least 3 initiations. From this, the forbidden latency set F is $\{0, 3\}$. Thus, we need to modify the reservation table such that 0 and 3 are the only forbidden latencies. Patel and Davidson's method suggests that any row of the modified reservation table should have × mark in columns represented by the *compatibility classes* or *derived compatibility classes* of the forbidden latency set. We restate the definitions of compatibility and derived compatibility classes for the sake of easy reference.

DEFINITION 6.1 *Two elements f_1 and f_2 of a set F are* ***compatible*** *if $|f_1 - f_2|$ is in F.*

It can be seen that if $|f1 - f2|$ is in F, then $(f1 - f2)$ `mod` **II** is also in F.

DEFINITION 6.2 *A **compatibility class** with respect to F is a set in which all pairs of elements are compatible.*

An algorithm to compute all the compatibility classes is given in [14, p. 99].

DEFINITION 6.3 *If $C = \{c_1, c_2, \cdots, c_n\}$ is a compatibility class, then $C' = \{(c_1 + I) \bmod \mathbf{II}, (c_2 + I) \bmod \mathbf{II}, \cdots, (c_n + I) \bmod \mathbf{II}\}$ is a **derived compatibility class** for any integer I.*

As an example, if $F = \{0, 3, 4, 5, 6, 7\}$, then $\{0, 3, 6\}$, $\{0, 4, 7\}$ and $\{0, 5\}$ are compatible classes and $\{1, 4, 7\}$ is a derived compatible class. The forbidden latency set $\{0, 3\}$ is itself a compatibility class. Thus, if we choose the compatibility/derived compatibility classes $\{0, 3\}$, and $\{1, 4\}$ to represent the first two rows of the CRT respectively, we obtain the modified reservation table shown in Figure 4(a).

The insertion of delays in the reservation table may increase the execution time of an operation[4]; this in turn may increase **RecMII**, which may affect **II**. Thus the above method of modifying the reservation table is applicable only when the introduction of delays does not affect **II**. This is possible either (1) when the loop does not contain recurrence cycles involving operations that are executed in this pipeline; or (2) when **ResMII** dominates **RecMII**. It was observed in [24] that **ResMII** dominates **RecMII** in more than 85% of the loops that they have considered.

Procedure 6.1 formally describes this delay insertion method to achieve a given number of initiations.

Procedure 6.1. Delay Insertion to Achieve $(m + 1)$ Initiations

Step 1. Derive the MS-State diagram described in Procedure 4.1. If the length of the longest path l equal m, then the given CRT supports $(m + 1)$ initiations; Go to Step 4.

Step 2. For each p from 1 to $(\lfloor(\mathbf{II} - 1)/m\rfloor - 1)$ do

Step 2.1. $\{p, 2p, \cdots, mp\}$ be a subset of permissible latencies.

Step 2.2. The permissible latency set

$$P = \{p, 2p, \cdots, mp, \mathbf{II} - p, \cdots, \mathbf{II} - mp\}.$$

Step 2.3. Compute the forbidden latency set F

$$F = \{0, 1, 2, \cdots, \mathbf{II} - 1\} - P.$$

Step 2.4. Derive all the compatibility classes of F using the procedure described in [14, p. 99].

Step 2.5. If the size of the largest compatibility class is less than $d_{\max}$, the maximum number of × marks in any row, try next value of p; go to Step 2.1.

Step 2.6. For each row of the CRT choose an appropriate compatibility or derived compatibility class. Modify the CRT to match the chosen compatibility class. Go to Step 4.

Step 3. Report failure to support $(m + 1)$ initiations. Exit.

Step 4. Report modified CRT that support P. End.

One remaining question is: Does our method (Procedure 6.1) always succeed? In other words, is it *always* possible to support $(m + 1) \leq \lfloor \mathbf{II}/d_{\max} \rfloor$ initiations in a CRT? We answer this question affirmatively in the following section.

6.2. *Achieving the Upper Bound on Initiations*

We establish that *UB_Init* can always be achieved, by introducing sufficient delays in the CRT.

