

Control Flow Regeneration for Software Pipelined Loops with Conditions

Dragan Milicev and Zoran Jovanovic, University of Belgrade

Abstract: We propose a new intermediate representation for software pipelined loops with conditions. The representation allows separation of operations from different paths and their conditional, as well as speculative scheduling, including speculative IFs. We also define an algorithm that transforms the representation into the executable code. The algorithm uses the notion of finite automata to represent the execution of separate paths as threads of control that are canceled or approved by executed IF operations. The approach may be used in conjunction with modulo scheduling or other techniques to reconstruct the control flow graph from the final (modulo) schedule directly. It inherently solves the problems of overlapped predicate lifetimes and speculation. The approach provides also a novel formal model for loop execution.

Keywords: instruction level parallelism, software pipelining, loops with conditions, code generation, control flow, finite automata, predicated software pipelining, reverse if-conversion

1 Introduction

Software pipelining is an approach that produces parallel code for loops by issuing a new iteration before the preceding one has been completed. Loops with conditional branches (IF structures) are more difficult to schedule than those without, because the outcomes of branches cannot be predicted at compilation time. There is a lot of techniques that define various heuristics to deal with the complexity of the problem. With respect of how they represent the loop code, perform scheduling upon that representation, and produce the executable code, these techniques can be categorized as single-phased or three-phased. Single-phased techniques use the control flow graph as the loop body representation and move operations across the edges of the graph. Since this representation corresponds directly to the executable program form, these techniques do not need a special code generation phase that would transform an intermediate representation into the executable code. However, these techniques suffer from some weaknesses that will be discussed later. On the other side, three-phased techniques use other intermediate representations that are more appropriate for scheduling. Therefore, they can produce more efficient code in presence of resource constraints. Consequently, they need a special pre- and post-scheduling phases to transform the initial control flow graph into the intermediate representation, and to regenerate the control flow graph in order to produce the executable code from the intermediate

representation. The techniques for control flow regeneration published so far pose some unnecessary boundaries to the scheduling phase, thus reducing its efficiency. Namely, they cannot deal with some scheduling issues (such as speculative execution, especially of IF operations) that can enlarge the amount of parallelism and that single-phased techniques exploit. This paper proposes an intermediate representation and a code generation algorithm that can be used in three-phased techniques to eliminate the recognized weaknesses of the existing approaches.

The paper is organized as follows. In Section 2, we discuss further our motivation for the research and the related work. In Section 3, we state the problem precisely. Section 4 presents the proposed solution in an intuitive manner, following a simple demonstrative example. Several interesting special cases that the proposed algorithm can cope with, but other approaches cannot, are shown in Section 5. Some implementational implications and preliminary experimental results are discussed in Section 6. The paper ends with conclusions.

2 Motivation and Overview of the Related Work

Single-phased techniques [4, 9, 10] use the control flow graph to represent the loop. Loop transformations are defined as operation moves across the edges of the graph along with the topological transformations of the graph due to the moves of IFs. The advantage of these techniques is that they keep the control flow graph up to date after each transformation, so they do not need a special post-scheduling code generation phase. These techniques allow speculative code motion—even speculative IFs are allowed. However, this representation is the main constraining factor of the techniques. It does not explicitly support data dependency and resource conflict analyses, which should be the main driving forces of the scheduling. This drawback has been recognized for long, and other representations have been proposed, based on the program dependence graph [5, 19]. They tend to represent data flow dependencies more directly, while implicitly encoding control flow in a manner that allows its regeneration [19].

Warter et al. [19] provide another point of view to the benefits of using other intermediate representations than the control flow graph: global scheduling techniques, which are also widely used for loops, consist of two phases—inter-block code motion and local (basic block) scheduling;

the engineering problem of global scheduling is to determine how to properly order these two phases to generate the best schedule. Therefore, an intermediate representation that could implicitly encode control flow would simplify the task of global scheduling to one that looks like local scheduling, by eliminating the need for an explicit code motion phase during scheduling.

