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

Efficient State-Diagram Construction Methods for Software Pipelining

Chihong Zhang
R. Govindarajan†
Sean Ryan
Guang R. Gao

CAPSL Technical Memo 28
March 5, 1999

To appear	in the Proceedings of the
	1999 International Conference on
	Compiler Construction (CC'99)
	Amsterdam, The Netherlands, March 20-28, 1999

†Supercomputer Edn. & Res. Centre, Indian Institute of Science,
Bangalore, 560 012, India. Email: govind@serc.iisc.ernet.in

Abstract

State diagram based approach has been proposed as an effective way to model resource constraints in traditional instruction scheduling and software pipelining methods. However, the constructed state diagram for software pipelining method (i) is very large and (ii) contains significant amount of replicated, and hence redundant, information on legal latency sequences. As a result, the construction of state diagrams can take very large computation time. For example, in modeling the resource constraints of the DEC Alpha 21064 processor, it took more than 24 hours on a 250MHz high-performance workstation to construct, say, 100,000 *distinct* latency sequences. In another experiment, out of the 224,400 latency sequences generated, only 30 were distinct. These make state diagram based approach impractical in real compiler implementation.

In this paper, we propose two methods for the efficient construction of state diagrams. In the first method, we relate the construction of state diagram to a well-known problem in graph theory, namely the enumeration of *maximal independent sets* of a graph. This facilitates the use of an existing algorithm as a direct method for constructing distinct latency sequences. The second method is a heuristic approach which exploits the structure of state diagram construction to eliminate redundancy at the earliest opportunity in an aggressive fashion. The heuristic method uses a surprisingly simple check which is formally shown to *completely* eliminate redundancy in the state diagram.

The two methods proposed in this paper greatly reduce the construction time of state diagram, making the state-diagram based approach for software pipelining efficient and practical. We apply the proposed methods to model two real processors, namely DEC Alpha 21064 and the Cydra VLIW processor. Our experiments reveal a significant reduction in the construction time of the state diagram by 3 to 4 orders of magnitude. For example, for the DEC Alpha processor, the proposed methods took roughly 1 second for the generation of first 100,000 distinct latency sequences. Lastly, we report the application of the proposed methods in a software pipelining framework and study the performance.

Contents

List of Figures

List of Tables

1 Introduction

Recent studies on modulo scheduling, an instruction scheduling method for loops [9, 11, 16, 17], in a production compiler has reported significant improvement (upto 35%) in the overall runtime for a suite of SPEC floating point benchmark programs [18]. On the other hand, rapid advances in VLSI technology and computer architecture present an important challenge for compiler designers: a modulo scheduler must be able to handle machine resource constraints much more complex than before and may need to search for an increasingly larger number of schedules before a desirable one is chosen. Therefore, in exploiting the advantage of modulo scheduling in a production compiler, it is important to be able to handle complex resource constraints efficiently and find a good modulo schedule very quickly.

Proebsting and Fraser proposed an interesting approach that uses finite state automata for modeling complex resource constraints in instruction scheduling [15]. Their approach was based on [13] and subsequently improved and applied to production compilers in [2]. This approach was extended to software pipelining methods in [6, 7]. This paper focuses on the efficient construction of finite state automata for software pipelining methods. In the above methods [2, 6, 7, 13, 15], processor resources are modeled using a finite state automata (or state diagram) which is constructed from the resource usage table for each instruction (class). Each path in the state diagram represents a legal latency sequence which could be directly used by the scheduler for modeling resource contention. The construction of the state diagram is typically done off-line and stored in some form so that the instruction scheduler, at the schedule time, only need to read this information. This has effectively reduced the problem of checking structural hazards in the scheduling method to a fast table lookup, resulting in several fold speedup [2, 7, 13]. In particular, the enhanced Co-Scheduling method, a state diagram based modulo scheduling method, reports a 2-fold speedup in the scheduling time (time to construct the software pipelined schedule) [7].

A major challenge facing this Co-scheduling method is the huge size of the state diagram. The size of the state diagram increases exponentially with (i) larger values of initiation interval (of the software-pipelined schedule) and (ii) pipeline function units which share resources (*e.g.,* function units sharing decode stages, and register read-write ports). Further, in the constructed state diagram a significant number of paths are found to contain replicated information regarding legal latency sequences. (Henceforth, we refer to these paths as redundant paths.) The previous work on reduced state diagram [7] has met with only limited success in eliminating this redundancy.

To illustrate the above problem, we present several experimental evidences. In one experiment conducted by us, the state diagram for the DEC Alpha 21064 processor with an initiation interval (**II**) equal to 3, contained 224,400 latency sequences, out of which only 30 were distinct. In another experiment, the state diagram for an **II** $= 16$ contained more than 74 Million

(74,183,493) distinct paths[1]. Lastly, to generate 100,000 distinct latency sequences in the above state diagram, it took more than 24 hours on an UltraSparc machine.

Though, it is theoretically interesting to construct the complete state diagram involving all non-redundant paths, it becomes inefficient for the instruction scheduler/software pipelining method to deal with all these latency sequences. In order to make the state diagram based approach attractive and practical for implementation in real compilers, it is important to provide the scheduler with a large subset of distinct latency sequences/paths in a short computation time.

In this paper, we propose two methods to drastically reduce the construction time of state diagrams. The first of these methods relates the construction to a well-known problem in graph theory, namely the enumeration of *maximal independent sets* of an interference graph. This facilitates the use of an existing enumeration algorithm as a direct fast method for constructing *all* latency sequences. We refer to this method as the Enumeration of Maximal Independent Set (E-MIS) method. Two major advantages of this method are that it is a direct method and it generates only distinct (non-redundant) latency sequences.