THEOREM 6.1 *Given an initiation interval* $\mathbf{II}$*, a CRT with at most* $d_{\max}$ × *marks on each row, it is* **always** *possible to achieve* $u = \lfloor \mathbf{II}/d_{\max} \rfloor$ *initiations by introducing delays in the CRT.*

Proof. We prove this theorem by constructing a modified CRT that supports the set $\{1, 2, \cdots, (u - 1)\}$, as a (sub)set of permissible latencies. We will only prove the theorem for the row(s) the CRT that consists $d_{\max}$ × marks. Other rows can also be modified in a similar way to support same permissible latency (sub)set.

Place the × marks at cycles $0, u, 2 * u, \cdots, (d_{\max} - 1) * u$. Thus the forbidden latency set contains

$$F' = \{0, u, 2 * u, \cdots, (d_{\max} - 1) * u\}.$$

It can be easily seen that the distance between any pair of × marks is already in the above subset of forbidden latencies F'. But, by the definition of forbidden latency, if f is forbidden, then $\mathbf{II} - f$ is also forbidden. Hence the following subset F'' is also contained in the forbidden latency set.

$$F'' = \{(\mathbf{II} - u), (\mathbf{II} - 2 * u), \cdots, (\mathbf{II} - (d_{\max} - 1) * u)\}.$$

Thus, the complete forbidden latency set for this CRT is $F = F' \cup F''$.

Note that the terms in F' are in the ascending order while that in F'' are in the descending order. Hence, if we can prove that the last term in F'' is greater than the first term in F', then all latencies in $[1, u - 1]$, are permissible. Since this forms a special form

A.P. of length $(u-1)$, at least $u = \lfloor \mathbf{II}/d_{\max} \rfloor$ initiations are guaranteed by Theorem 5.2. Thus, to complete the proof, we need to show that

$$\mathbf{II} - (d_{\max} - 1) * u \geq u.$$

Consider

$$L.H.S = \mathbf{II} - (d_{\max} - 1) * \left\lfloor \frac{\mathbf{II}}{d_{\max}} \right\rfloor \geq \mathbf{II} - (d_{\max} - 1) * \frac{\mathbf{II}}{d_{\max}} = \frac{\mathbf{II}}{d_{\max}}.$$

Introducing the floor function on the R.H.S,

$$L.H.S \geq \left\lfloor \frac{\mathbf{II}}{d_{\max}} \right\rfloor = u. \qquad \square$$

Theorem 6.1 shows that by placing the × marks at $u = \lfloor \mathbf{II}/d_{\max} \rfloor$ distance apart, we can always achieve u initiations. This also ensures that Procedure 6.1 will always succeed executing the iteration in Step 2 only once. That is, it will always be able the CRT supporting permissible latencies $(1, 2, \cdots, u)$.

7. Experimental Results

In order to get some idea about the usefulness of the theory developed in this paper, we conducted some experiments to address the following questions.

1. How useful is Theorem 5.2 in obtaining latency sequences that result in *Max_Init*?
2. How often does *Max_Init* achieve the maximum utilization, i.e., equal *UB_Init*?
3. In cases where *Max_Init* does not reach *UB_Init*, can the delay insertion method developed in Section 6 be used to achieve higher utilization? If so, how many delay cycles may be needed?
4. How does the performance of software pipelined schedules improve when delay insertion is applied?

We address questions 1–3 in Section 7.1 and question 4 in Section 7.2.

7.1. Experiments on MS Pipeline Theory

In the first part of the experiments, we considered several reservation tables which model the resource usage of different instructions (or instruction classes) of real processors. For each of these pipelines we generated the MS-state diagram for different **II** values, ranging

Table 3. Statistics of MS-State Diagram and Performance of Delay Insertion Method

Description	Number of Cases	%-age
Cases when *Max_Init* equals *UB_Init*	128	25.8
MS-State diagram consists of more than 10000 nodes	258	52.0
Theorem 5.2 yielded a *Max_Init* sequence	318	64.1
Description	**Arithmetic Mean**	**Median**
Number of states in MS-state diagram	5664	—
Ratio of *Max_Init* to *UB_Init*	0.76	0.69
Number of delays introduced to achieve *UB_Init*	9.5	1
Delays ratio to achieve *UB_Init*	0.26	0.06
Ratio of *Max_Init* obtained from Theorem 5.2	0.91	1.00

from 8 to 64. The MS-state diagram consists of a large number (greater than 10,000) of states for large values of **II**. Therefore, for pragmatic reasons, we restricted the size of the MS-state diagram to a maximum of 10,000 distinct states. Our algorithm also computes the longest path (corresponding to *Max_Init*). When *Max_Init* is less than *UB_Init*, our delay insertion method (Procedure 6.1) is applied to achieve a higher number of initiations I, for values of I from *Max_Init* and *UB_Init*. For each of these values, we compute how many delay cycles need to be inserted. In introducing delays, we have implemented a more realistic pipeline model in which delaying one stage of a pipeline at time t delays all subsequent resource usages (in all stages).