A number of three-phased techniques are based on *modulo scheduling* [11, 15, 16, 17, 18, 20]. Modulo scheduling uses a special scheduling table called *modulo reservation table* with II rows, where II (*initiation interval*) is the interval at which iterations are started. II is determined according to the data dependencies and resource usage of operations. Modulo scheduling places operations into the reservation table according to the data dependencies and resource constraints. It has been proved experimentally that modulo scheduling achieves good performance in presence of resource constraints [11, 14, 15], particularly when resource usage patterns are complex.

In order to benefit from these properties, the techniques for scheduling loops with conditions usually transform control dependencies into data dependencies by *if-conversion* [3, 19]. In this approach, a predicate is assigned to each operation; the operation is to be executed if and only if its predicate is computed to be true. Conditional operations (IFs) are considered to compute predicates. Thus, control dependent operations become data dependent on the conditional operations that compute their predicates. Predicated execution may be supported by hardware [12]. Otherwise, *reverse if-conversion* [19] or *kernel-recognition* [2, 15] strategies are used to regenerate the control flow graph from the modulo schedule. However, the promotion of control dependencies into data dependencies introduces constraints that do not exist in the original code, thus blocking speculative code motion. This can limit parallelism considerably. Furthermore, both approaches need explicit unrolling in order to resolve overlapped predicate lifetimes.

Recently, a technique called *split-path enhanced pipeline scheduling* (SP-EPS) [14] has been proposed. It tries to benefit from both approaches. It moves the operations across the edges of the control flow graph and transforms it iteratively, but it also uses modulo schedule of separate iteration paths to drive the moves. However, we still need an algorithm that would be able to transform modulo scheduled code directly into the control flow graph in order to decrease the

overhead of updating the control flow graph at each operation move. Such an algorithm may allow better understanding of properties of loops with conditions and a deeper study of the effects of various scheduling aspects to the generated code size and efficiency. In particular, our future work will use the representation described here to explain how SP-EPS achieves the final schedule and how it can be improved. The weakness of SP-EPS is that it uses modulo scheduling directly for intra-path kernels only, but indirectly for the operations from the transition paths. We will try to show how our representation can be used to modulo schedule all the operations, i.e., to mix intra- and inter-path operations into the same modulo schedule, and to produce the code directly from it. This might lead to a better resource usage of inter-path operations and to a smaller II .

3 Problem Statement

As a result, an intermediate representation of loops with conditions and an algorithm that will reconstruct the executable code in the form of the control flow graph from this representation are needed. The representation and the algorithm should be able to:

- (R1) implicitly encode control flow to support modulo scheduling and other approaches that are primarily based on data dependency and resource constraint analyses;
- (R2) support software pipelining of operations, i.e., placement of operations into other iterations than the initial ones; this placement may be conditional (to be explained later);
- (R3) support data dependency and resource conflict analyses during scheduling;
- (R4) enable speculative code motion, including speculative IFs;
- (R5) allow schedules with variable initiation interval (II).

In order to make further explanations more clear, we will use a simple example throughout the paper. We underline that the accent here is not on the scheduling and parallelization, but on the code generation phase. That is why we assume that the schedule has been already obtained without considering how. The example has been chosen to be simple and descriptive for the given purposes, and not aimed to show the benefits of software pipelining. It will not consider data dependencies and resource constraints, either. Moreover, the discussion on the preloop and postloop code generation, as well as the loop exit branches will be given later in the paper.