The second method uses a heuristic approach to eliminate redundant paths by exploiting the structure of state diagram construction and by employing an aggressive redundance removal approach. This is accomplished by enforcing a surprisingly simple *redundance constraint* which eliminates *completely* all redundant paths. We refer to this method as the Redundancy Prevention (RP) method, as it identifies, at the earliest in the construction process, states which could cause redundancy and prunes them aggressively. We formally establish that the proposed heuristic results in redundance-free state diagram. However, the redundance constraint used by the heuristic is only a necessary (but not sufficient) condition. As a consequence, the aggressive pruning may eliminate some non-redundant paths as well. However, we find that the RP method to work well in practice.

We compare the efficiency of the proposed methods with that of the reduced state-diagram construction (RD) method in modeling two real processors, namely the DEC Alpha 21064 processor and the Cydra VLIW processor [3]. Our experimental results reveal that the proposed methods result in significant reduction in the construction time by about 3 to 4 orders of magnitude. For example, the RP and E-MIS method took only 1.2 and 1.5 seconds respectively, to construct the first 100,000 distinct latency sequences for the DEC Alpha processor for a particular **II**, while the RD method took more than 24 hours. Another interesting observation made from our experiments is that the RP method, though a heuristic approach which can possibly eliminate non-redundant paths, does reasonably well in enumerating *all* non-redundant paths. In fact, for the two processors modeled and for small values of **II** less than 16, it did not miss a single path. Lastly, we use the latency sequences constructed by RP and E-MIS

[1] The number of actual paths (including the redundant ones) is far too many to count and generate!

methods in the Co-Scheduling framework to construct software pipelined schedules. Initial experiments reveal that the proposed methods do seem to perform competitively, and in a reasonable computation time. This provides an empirical evidence for the use of state diagram approach as a practical and efficient method in software pipelining methods.

In Section 2, we present a brief review of the state diagram based software pipelining method. The problem formulation and the proposed approaches are informally discussed in Section 3. Section 4 and 5 respectively present the details of the two proposed methods. In Section 6 we compare the performance of E-MIS and RP methods with the reduced state diagram construction. Section 7 discusses related work and concluding remarks are provided in Section 8.

2 Background and Motivation

Software pipelining has been found to be an efficient compilation technique that results in significant performance improvement at runtime [9, 11, 16, 17]. Modeling of resource constraints (or structural hazards) in software pipelining is becoming increasingly complex in modern processor architectures. However, an efficient resource model is crucial for the success of the scheduling method [2, 4, 6, 15]. In this section, first, we review the state diagram based resource model used in software pipelining [6]. Subsequently we motivate our approaches to the efficient construction of state diagram.

2.1 Background

Conventional approaches to model resource constraints uses a simple reservation table (see Figure 1(a)) [11, 9, 16, 17]. The state diagram approach is a systematic and effective way to represent the resource usages and conflicts when multiple operations of a loop are scheduled [6]. Such an approach is efficient as it uses the underlying notions of forbidden (permissible) latencies in the classical pipeline theory.

A state diagram represents legal latency sequences that do not cause structural hazards in the pipeline. Each state in a state diagram represents the current state of the pipeline (operating under software pipelining with an Initiation Interval (**II**)). Associated with each state is the set of permissible latencies at which new initiations can be made from the current state. An edge from state S to S' in the state diagram represents an initiation with a latency l cycles after S. If the current state S has permissible latencies $\{p_1, p_2, \cdots, p_k\}$, then the new state S' will have permissible latencies $p = ((p_i - l) \texttt{ mod } \mathbf{II})$, for each $i \in [1, k]$, provided p is a permissible latency in the initial state S_0. The state diagram for the above reservation table is shown in Figure 2. For a quick review of related concepts and the construction of state diagrams refer to Appendix A.

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

(a) Reservation Table

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

(b) Resource Usage when $\mathbf{II} = 8$.

Figure 1: A Reservation Table and its Resource Usage for an **II**

A path $S_0 \xrightarrow{p_1} S_1 \xrightarrow{p_2} S_2 \cdots \xrightarrow{p_k} S_k$ in the 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. For example, the path $S_1 \xrightarrow{2} S_2 \xrightarrow{2} S_3 \xrightarrow{2} S_4$ corresponds to initiations at time steps $(0, 2, 4, 6)$. These offset values for the path are collectively referred to as **offset set**, or in short form `OffSet`. Once the state diagram is constructed, then the `OffSets` corresponding to various paths can be used to guide the enhanced Co-Scheduling method to make quick and "good" decision about when (at what offset value) to schedule an instruction in the pipeline [7].