We collect the following statistics from our experimental setup. For the 10 reservation tables and the 56 different initiation intervals considered, a total of 496 combinations resulted.[5] In all these cases we measured the above parameters. The results are tabulated in Table 3. We observe that only in 128 (26%) of the 496 test cases, *Max_Init* equals the *UB_Init*. Further, on the average, *Max_Init* corresponds only to 69% (median) or 76% (arithmetic mean) of *UB_Init*. In the remaining 74% of the cases where *Max_Init* doesn't reach *UB_Init*, only 70% utilization of the MS-pipelines is achieved. However, in these cases, we find that by introducing a small delay, often 1 cycle, the upper bound can be reached in more that 57% of the cases. In other cases, with a delay of 1 cycle, upto 90% of *UB_Init* can be realized. Thus a significant performance improvement is expected when delays can be inserted in the pipelines.

The average value (arithmetic mean) for the delay introduced to achieve the *UB_Init* is 9; the median value for this is 1. In order to compare the delays introduced for different **II**s, we define a metric called *delay ratio* which is the ratio of delay cycles to **II**. The mean and median value for the delay ratio required to achieve *UB_Init* are 0.26 and 0.06 respectively. Further, we noticed that the use of Theorem 5.2 was quite effective. In 318 out of 496 test cases, the theorem obtained a latency sequence that results in *Max_Init*. On an average, in the 496 test cases considered, determining the longest special form of A.P. covers 91% of *Max_Init*. This result, reported in the last row of Table 3, shows that even though an MS-state diagram may involve several thousands of states, determining the longest special

form A.P. reveals a path in the state diagram that supports at least 90% of *Max_Init*. This implies that the state diagram construction time can be saved in many cases if the loop consists of fewer (that 90% of *Max_Init*) operations to be initiated in a given pipe.

7.2. Experiments on Software Pipelining

Next we consider the impact of delay insertion on software pipelining. To what extent is MS pipeline theory needed in scheduling loops, and how does it improve the performance? To address these questions, we considered a set of 35 software pipelineable loops extracted from Media Benchmarks [16], a common benchmark in embedded systems research. Further, it is quite likely that a single embedded processor, such as the one in a cell phone, may execute the GSM software or the MP3 software or MPEG decoder/encoder. Thus, the processor should be optimized for all the loops in these applications. However, we considered only a few tens of the loops due to various constraints, such as extracting the loops in a form compatible to our software pipeline scheduler.

We considered an architecture configuration with 2 **Integer** ALU's and one each of **Load**, **Store**, **FP Add**, **FP Multiply**, and **FP divide** units. Two different architectures A1 and A2 were considered in our experiments. Both these architectures had some structural hazards in some or all of the FUs, with architecture A2 having more structural hazards than A1. The reservation tables for the functional units (FUs) in these architectures were somewhat arbitrary (due to the lack of commercial processors containing pipelined FUs with hazards), but we have tried to make them as plausible as possible by matching the execution latency and throughput of modern processor architecture.

We have used Huff's bi-directional slack scheduling algorithm [13] as our software pipelining method. In this we have added a simple heuristics to modify the FU configuration (through delay insertion) when a large number of instructions that are executed in that FU got (scheduled and) descheduled in the scheduling process. These FU types are potential candidates for delay insertion and thereby to achieve a higher *Max_Init* and hence schedule the instructions in the loop without causing structural hazards. We have also experimented with a variation of Huff's slack scheduling algorithm, which is only uni-directional, in the sense that it always tries to schedule instructions from the As Soon As Possible (ASAP) time or earliest start (Estart) time [13]. This approach worked better on some loops as opposed to the bi-directional heuristics which tries to schedule instructions either from Estart or from Latest start (Lstart) time. The bi-directional heuristic is especially meant to reduce the register pressure of the software pipelined schedule. Since our experiments did not focus on register pressure, we expect both uni- and bi-directional slack scheduling methods to produce equally good schedules. We refer to the two scheduling methods as Uni-directional Slack Scheduling (USS) and Bi-directional Slack Scheduling (BSS) and report the results from these two methods.