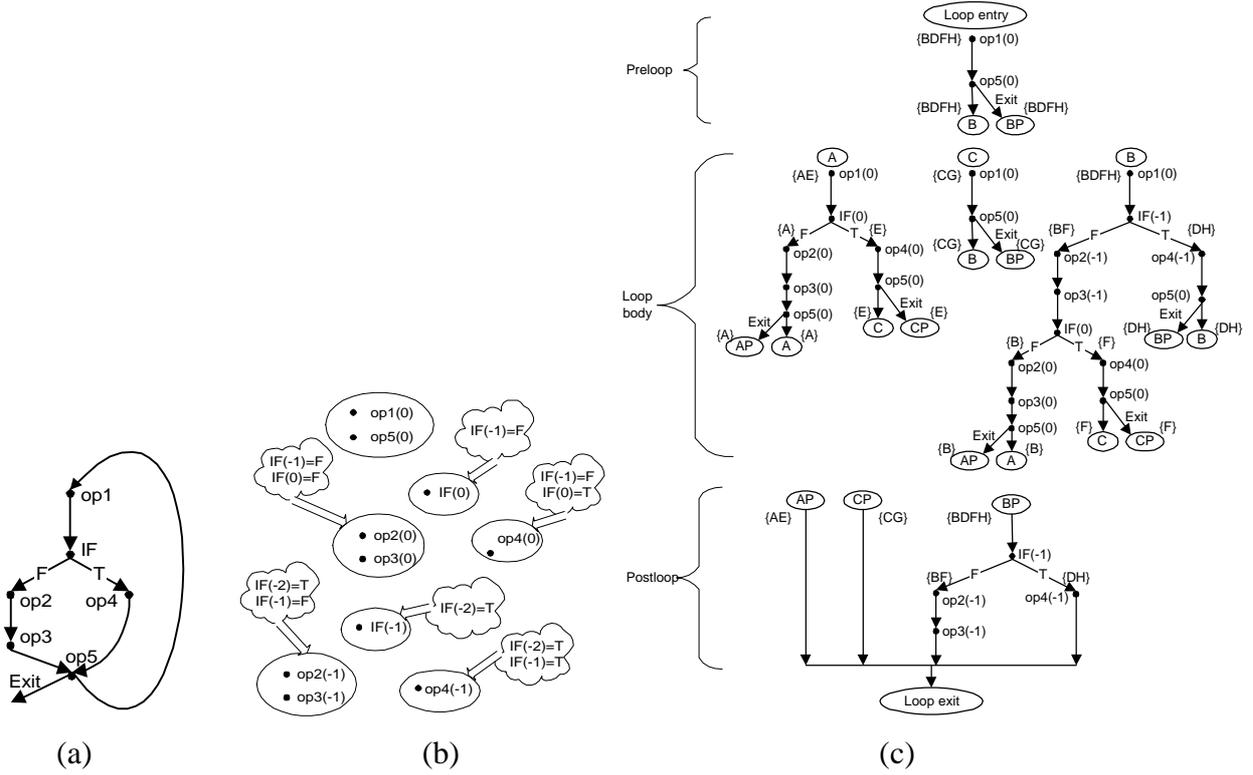


Figure 1: Sample loop. (a) The control flow graph of the initial loop. (b) The assumed final schedule, informally represented (the input for the code generation algorithm). (c) The generated control flow graph (the output from the code generation algorithm).

The control flow graph of the initial sample loop is shown in Figure 1a. The assumed final schedule for a considered current iteration is informally presented in Figure 1b. This schedule is the input for the code generation algorithm. Indices in parentheses represent the original iterations of the operations: 0 denotes the current, -1 the previous iteration. We assume that the IF operation, along with the control dependent operations $op2$, $op3$, and $op4$, has been moved from the current iteration into the next iteration conditionally: they are to be executed in the current iteration only if the outcome of the IF from the previous iteration were False; otherwise, they are to be executed in the next iteration. This conditional movement of operations has been stated in the requirement R2. As a result, an iteration of the initial loop will be executed in one or two stages, depending on the outcome of the IF from the previous iteration. The final executable loop code that should be obtained by the code generation algorithm is shown in Figure 1c. The loop consists of three transformed kernels A, B, and C, connected by the loop-back edges.

4 The Proposed Solution

The proposed solution is based on an intermediate representation that uses *predicate matrices* instead of single predicates. The concept of the predicate matrix has been proposed in [6] to introduce an execution model of software pipelined loops with conditions, and has been used later in a scheduling technique called *predicated software pipelining* (PSP) [7]. To make this paper self-contained, we will briefly present the PSP intermediate representation (PSP IR). When a software pipelined loop is represented with the PSP IR, its execution can be modeled by transitions of a nondeterministic finite automaton (NFA) [1]. The transformation of the NFA into the equivalent deterministic one (DFA) is actually the control flow regeneration algorithm. The proposed approach does not assume any special hardware facilities. A formal proof of correctness of the algorithm may be found in [8].