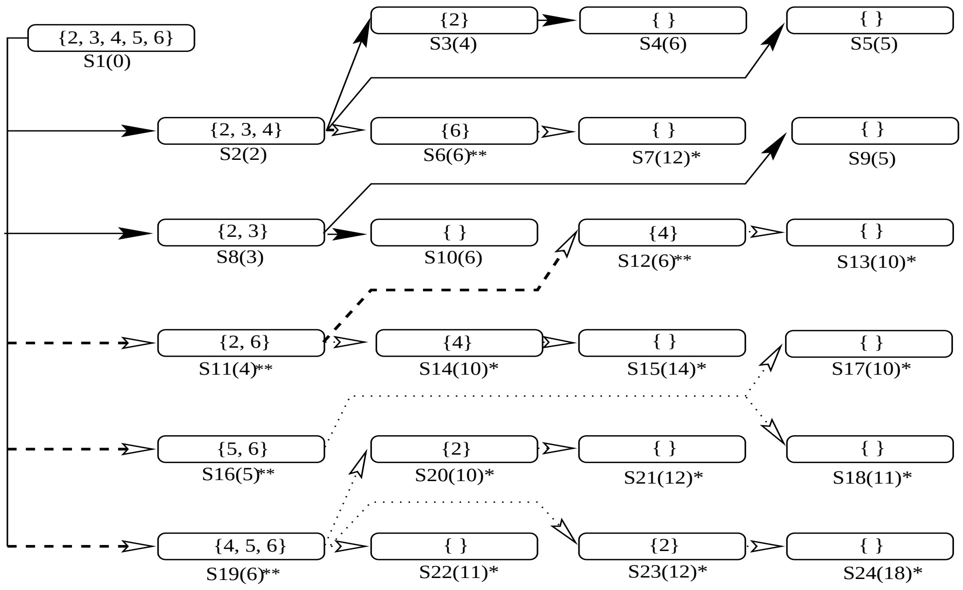

Figure 2: State Diagram for the Example Reservation Table

2.2 State Diagram Explosion and the Redundancy Problem

The major challenge in the construction of a state diagram is that it can become extremely huge, consisting several million states, even for moderate values of **II**. For example, for the DEC Alpha architecture, the state diagram consists of 10, 648, and 224,400 paths when **II** is varied from 1 to 3! Fortunately, several paths in the state diagram are shown to be redundant [7, 8] in the sense that the corresponding `OffSets` are equal. In the above 3 state diagrams for DEC Alpha, only 3, 9, and 30 paths are *distinct*.

For the state diagram shown in Figure 2, the paths $S_1 \xrightarrow{2} S_2 \xrightarrow{2} S_3 \xrightarrow{2} S_4$ and the path $S_1 \xrightarrow{4} S_{11} \xrightarrow{6} S_{14} \xrightarrow{4} S_{15}$ correspond to the same `OffSet` $(0, 2, 4, 6)$, and hence one of them is redundant. The reduced state diagram construction (RD) method proposed in [7] recognizes and eliminates redundant paths by making a simple observation. All states which correspond to initiation times[2] greater than **II** are pruned. Consider the the state S_{14} which corresponds to an initiation at time 10 (shown in brackets adjacent to the state number in Figure 2). In Figure 2, these states (*e.g.,* S_7, S_{13}, and S_{14}) are marked with a '$*$' and the arcs leading to these states are shown as dotted lines.

The following section states this problem of state explosion and informally presents the proposed approaches.

3 Problem Formulation

The RD construction method [7] has the limitation that it is somewhat inefficient in removing the redundancy and also cannot completely eliminate all redundant paths. This is especially so for architectures involving multiple instruction types where different function units may share some of the resources, a common feature in real architectures.

In order for state diagram based approach to be useful in real compilers, it is important to deal with the redundancy-elimination in an efficient way. Further, even after eliminating all redundant paths, the state diagram may still consist of a large number of distinct paths. Hence, for practical reasons, it is important to construct a large subset of distinct `OffSets` in a short computation time so that the scheduler can use this information effectively.

In this paper we propose two efficient solutions for the above problem. In the first solution, the E-MIS method, we relate the generation of `OffSets` to a well-known graph theory problem, *viz.,* enumerating the set of maximal independent sets. This facilitates the use of an existing efficient algorithm for the enumeration to be used as a direct method for generating the `OffSets` without incurring any redundancy.

[2] Initiation time is the sum of the latency values from S_0 to that state along the path. Initiation time modulo **II** corresponds to an offset value.

We motivate the second approach using the state diagram shown in Figure 2. Consider the states S_6 and S_{11} in Figure 2 which lead to only redundant paths. Hence these states should be eliminated at the generation time itself to prevent redundancy. These states (S_6, S_{11}, S_{16} and S_{19}) marked with a double star '$**$' in Figure 2. The RP method identifies these states using a necessary (but not sufficient condition) and prunes them aggressively to prevent redundancy.

Throughout our discussion in this and in the following two sections, for reasons of simplicity, we have considered only state diagrams for function units that do not share resources, However, it is straightforward to extend the ideas of the original RD method, the proposed RP and E-MIS methods for pipelines that share resource. Using such an extension we model the DEC Alpha 21064 processor and the Cydra VLIW processor where resources are shared among function units.

4 A Graph Theoretic Approach for `OffSets` Generation

In this section we relate the generation of `OffSets` to a well-know graph theory problem which facilitates the use of an existing algorithm as a direct and efficient method. This is based on a correspondence between the set of `OffSets` and the maximal compatibility classes which was established in [7].

First, we denote the set of permissible offset values by $\mathcal{O} = \{o_0, o_1, \cdots, o_{n-1}\}$ which includes all initial permissible latencies and 0 [7]. For the example discussed in Section 2.1, the set of permissible offset values is $\mathcal{O} = \{0, 2, 3, 4, 5, 6\}$. The following definitions are useful in the discussion.

Definition 4.1 *A path $S_0 \xrightarrow{p_1} S_1 \xrightarrow{p_2} S_2 \cdots \xrightarrow{p_k} S_f$ in the state diagram is called* **primary** *if the sum of the latency values does not exceed* **II**, *i.e., $p_1 + p_2 + \cdots + p_k < \mathbf{II}$. A path is called* **secondary** *if $p_1 + p_2 + \cdots + p_k > \mathbf{II}$.*

Definition 4.2 *A* **compatibility class** *$\mathcal{C}$ (with respect to $\mathcal{O}$) is a subset of $\mathcal{O}$ such that for any pair of elements c_1 and c_2 in $\mathcal{C}$, $(c_1 - c_2)$ `mod` **II** is in $\mathcal{O}$. A compatibility class is* **maximal** *if it is not a proper subset of any other compatibility class.*

Two compatible classes for $\mathcal{O}$ are $\{0, 2, 4, 6\}$ and $\{0, 2, 5\}$. These compatibility classes are maximal.