For each architecture we report the performance (in terms of the initiation interval **II** achieved for the loops) with and without modification of the reservation tables. These

results are summarized in Table 4. We refer to the default configuration without modification of reservation table as "*Default Configuration*" and the one where modulo scheduling algorithm chooses the best configuration to suit the loop (or the current **II**) as "*MS Configuration*." As can be expected *MS Configuration* performed significantly better than the *Default Configuration* for both architectures A1 and A2 and for both scheduling methods USS and BSS. To further highlight the usefulness of MS pipeline theory, we also report the results for yet another configuration, termed as "*Best Configuration*" for the two architectures A1 and A2. The *Best Configuration* used some of the reservation tables used by the *MS Configuration*, but these reservation tables were fixed for the given FU type for all the loops. That is, the *Best Configuration* is similar to *Default Configuration* in that they both do not allow modification of the reservation tables, but the former used possibly some of the best reservation tables (with fewer forbidden latencies) used by the *MS Configuration*.

In Table 4, we report the comparison between *Default* and *MS* configurations as well as between *MS* and *Best* configurations. For each comparison we report in how many cases or loops did each configuration achieve a lower **II** compared to the other, and what is the average percentage improvement in **II**. We also report the number of cases in which both configurations achieved the same **II**. In a few loops (2 or 3), some of the configurations (typically *Default* or *Best* configuration) failed to find a schedule in the given amount of trials. We observe that *MS Configuration* performed significantly better than *Default Configuration* in roughly 50% of the loops, and the average percentage improvement in **II** varied from 25% to 104%. This can be expected as the modification to the reservation tables through delay insertion reduces the number of forbidden latencies, and hence improves *Max_Init*. This in turn helps to achieve a lower **II** in a larger number of loops. Note that this improvement in **II** is despite the fact that some of the modified reservation tables in *MS Configuration* had a longer latency due to the delays introduced.

Next we address the question: "So What? If one uses reservation tables with fewer forbidden latencies (than in the *Default* configuration), it will perform better. Why do we need to modify the reservation tables through delays?" However, there is no single resource usage pattern that can give fewer forbidden latencies for all **II** values. That is, the forbidden latency set is sensitive to the values of **II**, and hence unless one chooses the appropriate reservation table, it is not possible to achieve the same performance as in *MS Configuration*. The comparison of *MS Configuration* with *Best Configuration* reveals this fact. *MS Configuration* finds better schedules in roughly 50% of the loops, with an average percentage improvement in varying between 22–60%. In the remaining cases the performance for both configurations are same. In a few cases, 2 or 3 loops, *Best Configuration* performed better than *MS Configuration*. Theoretically this is impossible, as the scheduling method in *MS Configuration* is supposed to have tried the reservation table of *Best Configuration* before going for a higher **II**. however, due to various threshold values used, such as the number of descheduled instructions and the number of (schedule) trials for each **II**, it is possible that the scheduling method would have exhausted its budget of trials for the lower **II** before it reaches the best configuration reservation table. When some of these threshold values are increased to higher values, *MS Configuration* will uniformly perform better than *Default* and *Best Configuration*. This establishes the

Table 4. Performance of *Default*, *MS*, and *Best* Configurations

		MS vs. Default Configuration					MS vs. Best Configuration				
		MS Conf. Better		Default Better		Both Same	MS Conf. Better		Best Better		Both Same
Sched. Method	Architecture	No. of Cases	Avg. Impr.	No. of Cases	Avg. Impr.	No. of Cases	No. of Cases	Avg. Impr.	No. of Cases	Avg. Impr.	No. of Cases
BSS	A1	18	92.5%	0	—	16	14	54.8%	0	—	22
BSS	A2	16	25.9%	0	—	18	14	22.1%	3	21.0%	17
USS	A1	17	104.2%	0	—	15	10	59.9%	1	128.6%	22
USS	A2	17	40.5%	0	—	19	17	22.0%	2	17.0%	17

usefulness of MS-pipeline theory and delay insertion method in the context of software pipelining methods.