The PSP Intermediate Representation

PSP defines a *predicate instance* as a virtual Boolean variable that represents possible outcomes of one IF operation in one iteration. The model separates the notions of a predicate and an IF operation, as if-conversion does: a predicate controls scheduling, while an IF operation computes its value. However, unlike other techniques that consider only paths of a single iteration, PSP uses the term *path* for an execution trace of the whole loop. Because the number of iterations that a loop may execute is taken to be finite but unlimited [13], PSP uses a notion of a *predicate matrix* as a limited mathematical concept that can represent an unlimited *set of paths*.

Each IF construct and its IF operation in the compiled code is assigned a row in the predicate matrix. PSP observes a loop execution relative to the current (referential) iteration. The current iteration is denoted with 0, the next one with 1, the previous one with -1, etc. A set of paths is represented by a predicate matrix with m rows, where m is the number of IFs in the initial loop, and n columns, where n is an arbitrary number that limits the scope of predicates. The columns are indexed, with the column i referring to the i -th iteration relative to the current. The elements of the column 0 refer to the current iteration and will be underlined in our notation. The element in row i and column j represents the instance of the predicate (IF operation) i in the j -th

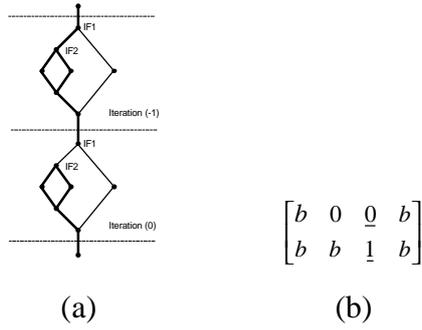


Figure 2: Formal representation of a set of paths that span over unlimited number of iterations by a predicate matrix. (a) A control flow graph of two adjacent iterations and an unlimited set of paths that pass through the bolded branches. The left branch of each IF is the False (0) branch. (b) The predicate matrix that represents the set.

iteration relative to the current. An element may be 0 (for False), 1 (for True), and b (for both outcomes of the predicate instance). A value $x \in \{0, 1\}$ of an element (i, j) means that the matrix represents the set that includes only those paths that pass through the outcome x of the predicate instance (i, j) (see Figure 2 for an example).

Since a predicate matrix is of a limited width, but should represent an unlimited set of paths, it is assumed that all other columns of its virtual unlimited extension are filled with bs . Therefore, the limited width of a matrix bounds only the scope of elements that are not equal to b . This kind of limitation is quite natural, because each (finite) software pipelined schedule must be limited in the level of pipelining, which corresponds to the scope of predicate instances.

Because a predicate matrix represents a set of paths, all usual set operations and relationships may be defined for predicate matrices. For example, the subset relationship may be defined as follows: $pm1 \subseteq pm2$ iff for each i, j : $pm1(i, j) \subseteq pm2(i, j)$; the "inclusion" relationship for elements is defined as: $0 \subseteq b$, $1 \subseteq b$, $v \subseteq v$, $v \in \{0, 1, b\}$. Similarly, the intersection operation of two predicate matrices $pm1$ and $pm2$ may result in an empty set (\emptyset) if $pm1$ and $pm2$ have opposite elements at one position (opposite are 0 and 1), or a predicate matrix $pm = pm1 \cap pm2$ where the element (i, j) of pm is that one of $pm1(i, j)$ and $pm2(i, j)$ which is included into the other. However, some path sets may not be represented by a single predicate matrix; for example, the

```

op1(0) [b]
IF(0)  [b]
op2(0) [0]; op4(0) [1]
op3(0) [0]
op5(0) [b]

```

Figure 3: The initial PSP IR of the sample loop. Each operation is assigned a predicate matrix with a single column 0, and the index 0 (in parentheses).

result of the union of two predicate matrices $[1 \ \underline{b}] \cup [0 \ \underline{1}]$. This is not a problem in our approach because neither initial representation of the loop body code, nor any of the scheduling transformations may produce a path set that cannot be represented by a single matrix. Predicate matrix operations and relationships thus constitute a set calculus over the power set of the entire (unlimited) set of all possible paths, referred to as the B -set (represented by a matrix with all bs).