Lemma 4.1 *The `OffSet` of any path from the start state S_0 to the final state S_f in the state diagram forms a maximal compatibility class of $\mathcal{O}$.*

Lemma 4.2 *For each maximal compatibility class C of permissible offset values, there exists a* **primary path** *in the state diagrams whose* `OffSet` *O is equal to C.*

Hence obtaining all maximal compatible classes is a direct way of obtaining the `OffSets`. In order to obtain all maximal compatible classes of $\mathcal{O}$, we represent the compatibility information (between pairs of permissible offset values) in the form of an interference graph. The interference graph G for $\mathcal{O}$ has n vertices, each vertex representing one permissible offset value in $\mathcal{O}$. Two vertices v_1 and v_2 in G are connected by an edge, if the corresponding offset values are not compatible. It is possible that a vertex may not share an edge with any other vertex in the graph. The offset value corresponding to such a vertex is compatible with other permissible offset values.

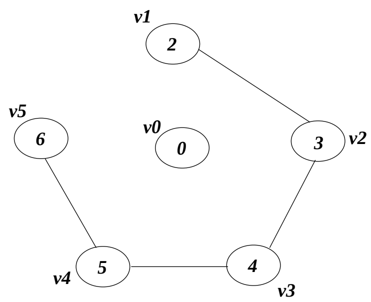

Figure 3: Interference Graph for Permissible Offset Set {0,2,3,4,5,6}

The interference graph for the permissible offset values of our example, $\mathcal{O} = \{0, 2, 3, 4, 5, 6\}$ is shown in Figure 3. The offset value corresponding to each vertex is shown in circle in the figure. There is an edge between vertex v_1 (corresponding to offset 2) and v_2 (corresponding to offset 3) as 2 and 3 are not compatible in $\mathcal{O}$. Next we state the definitions of maximal independent sets from [5].

Definition 4.3 *A subset S of vertices is an* **independent set**, *if there exists no edge between any two vertices in S. An independent set is said to be* **maximal**, *if it is not contained in any other independent set.*

In our example graph, $\{v_0, v_1, v_3, v_5\}$ and $\{v_0, v_2, v_4\}$ are maximal independent sets.

From the above definitions and the description of the interference graph, it clearly follows that each maximal independent set in the graph corresponds to a maximal compatibility class, and hence an `OffSet` in the reduced state diagram. Thus to generate the set of all `OffSets` of the state diagram, one needs to enumerate all maximal independent sets in the interference graph.

An efficient algorithm for this is reported in [19]. This enumeration of MIS is a direct method for the generation of *all* `OffSets`. An attractive features of this approach is that it does not incur any redundancy in the generation of `OffSets`. The complexity of the E-MIS method is $O(\alpha \cdot \beta \cdot \gamma)$ where α, β and γ represent, respectively, the number of vertices, the number of edges, and the number of MIS present in the given graph. The space complexity of the algorithm is $O(\alpha + \beta)$. The number of vertices α in the interference graph is equal to n, the number permissible offset values. Further, since each edge in the graph represents non-compatibility between a pair of offset values, there can be at most $O(n^2)$ edges. Lastly an upper bound for γ can be obtained by further relating maximal compatibility classes to the largest *antichain*[3] of powerset of $\mathcal{O}$ [12]. The cardinality of the largest antichain is given by $\binom{n}{\lfloor n/2 \rfloor}$. Not surprisingly, this bound on γ is exponential in n.

5 Redundancy Prevention Method for State Diagram Construction

In this section, we present the details of the RP method which exploits the the structure of the state diagram construction to identify and eliminate redundancy at the earliest opportunity. Further an attractive feature of the RP method is that it follows an aggressive approach for redundance elimination, even though this may mean missing a few `OffSets`. However our experimental results show that this aggressiveness pays well in state diagrams for real architecture without much loss of information.

5.1 The Redundancy Prevention Algorithm

In this discussion we will assume that the construction of the state diagram proceeds top down, in a depth-first fashion. In our example state diagram (Figure 2), consider the subtrees with their roots at S_6, S_{11}, S_{16}, and S_{19}. All paths in these subtrees are redundant. Thus if our construction method can somehow identify these states and eliminate them before the subtrees rooted on these state are constructed, then the redundant paths through these states can be eliminated.

If we carefully study the example state diagram, we find an interesting characteristic for these four states — the issue time or offset value of a state S appears in the subtree rooted at one of its siblings S'. In particular, the sibling S' is constructed earlier than S in the depth-first construction method. We refer to this sibling S' as a *left sibling* of state S. Notice that there could be more than one left sibling for state S. For convenience, we use the term "*left sibling subtree*" for the tree rooted at a left sibling. For instance, subtrees rooted at S_2 and S_8 are the left sibling subtrees of state S_{11}.

[3]An antichain A is a subset of S such that for any two elements a_1 and a_2 in A, a_1 is not a proper subset of a_2.

If we inspect this property – the offset value of a state S appearing in the left-sibling subtree — a little more carefully it reveals the fact that the partial `OffSet`, consisting of offset values for all states from the start state to S, is a subset of the `OffSet` for another path in the left-sibling subtree. Note that all paths through state S will have this same partial `OffSet`. Further, as the values in an `OffSet` are mutually *compatible*, the partial `OffSet` determines, to a great extent, what other values could be in the `OffSets` for the paths through S. As a consequence, it is very likely that all these paths could be redundant, and hence can be eliminated by not generating the state S and the subtree rooted at S. The RP method aggressively removes these states to prevent redundancy. Note, however, it is only *likely* that all paths through S are redundant. In other words, a state that satisfies the above property may have a path that is not redundant. We defer a discussion on this to Section 5.2. Based on the above observation, we introduce the following constraint that is used in the RP method.