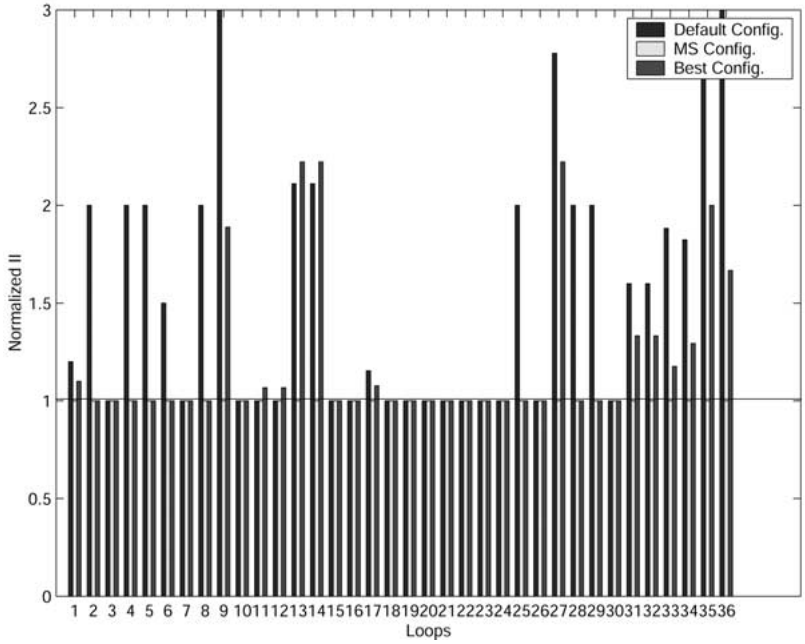
Figures 8 and 9 show bar charts comparing the performances under the *Default*, *MS*, and *Best* configurations for the various loops. The bar charts show the normalized **II** value, normalized with respect to the *MS Configuration*. In a few cases where a particular method did not construct a schedule within the given number of trials, we show the corresponding bar extending to the maximum y-axis value. The above experiments also show that if an architecture has the capability to reconfigure resource usage patterns of a pipeline unit, it can greatly benefit by delay insertion and achieve lower **II** in a larger number of loops. In that sense, this work also has relevance to reconfigurable architecture, although here we do not address how the reconfiguration can be achieved in hardware.

Lastly, we address the question how the MS-pipeline theory is specifically applicable to hardware-software co-design in embedded systems. Let us consider the design of a media processor which is expected to execute a set of the loops, say from a media application. It may be the case that there is no single resource usage pattern in each FU type that "fits" the lowest **II** for each loop. Further let us assume that the embedded processor being designed does not have the ability to introduce delays in the pipeline (i.e., modify the reservation table of the FU) to suit the loop being executed and its **II**. In this case, the designer with the help of the MS-pipeline theory can explore the different reservation tables which will best "fit" the **II**'s of the different loops, and can choose the one that best "fits" many of the loops; that is, choose the reservation table for which many of the loops can be scheduled at a lower **II**. Without the MS-pipeline theory, the designer has to do this in a trial-and-error approach, since classical pipeline theory, as discussed in Section 2.3, which doesn't take into account the **II** values of the different loops, will not help. Whereas MS-pipeline theory can guide the designer to choose the appropriate resource usage pattern.