A single-iteration path of the initial loop body code can be represented with a predicate matrix with a single zero-column. Nested IFs are treated as follows. If an IF-THEN-ELSE construct IF2 is nested inside e.g. True path of the IF1 construct, it is also assigned a row of the predicate matrix. Then, all the paths that pass through the False branch of IF1 have b elements in the row of IF2. Consequently, each operation of the initial loop body can be assigned an initial predicate matrix with a single zero-column that represents the paths on which the operation is executed in the initial loop. Loop exit branches are not considered; they are called BREAK operations and will be discussed later. The initial PSP IR for our sample loop is in Figure 3.

In order to distinguish different instances of a same operation that originate from different iterations of the initial loop, an integer index is also assigned to each operation. The index is initially equal to 0. When an operation is moved (scheduled) into another iteration, its index is changed. Index i of an operation instance denotes that this instance belongs to the iteration i (relative to the current) of the initial loop. An operation instance is thus defined as a triple $\langle \text{operation, index, predicate matrix} \rangle$. The PSP IR consists of operation instances scheduled according to dependency rules and available resources. The initial representation is obtained as described and transformed by the scheduler.

Scheduling Issues

This subsection lists the transformations that should be applied on operation instances during scheduling in order to keep the PSP IR consistent. The *split* transformation makes two operation instances out from one. The resulting instances have the same operation and index, but their predicate matrices have the opposite values at one position that was equal to b in the initial instance's matrix (Figure 4a). Its purpose is to differentiate two instances of the same operation

<pre> op1(0)[<u>b</u>] IF (0)[0<u>b</u>]; IF (0)[1<u>b</u>] op2(0)[0<u>0</u>]; op2(0)[1<u>0</u>] op4(0)[0<u>1</u>]; op4(0)[1<u>1</u>] op3(0)[0<u>0</u>]; op3(0)[1<u>0</u>] op5(0)[<u>b</u>] </pre> <p style="text-align: center;">(a)</p>	<pre> op1(0)[<u>b</u>] IF (-1)[1<u>b</u>]; IF (0)[0<u>b</u>] op2(-1)[1<u>0</u>]; op2(0)[0<u>0</u>] op4(-1)[1<u>1</u>]; op4(0)[0<u>1</u>] op3(-1)[1<u>0</u>]; op3(0)[0<u>0</u>] op5(0)[<u>b</u>] </pre> <p style="text-align: center;">(b)</p>
---	---

Figure 4: Transformations applied to the initial schedule of the sample loop. (a) The schedule after splitting operations *IF*, *op2*, *op3*, and *op4* at (1,-1). (b) The schedule after moving down the bolded operations from (a) into the next iteration. The operation ordering is irrelevant.

and to allow their separate scheduling depending on the outcome of an IF, i.e., the conditional placement of operations (the requirement R2 of the problem statement). This may be useful if an operation is constrained by different data dependencies across two branches of an IF. For example, APP [15] and SP-EPS [14] schedule separate paths in separate modulo tables, which is equivalent to splitting operation instances on certain predicate instances. The *unify* transformation is dual and may be used to merge two unnecessarily split operation instances in the final schedule [7].

Software pipelining is supported by the *movedown* and *moveup* transformations. The *movedown* transformation moves an operation instance into the next iteration. In order to preserve the relative reference to the original iteration and to the predicate instances that control the moved operation instance, its index is decremented then, and its predicate matrix is shifted one place left (Figure 4b). The *moveup* transformation is completely dual.

The final schedule for the sample loop after the *movedown* transformations of the chosen operation instances is shown in Figure 4b, as the PSP IR of the schedule in Figure 1b. The ordering of operation instances is controlled by the dependency and resource conflict analyses of the scheduler and is not relevant to the code generation algorithm.

Data and control-dependency analyses take into consideration the paths on which two operation instances should be executed. Two operation instances may be dependent only if there is a path on which both instances are to be executed, i.e., only if the intersection of their predicate matrices is not empty. An IF operation instance with row i and index j is said to compute the predicate instance (i, j) . An operation instance opi is control dependent on an IF operation instance $opif$ iff opi and $opif$ do not have disjoint matrices and $opif$ computes the predicate (i, j)

Iteration:	... i	$i+1$	$i+2$	$i+3$	$i+4$...
IF1 outcome:	... 0	0	1	0	1 ...
IF2 outcome:	... 1	0	1	1	0 ...
Sequence:	... $\begin{bmatrix} ? & \underline{0} & 0 \\ b & \underline{1} & b \end{bmatrix}$	$\begin{bmatrix} 0 & \underline{0} & 1 \\ b & \underline{0} & b \end{bmatrix}$	$\begin{bmatrix} 0 & \underline{1} & 0 \\ b & \underline{1} & b \end{bmatrix}$	$\begin{bmatrix} 1 & \underline{0} & 1 \\ b & \underline{1} & b \end{bmatrix}$	$\begin{bmatrix} 0 & \underline{1} & ? \\ b & \underline{0} & b \end{bmatrix}$...