Definition 5.1 (Redundancy Constraint) *A state S is said to satisfy the redundancy constraint (RC) if the issue time of state S is the same as that of a state S' and S' has occurred in one of the left sibling subtrees of S.*

The basic steps in the RP method are:

(1) Follow the depth-first construction rule to construct the state diagram (as in Procedure A.1 in Appendix A).

(2) If a given state S has already created k child states $S_1, S_2, \cdots, S_k$ and is going to create a new child state S_{k+1} (it should be noted here that, at this time, all states in the subtrees rooted at $S_1, S_2, \cdots, S_k$ have been constructed), the redundancy constraint checks whether the issue time (offset value) of S_{k+1} has occurred in subtrees rooted at $S_1, S_2, \cdots, S_k$. If so, the state S_{k+1} will not be created; else it is added to the state diagram.

Step 2 sounds time-consuming because for each new child state to be created we need to check all states in the left sibling subtrees. It should be noticed here that the range of offset values is fixed, *i.e.,* from 0 to $\mathbf{II}-1$. Therefore, it is not difficult to construct an implementation for the RP method the RC check could be done as a simple table lookup. The table, called "`left_sibling_states`" records the issue time of all states in the left sibling subtrees. The size of this table depends on the model, but is very small. Further, for computational efficiency, the RP method constructs the state diagram as a tree rather than as a directed acyclic graph [7]. This means that certain states, *e.g.,* the final state of the state diagram which contains an empty permissible latency set may be repeated several times (states S_5, S_9, and S_{10}) as shown in Figure 2.

The attractive features of this approach are that it exploits the structure of state diagram construction (top-down and left-to-right) to eliminate redundancy at the earliest opportunity.

Second, the RP method is aggressive in redundance elimination. Lastly, as a comparison, the E-MIS method is an exact method, and hence has the overhead of having to construct the complete interference graph, before the generation of `OffSets`.

5.2 Properties of RP Method

Note that the RP method uses the redundancy constraint to eliminate states that could possibly lead to redundant paths. This raises two questions: (i) Does the RP method eliminate **all** redundant paths? (ii) Will the RP method ever eliminate a path that is non-redundant? We answer these two questions in this subsection.

Theorem 5.1 *There are no redundant paths in the state diagram tree constructed by the RP method.*

The proof of this theorem proceeds by showing that, at any point in time, the partial state diagram tree constructed so far does not have any redundancy. The details are given in Appendix B.

We remark that the RP method uses the redundancy constraint which is only a necessary, but not sufficient condition. As a result it may miss some non-redundant paths due to the aggressive pruning employed by this method. However, we argue that there is still a good chance that of the "missed" `OffSets` will reappear in some other paths of the state diagram. In Section 6 we study empirically how many non-redundant paths does RP miss in the state diagram and whether they have any influence on the constructed software pipelined schedule.

6 Experimental Results

In this section we report our experience in generating the `OffSets` using the proposed methods for various values of **II** for two real processors, namely the DEC Alpha 21064 superscalar processor and the Cydra VLIW processor. Both architectures have complex resource usage pattern with sharing of resources. The RP and the E-MIS methods have been implemented in our Modulo Scheduling Testbed (MOST) framework [1].

6.1 Construction Time Comparisons

In order to compare the construction speed of RP, E-MIS and RD methods in a fair manner, all the three methods were run to generate a large subset of `OffSets`, consisting of the first, say, 100,000 *distinct* `OffSets`. Tables 4 (in Appendix C) compares the construction time for

OffSets generation for the three methods on an Ultra-450 Workstation (with a 250 MHz clock). We observe that the RD method is much slower than RP and E-MIS. For example, the RD method failed to generate 100,000 distinct OffSets even after running for 24 hours for the Alpha architecture, for an **II** $= 8$. Hence for larger values of **II**, we only compared the RP and the E-MIS methods.

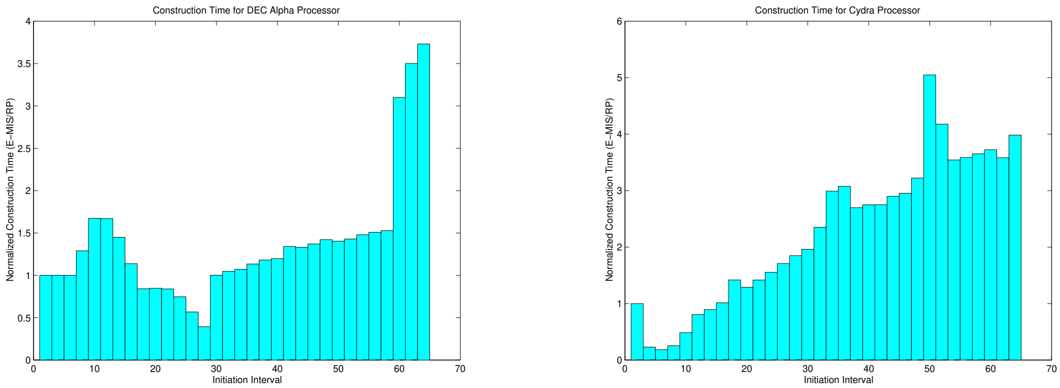

Figure 4: Comparison of Construction Time for RP and E-MIS Methods

Figure 4 shows the normalized (w.r.t. the RP method) construction time taken by the E-MIS methods for various values of **II** for the two architectures. We observe that the RP and the E-MIS methods are competitive: E-MIS performs better for small values of **II** while RP performs better for moderate to large **II**s. This is not surprising as RP is a very efficient heuristics that we employ to get the OffSets quickly while the E-MIS method, being inherently an exact method, is slow, especially for large values **II**. This is due to the fact that the E-MIS method needs to construct the entire (large) interference graph before the generation of OffSets.