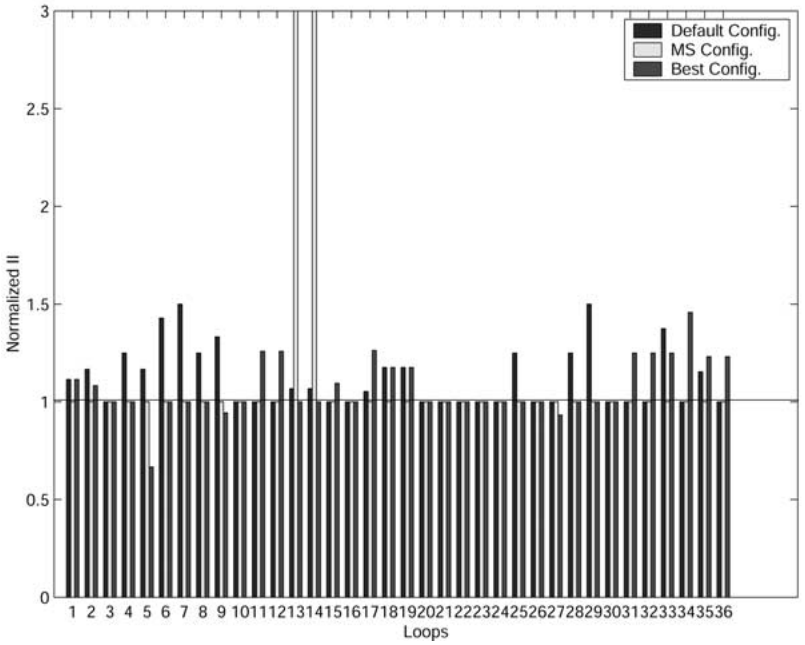
8. Conclusion

In this paper, we have developed the theory of MS-pipelines. In particular we have proposed a new representation for the MS-state diagram which can be analyzed to choose latency sequences that improve the number of initiations in the MS-pipeline. We have established a necessary and sufficient condition, on the permissible latency set, to achieve the upper bound *UB_Init* on the number of initiations in an MS-pipeline. A sufficient condition to achieve a given number of initiations in an MS-pipeline has also been established.

Using the sufficiency condition on the permissible latency set, we have developed a method, taking strong hints from [19], [14], that introduces delays in the MS-pipeline to support a given number of initiations in the pipe. Using the delay insertion method, we establish that the maximum number of initiations can *always* be achieved. Initial experimental evaluation establish the applicability of MS-pipeline theory and the usefulness of the delay insertion procedure in software pipelining methods. This design