Figure 5: A sample execution trace (path) of a loop with two IFs. The sequence of outcomes may be represented by a sequence of predicate matrices of a certain format.

which has a non- b value in the opi 's matrix. If an operation instance opi is control dependent on an IF instance $opif$ and is scheduled before it, it is speculative.

Other issues, such as resource conflict analysis and renaming due to non-true data dependence elimination and speculation, also need special consideration in the context of the PSP IR, but are out of scope of this paper, because they do not have any impact on the code generation algorithm. Besides, their treatment might heavily depend on the very scheduling technique. For example, SP-EPS [14] avoids resource conflicts between operations from different paths (modulo tables) by executing IFs early enough (path splitting in SP-EPS).

It is important to underline again that the final schedule in the PSP IR need not be obtained by iteratively applying the described transformations as in [7]. It might be obtained by modulo scheduling when different modulo schedules are produced for different outcomes (paths) [14, 15]. It is important only that the operation instances from different paths are distinguished by corresponding predicate matrices, and that the needed predicate matrix shift and index change is applied if an operation is scheduled in an iteration other than the original.

Loop Execution Model and the NFA

In our preliminary observations we will ignore the startup and shutdown phases of the loop execution. We will consider the steady state of the loop execution only, which is of major interest. Let us consider one execution trace (a path) of a loop, defined by a (theoretically) infinite sequence of outcomes. For a loop with two IFs, one path is shown in Figure 5.

The path may be described with a sequence of predicate matrices of a certain format, as in Figure 5. By a "predicate matrix format" we mean a set of matrices with non- b elements at certain positions. For the previous example, one matrix format might be (x stands for a non- b element):

State	Outc.[00]	Outc.[10]	Outc.[01]	Outc.[11]	Scheduled IFs	Predicate vector [IF(-1);IF(0)]
A[000]	AE	AE	∅	∅	IF(0)	[b0]
B[100]	AE	∅	∅	∅	IF(-1), IF(0)	[00]
C[010]	BF	BF	BF	BF		[bb]
D[110]	∅	BF	∅	BF	IF(-1)	[1b]
E[001]	∅	∅	CG	CG	IF(0)	[b1]
F[101]	∅	∅	CG	∅	IF(-1), IF(0)	[01]
G[011]	DH	DH	DH	DH		[bb]
H[111]	∅	DH	∅	DH	IF(-1)	[1b]

Figure 6: The NFA for the sample loop and the given final schedule. The rows of the table represent the states of the NFA, and the columns represent the IF outcomes as input symbols. The states are identified by letters A..H. ∅ stands for the error state.

$\begin{bmatrix} x & \underline{x} & x \\ b & \underline{x} & b \end{bmatrix}$. It is assumed that the matrix format has no b elements between non- b elements in any

row. Such b elements are called " b -holes." Note the relation between two adjacent matrices in the sequence: the successor is obtained by shifting the predecessor one position left, discarding its left-edge elements, and filling its right-edge elements by the upcoming outcomes. An edge element is one that has a b element by one of its sides. We will refer to this rule as the "sequencing rule."

Execution of a loop may be regarded as a (theoretically infinite) transition sequence of a nondeterministic finite automaton (NFA), whose states correspond to the iterations of the sequence. The NFA is constructed as follows. First, the predicate matrix format is defined, which:

- 1) has a non- b element at each position at which there is a non- b element in a predicate matrix of any operation instance in the final schedule;
- 2) has a non- b element at the position (i, j) for each IF operation instance in the final schedule that computes the predicate instance (i, j) ;
- 3) does not have b -holes. The explanations of these requirements will be given soon.