6.2 How Many Paths Does RP Miss?

As mentioned earlier the RP method can miss some non-redundant paths during the construction. It is important to verify that RP will not miss too much useful information. On the other hand, since E-MIS is capable of generating all distinct OffSets, we can compare the number of OffSets generated by these two methods if these methods were allowed to complete their execution without any restriction on the maximum number of OffSets generated. The number of distinct OffSets in the state diagram is significantly large, even for small values of **II**. Further the construction time grows exponentially with larger values of **II**. This has limited our experiments to values of **II** less than 16 for the Alpha processor and less than 12 for the Cydra processor. The results are tabulated in Table 1. From the experimental results, we observe that the RP method generates **all** non-redundant paths in the considered cases. It didn't miss a single path! This is somewhat surprising to us, as we anticipated that the RP heuristic to eliminate some non-redundant paths as well. However, we believe that the RP method may still miss some non-redundant paths for large values of **II** and/or for other architectures.

II	DEC Alpha Processor			Cydra Processor		
	No. of Paths Generated		No. of Missed Paths	No. of Paths Generated		No. of Missed Paths
	RP	E-MIS		RP	E-MIS	
1	3	3	0	4	4	0
2	9	9	0	16	16	0
3	30	30	0	64	64	0
4	93	93	0	256	256	0
5	288	288	0	1,024	1,024	0
6	894	894	0	4,096	4,096	0
7	2,775	2,775	0	131,072	131,072	0
8	8,613	8,613	0	589,824	589,824	0
9	26,733	26,733	0	7,340,032	7,340,032	0
10	82,974	82,974	0	32,505,856	32,505,856	0
11	257,535	257,535	0	142,606,336	142,606,336	0
12	799,338	799,338	0	N/A	N/A	N/A
13	2,480,988	2,480,988	0	N/A	N/A	N/A
14	7,700,499	7,700,499	0	N/A	N/A	N/A
15	23,900,835	23,900,835	0	N/A	N/A	N/A
16	74,183,493	74,183,493	0	N/A	N/A	N/A

Table 1: Missed Paths in the RP Method

6.3 Application to Co-Scheduling

Lastly we attempt to answer the question how critical are the missed `OffSets`, if any, in terms of the quality of the constructed software pipelined schedule, when the the enhanced Co-Scheduling method uses the `OffSets` generated by the RP method. Since the RP method did not miss any path for the DEC Alpha and Cydra processor (for the values of **II** that we could experiment), its application in the enhanced Co-Scheduling should perform at least as good as any other `OffSet` generation method, *e.g.,* E-MIS or RD methods, applied to Co-Scheduling. What happens if the RP method does miss some `OffSets`? To answer to this question, we considered the reservation tables used in [7] which model function units with complex resource usage, but without any sharing of resources. In these reservation tables the RP method misses a few paths even for small values of **II**. For example, for one of the reservation tables, the RP method generated only 22 out of 26 distinct `OffSets`. Also, to reduce the complexity of the enhanced Co-Scheduling method, the scheduling method used all distinct `OffSets`, up to a maximum of 1000, generated either by RP or E-MIS method. The enhanced Co-Scheduling was applied to a set of 737 loops consisting of single basic block extracted from scientific benchmark programs. We compare the constructed software pipelined schedules, in terms of the initiation interval and the construction time for schedule in Table 2. In order to have a fair comparison, we did not include the time for the generation of `OffSets` in the scheduling time. We observe that both methods perform more or less alike. This once again establishes that the `OffSets` missed by the RP method are not really critical in terms of the quality of the constructed schedule.

Measure	RP-Based Co-Scheduling Method Better		E-MIS-Based Co-Scheduling Method Better		Both Same
	# Cases	% Improvement	# Cases	% Improvement	# Cases
Initiation Interval	18	1.11%	16	0.16%	703
Scheduling Time	289	94.18%	448	94.4%	0

Table 2: Effectiveness of RP and E-MIS in Software Pipelining

6.4 Summary of Results

To summarize our experimental results:

- The RD method is too slow in the construction of state diagram for architectures in which function units share resources. Hence it is impractical to apply this for modeling real processors.
- Both the RP and E-MIS methods are much faster than RD, by 3 to 4 orders of magnitude, and their construction time is acceptable for implementation in real compilers.
- In terms of construction speed, the E-MIS method performs better for small values of **II** while RP method performs better for large **II** values.
- Though the RP method can not guarantee to always generate *all* distinct `OffSets`, we found that it is capable of generating all non-redundant `OffSets` in the experiments that we conducted. Further, we found that the generated `OffSets` when applied to the Co-Scheduling method can get equally good performance, in terms of the constructed schedule.

7 Related Work

Finite state automaton for modeling resource usage was proposed in [2, 13, 15] for instruction scheduling methods. These methods use ideas from the classical pipeline theory, especially the notion of forbidden and permissible latency sequences. The size of the constructed automaton was found to be large in [13] which was subsequently improved by Proebsting [15] and

Bala [2]. The Co-Scheduling framework proposed in [6] and subsequently extended in [7] is a state diagram-based software pipelining method. The size of the constructed state diagram is an even more serious problem in software pipelining than in instruction scheduling.

In this paper, we have proposed two methods to efficiently generate the distinct `OffSets` of a state diagram. Both methods avoid redundancy in the construction and hence are found to be very efficient. Compared to the reduced state diagram construction method, the proposed RP and E-MIS methods result in a significant reduction in the construction time, by 3 to 4 orders of magnitude. Lastly, an alternative method to model resource usage was proposed by Eichenberger and Davidson [4]. This method relies on the use of global resource table, but reduces the cost of structural-hazard checking by reducing the machine description. Their approach uses forbidden latency information to obtain a minimal representation for individual reservation tables of different function units.