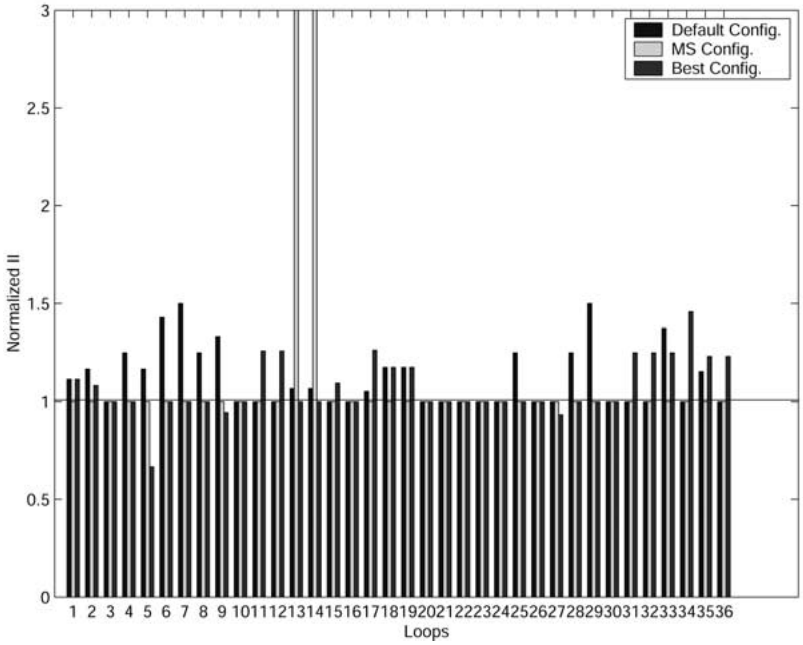


(a) Performance for Architecture A1 and BSS

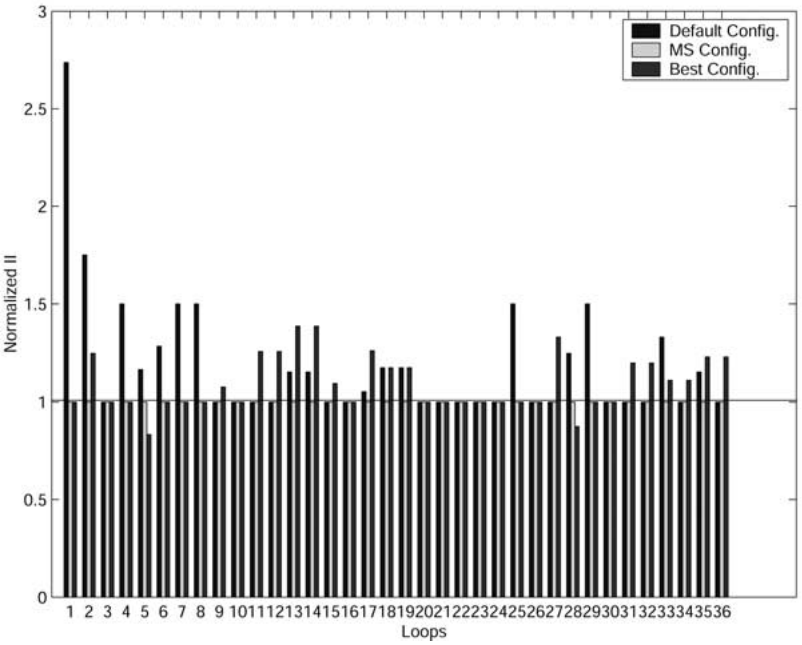


(b) Performance for Architecture A1 and USS

Figure 8. Performance results on software pipelineable loops for architecture A1.



(a) Performance for Architecture A2 and BSS



(b) Performance for Architecture A2 and USS

Figure 9. Performance results on software pipelineable loops for architecture A2.

methodology is extremely useful in designing ASIPs and embedded processors that exploit higher ILP. The proposed theory and the delay insertion methods are also useful in the context of reconfigurable architectures [28], [29] for exploiting greater ILP.

Acknowledgements

The authors are thankful to the anonymous reviewers for their numerous suggestions which helped to improve the presentation of the paper.

Notes

1. Latency 0 is forbidden for single function pipelines—pipelines which support exactly one type of operation.
2. With this folding, multiple × marks separated by **II** may be placed in the same column of the CRT. However, fortunately, the modulo scheduling constraint already prohibits such occurrences. When the modulo scheduling constraint is violated, delays can be introduced to rectify the problem [10]. Thus the cyclic reservation table will not have two × marks on the same column of the CRT.
3. In [10] a similar representation of MS-state diagram involving collision vectors was developed. However, we prefer the new representation proposed in this paper as it is directly useful in establishing some of the properties of the MS-state diagram.
4. An increase in the execution time will also increase the live ranges of variables which in turn may increase the register pressure. We do not consider such effects in this paper.
5. We did not include cases where either $d_{\max}$ is greater than **II** or *UB_Init* equals to 1. While in the former case modulo scheduling constraint was violated, in the latter *UB_Init* was trivially achieved.

References

1. Altman, E. R., 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*, La Jolla, CA, June 18–21, 1995, pp. 139–150.
2. Bala, V. and N. Rubin. Efficient Instruction Scheduling Using Finite State Automata. In *Proc. of the 28th Ann. Intl. Symp. on Microarchitecture*, Ann Arbor, MI, 1995, pp. 46–56.
3. Chaar, J. K. 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. Dehnert, J. C., P. Y.-T. Hsu, and J. P. Bratt. Overlapped Loop Support in the Cydra 5. In *Proc. of the Third Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, April 3–6, 1989, pp. 26–38.
5. Dehnert, J. C. and R. A. Towle. Compiling for Cydra 5, *Journal of Supercomputing*, vol. 7, pp. 181–227, May 1993.
6. Eichenberger, A. E., E. S. Davidson, and S. G. Abraham. Minimum Register Requirements for a Modulo Schedule. In *Proc. of the 27th Ann. Intl. Symp. on Microarchitecture*, San Jose, CA, Nov. 30–Dec. 1994, pp. 75–84.
7. Gasperoni, F. and U. Schwiegelshohn. Efficient Algorithms for Cyclic Scheduling, Res. Rep. RC 17068, IBM T. J. Watson Res. Center, Yorktown Heights, NY, 1991.
8. Govindarajan, R., 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*, San Jose, CA, Nov. 30–Dec. 2, 1994, pp. 85–94.