8 Conclusions

In this paper we have proposed two efficient methods to construct distinct paths in the state diagram used for modeling complex pipeline resource usage in software pipelining. The methods proposed in this paper, namely the E-MIS method and the RP method completely eliminate the generation of redundant paths. The first of these methods, the E-MIS method is obtained by relating the `OffSets` generation to a well-known graph theoretic problem, *viz.,* the enumeration of maximal independent sets of an interference graph. The second method, the RP method, uses a simple but very effective heuristic to prevent the construction of states that may cause redundant paths. We formally establish that this heuristic results in redundance-free state diagram. We compare the performance of RP and E-MIS methods in modeling two real processors, namely the DEC Alpha processor and the Cydra VLIW processor. The RP and E-MIS methods were found to be much superior than the RD method, by 3 to 4 orders of magnitude. When compared between themselves, the RP and the MIS methods perform competitively, the RP performing better for larger values of **II** while the E-MIS performing better for small **II**. Lastly, we have applied the `OffSets` generated by these methods in the enhanced Co-Scheduling method and reported their performance.

References

[1] E. R. Altman. *Optimal Software Pipelining with Function Unit and Register Constraints.* Ph.D. thesis, McGill U., Montréal, Qué., Oct. 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, MI, Nov. 29–Dec.1, 1995.

[3] G. Beck, D.W.L. Yen, and T. L. Anderson. The Cydra-5 minisupercomputer: Architecture and implementation. *Journal of Supercomputing*, 7, May 1993.

[4] 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*, pages 12–22, Philadelphia, PA, May 21–24, 1996.

[5] M.C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, New York, 1980.

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

[7] R. Govindarajan, N.S.S. Narasimha Rao, E. R. Altman, and G. R. Gao. An enhanced co-scheduling method using reduced ms-state diagrams. In *Proc. of the 12th Intl. Parallel Processing Symp.*, Orlando, FL, Mar. 1998. IEEE Computer Society. Merged with 9th Intl. Symp. on Parallel and Distributed Processing.

[8] R. Govindarajan, N.S.S. Narasimha Rao, Erik R. Altman, and Guang R. Gao. An enhanced co-scheduling method using reduced ms-state diagrams. CAPSL Technical Memo 17, Dept. of Electrical & Computer Engineering, University of Delaware, Newark 19716, U.S.A., Feb. 1998. Also as Tech. Report TR-98-06, Dept. of Computer Science & Automation, Indian Institute of Science, Bangaloe, 560 012, India. (available via `http://www.csa.iisc.ernet.in/~govind/papers/TR-98-2.ps.gz`).

[9] 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.

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

[11] 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.

[12] C.L. Liu. *Introduction to Combinatorial Mathematics.* McGraw-Hill Book Co., New York, NY, 1968.

[13] Thomas Muller. Employing finite state automata for resource scheduling. In *Proc. of the 26th Ann. Intl. Symp. on Microarchitecture*, Austin, TX, Dec. 1–3, 1993.

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

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

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

[17] 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.

[18] John Ruttenberg, G. R. Gao, A. Stouchinin, and W. Lichtenstein. Software pipelining showdown: Optimal vs. heuristic methods in a production compiler. In *Proc. of the ACM SIGPLAN '96 Conf. on Programming Language Design and Implementation*, pages 1–11, Philadelphia, PA, May 21–24, 1996.

[19] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM Jl. on Computing*, 6(3):505–517, Sep. 1977.

A Review of State Diagram Approach

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, henceforth referred to as MS-pipeline, 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 [4].

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 1(b) are $\{0, 1, 7\}$ and $\{2, 3, 4, 5, 6\}$ respectively.

To analyze the MS-pipeline and to identify valid latency sequences that do not cause a collision, we construct the state diagram [6]. The initial state in the state diagram represents an initiation at time 0. Each state in the state diagram is represented by the set of permissible

[4]Modulo scheduling constraint prevents two `X` marks separated by **II** cycles to be mapped to the same column in the CRT.

latencies in that state. Thus, the initial state consists of the initial permissible latencies. The construction of the state diagram proceeds as follows.

Procedure A.1: Construction of 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 **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'.

B Proof of Theorem 5.1

Theorem 5.1. *There are no redundant paths in the state diagram tree constructed by the RP method.*

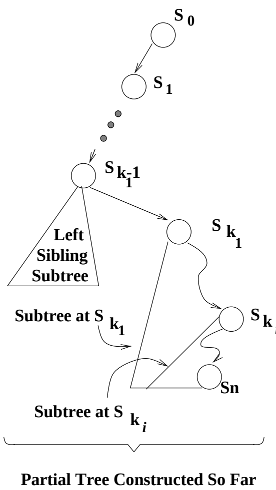

Figure 5: Partial State Diagram Tree

Proof Sketch: At any point in time, we consider the partial state diagram tree constructed so far and prove that there are no redundant paths in it. Assume that the next path generated by the RP method in the partial state diagram is P. Let

$$P = S_0 \xrightarrow{l_1} S_1 \xrightarrow{l_2} S_2 \cdots \xrightarrow{l_n} S_n$$

be the path with the corresponding `OffSet` $\{o_0, o_1, \cdots, o_n\}$. Now we will show that P is a non-redundant path. The proof is by contradiction. Suppose P is redundant. In the path P, let S_{k_1} be a state such that all states $S_1, S_2, \cdots, S_{k_1-1}$ are left-most children of their parents (respectively, $S_0, S_1, \cdots, S_{k_1-2}$) and S_{k_1} is not a left-most child. Note that k_1 can be such that $1 \leq k_1 \leq n$ (refer to Figure 5). Since P is redundant, then either (A) the offset value o_{k_1} corresponding to the state S_{k_1} must appear in the left-sibling tree(s) of S_{k_1} or (B) the offset values $o_{k_1+1}, \cdots, o_n$ must appear in the subtree rooted at S_{k_1}. Clearly (A) cannot be true since if o_{k_1} appears in the left-sibling subtree, then it implies that S_k satisfies RC, and hence would have been eliminated by the RP method. Suppose (B) is true, *i.e.*, $o_{k_1+1}, \cdots, o_n$ appears in the subtree rooted at S_{k_1}. In the path

$$P_1 = S_{k_1} \xrightarrow{l_k} S_{k_1+1} \xrightarrow{l_n} S_n$$

we can find a k_2 such that the states $S_{k_1+1}, \cdots, S_{k_2-1}$ are left-most children and S_{k_2} is not. Applying the previous argument and recursively reducing the path, we arrive at a subtree rooted k_i where all subsequent states $(S_{k_i+1}, \cdots, S_n)$ are left-most children. At this stage, argument (B) — the offset values $o_{k_i+1}, \cdots, o_n$ must be redundant in the subtree rooted at S_{k_i} — is also not true, since this is the only path (left-most path) constructed so far in this subtree. This contradicts the fact that P is redundant. Hence the theorem.

C Details of Experimental Results

In this section we report the detailed comparison of the execution time of RP and E-MIS methods for constructing the state diagram, up to a maximum of 100,000 distinct `OffSets`.

II	Construction Time in Seconds			Normalized time w.r.t. RP method	
	E-MIS	RP	RD	(E-MIS/RP)	(RD/RP)
2	0.01	0.01	0.01	1	1
4	0.01	0.01	0.67	1	67
6	0.12	0.12	151.48	1	1,262
8	1.47	1.14	> 24hours	1.28	>75,789
10	19.52	11.67	N/A	1.67	N/A
12	26.89	16.1	N/A	1.67	N/A
14	25.58	17.65	N/A	1.45	N/A
16	22.24	19.54	N/A	1.14	N/A
18	19.78	23.50	N/A	0.84	N/A
20	25.05	29.52	N/A	0.85	N/A
22	26.58	31.64	N/A	0.84	N/A
24	25.61	34.29	N/A	0.75	N/A
26	20.85	36.72	N/A	0.57	N/A
28	15.69	39.84	N/A	0.39	N/A
30	44.23	44.16	N/A	1	N/A
32	49.9	47.64	N/A	1.05	N/A
34	56.26	52.52	N/A	1.07	N/A
36	62.99	55.53	N/A	1.13	N/A
38	70.04	59.27	N/A	1.18	N/A
40	77.44	64.59	N/A	1.2	N/A
42	90.27	67.3	N/A	1.34	N/A
44	94.91	71.35	N/A	1.33	N/A
46	104.24	76.04	N/A	1.37	N/A
48	114.6	80.59	N/A	1.42	N/A
50	121.36	86.45	N/A	1.51	N/A
52	131.89	92.23	N/A	1.43	N/A
54	142.47	96.35	N/A	1.48	N/A
56	153.87	102.05	N/A	1.51	N/A
58	165.15	107.98	N/A	1.53	N/A
60	402.61	129.88	N/A	3.1	N/A
62	482.34	137.78	N/A	3.5	N/A
64	544.01	145.8	N/A	3.73	N/A

DEC Alpha Processor

Table 3: Comparison of Construction Time for the 3 Methods for Alpha Architecture

II	Construction Time in Seconds			Normalized time w.r.t. RP method	
	E-MIS	RP	RD	(E-MIS/RP)	(RD/RP)
2	0.01	0.01	4296.69	1	429,700
4	0.03	0.13	> 24hours	1	> 664,615
6	0.39	2.13	N/A	0.17	N/A
8	15.37	60.4	N/A	0.25	N/A
10	41.03	84.4	N/A	0.48	N/A
12	77.2	95.7	N/A	0.81	N/A
14	95.99	107.35	N/A	0.89	N/A
16	121.18	119.43	N/A	1.01	N/A
18	172.99	121.97	N/A	1.41	N/A
20	180.62	140.05	N/A	1.29	N/A
22	219.41	154.82	N/A	1.42	N/A
24	258.97	166.71	N/A	1.55	N/A
26	302.13	176.67	N/A	1.71	N/A
28	350.29	189.48	N/A	1.84	N/A
30	399.34	203.57	N/A	1.96	N/A
32	511.08	217.32	N/A	2.35	N/A
34	666.32	222.65	N/A	2.99	N/A
36	759.61	246.98	N/A	3.08	N/A
38	713.47	264.40	N/A	2.7	N/A
40	767.30	279.28	N/A	2.74	N/A
42	810.42	294.69	N/A	2.75	N/A
44	893.61	308.25	N/A	2.9	N/A
46	962.55	325.87	N/A	2.95	N/A
48	1106.17	343.08	N/A	3.22	N/A
50	1735.5	343.8	N/A	5.04	N/A
52	1579.46	378.06	N/A	4.18	N/A
54	1391.99	392.93	N/A	3.54	N/A
56	1478.93	412.2	N/A	3.59	N/A
58	1573.37	431.0	N/A	3.65	N/A
60	1675.52	449.97	N/A	3.72	N/A
62	1773.45	495.05	N/A	3.58	N/A
64	1970.47	494.75	N/A	3.98	N/A

Cydra Processor

Table 4: Comparison of Construction Time for the 3 Methods