9. Govindarajan, R., E. R. Altman, and G. R. Gao. A Framework for Resource-Constrained Rate-Optimal Software Pipelining, *IEEE Trans. on Parallel and Distrib. Systems*, vol. 7, no. 11, pp. 1133–1149, Nov. 1996.
10. Govindarajan, R., 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*, San Jose, CA, Feb. 3–7, 1996, pp. 52–61.
11. Govindarajan, R., N. S. S. Narasimha Rao, E. R. Altman, and G. R. Gao. Enhanced Co-Scheduling: A Software Pipelining Method using Modulo-Scheduled Pipeline Theory, *Intl. Journal of Parallel Programming*, vol. 28, no. 1, pp. 1–46, Feb. 2000.
12. Gupta, R. K. and G. De Micheli. Hardware-Software Cosynthesis for Digital Systems, *IEEE Design & Test of Computers*, pp. 29–41, Sept. 1993.
13. Huff, R. A. Lifetime-Sensitive Modulo Scheduling. In *Proc. of the ACM SIGPLAN '93 Conf. on Programming Language Design and Implementation*, Albuquerque, NM, June 23–25, 1993, pp. 258–267.
14. Kogge, P. M. *The Architecture of Pipelined Computers*. McGraw-Hill Book Co., New York, NY, 1981.
15. Lam, M. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proc. of the SIGPLAN '88 Conf. on Programming Language Design and Implementation*, Atlanta, GA, June 22–24, 1988, pp. 318–328.
16. Lee, C., M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. In *Proc. of the 30th Ann. Intl. Symp. on Microarchitecture*, Research Triangle Park, NC, Dec. 1–3, 1997, pp. 330–335.
17. Llosa, J., M. Valero, E. Ayguadé, and A. González. Hypernode Reduction Modulo Scheduling. In *Proc. of the 28th Ann. Intl. Symp. on Microarchitecture*, Ann Arbor, MI, Nov. 29–Dec. 1995, pp. 350–360.
18. Muller, T. Employing Finite State Automata for Resource Scheduling. In *Proc. of the 26th Ann. Intl. Symp. on Microarchitecture*, Austin, TX, Dec. 1–3, 1993.
19. Patel, J. H. and E. S. Davidson. Improving the Throughput of a Pipeline by Insertion of Delays. In *Proc. of the 3rd Ann. Symp. on Computer Architecture*, Clearwater, FL, Jan. 19–21, 1976, pp. 159–164.
20. Philips Semiconductors. TriMedia. http://www.semiconductors.com/trimedia/.
21. Proebsting, T. A. and C. W. Fraser. Detecting Pipeline Structural Hazards Quickly. In *Conf. Rec. of the 21st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, Portland, OR, Jan. 17–21, 1994, pp. 280–286.
22. Rau, B. R. and C. D. Glaeser. Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing. In *Proc. of the 14th Ann. Microprogramming Work.*, Chatham, MA, Oct. 12–15, 1981, pp. 183–198.
23. Rau, B. R. and J. A. Fisher. Instruction-Level Parallel Processing: History, Overview and Perspective, *Journal of Supercomputing*, vol. 7, pp. 9–50, May 1993.
24. Rau, B. R. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *Proc. of the 27th Ann. Intl. Symp. on Microarchitecture*, San Jose, CA, 1994, pp. 63–74.
25. Texas Instruments. TMS 320C6000. http://www.ti.com/sc/docs/products/c6000.
26. Reiter, R. Scheduling Parallel Computations, *Journal of the ACM*, vol. 15, no. 4, pp. 590–599, Oct. 1968.
27. Wang, J., C. Eisenbeis, M. Jourdan, and B. Su. Decomposed Software Pipelining: A New Approach to Exploit Instruction-Level Parallelism for Loop Programs, Res. Rep. No. 1838, Institut Nat. de Recherche en Informatique et en Automatique (INRIA), Rocquencourt, France, Jan. 1993.
28. Waingold, E., M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Barring It to All Software: Raw Machines, *IEEE Computer*, vol. 30, no. 9, pp. 86–93, Sept. 1997.
29. Weinhardt, M. Compilation and Pipeline Synthesis for Reconfigurable Architectures Loops. In *Reconfigurable Architectures—High Performance by Configware (Proc. of the RAW'97)*, April 1997.
30. Zhang, C., R. Govindarajan, S. Ryan, and G. R. Gao. Efficient State-Diagram Construction Methods for Software Pipelining. In *Proc. of the Compiler Construction Conference*, Amsterdam, The Netherlands, March 1999.