For the sample loop and its final schedule in Figure 4, the predicate matrix format is $[x \ x \ \underline{x}]$. The states of the NFA are defined by predicate matrices that have all variations of the non- b elements of the constructed matrix format. The NFA for the example is given in Figure 6.

The meaning of the NFA being in a state is that the actual execution of the loop is (supposed to be) following one of the paths included in the state's matrix. The corresponding operation instances of the final schedule are to be executed in that state; these are the instances

whose predicate matrices are supersets of the state's matrix. Since only IF operations are significant to the control flow and execution of the NFA, the IF instances that are to be executed in each state are shown in Figure 6. The outcomes of IF instances are "input symbols" of the NFA, because they define its transitions as follows. If the executed IF operation instance produces an outcome that contradicts the predicate instance value defined by the assumed state's matrix, the assumption of being in this state is found to be incorrect. The successor for this outcome is the error state (\emptyset), meaning that the set of the supposed paths determined by the state's matrix is to be canceled. Otherwise, if no contradiction is found, the successors are determined by the sequencing rule: these are the states with the matrices obtained by shifting the state's matrix one place left and varying the right-edge elements. The successor set of states has 2^m elements. At this moment, nondeterminism occurs, because it is not known into which state the execution should pass.

Consider the state F defined by $[1 \ 0 \ \underline{1}]$ in Figure 6. Both instances IF(-1) and IF(0) from the final schedule (Figure 4) are to be executed in this state because their predicate matrices are supersets of the F's matrix. If IF(-1) computes 1 (columns $[1 \ \underline{0}]$ and $[1 \ \underline{1}]$), which contradicts to the supposed 0 of the state F $[1 \ 0 \ \underline{1}]$ at (1,-1), the assumption is found incorrect, and the NFA should pass to \emptyset . Similarly, if IF(0) computes 0 (columns $[0 \ \underline{0}]$ and $[1 \ \underline{0}]$), which contradicts to the supposed 1 of the state F $[1 \ 0 \ \underline{1}]$, the NFA should pass to \emptyset . For the remaining outcome $[0 \ \underline{1}]$ (IF(-1)=0, IF(0)=1), no contradiction is found, and the NFA should pass nondeterministically to one of the states C and G, because they might be F's successors in a legal execution trace.

The reasons for the stated requirements for matrix format construction are clear now. The first requirement enables distinguishing all necessary states according to the predicate matrices of operations, which assures their conditional execution. The second one enables finding contradictions or accordance for all scheduled IF instances in each state and defining the transitions of the NFA. The third one enables determining the legal successors of states without loss of history.

	[00]	[10]	[01]	[11]
\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
AE [00b]	AE	AE	CG	CG
BF [10b]	AE	\emptyset	CG	\emptyset
CG [01b]	BDFH	BDFH	BDFH	BDFH
DH [11b]	\emptyset	BDFH	\emptyset	BDFH
BDFH[1bb]	AE	BDFH	CG	BDFH

Figure 7: Construction of the DFA for the NFA in Figure 6. The DFA states are sets of NFA states. The predicate matrices assigned to the DFA states are unions of the corresponding NFA states. The states are listed in order of appearance. The states of the tailing SCC are shaded.

Control Flow Regeneration

The process of control flow regeneration consists of two steps. First, the DFA equivalent to the defined NFA is constructed. The DFA defines the set of kernels for the transformed loop, one for each DFA state, and the deterministic transitions between these kernels, which will be the loop back edges. Second, the acyclic control flow graph is constructed for each kernel.

The construction of an equivalent DFA for the given NFA is well known [1], except that the automata here do not have either "starting" or "ending" states. A state of the DFA is a set of NFA states, i.e., an element of the power set of the NFA states. The set of input symbols of the DFA is the same as of the NFA. The initial set of states of the DFA consists of the NFA state sets that exist as destinations in the NFA transition table; this is the set: $\{S=\{S_i \mid i=1,\dots,k, S_i \text{ is an NFA state} \mid S \text{ is a successor of an NFA state}\}$. Then, an iterative procedure is applied: for each unprocessed DFA state $T=\{S_i \mid i=1,\dots,k, S_i \text{ is an NFA state}\}$, transitions are defined as:

$$TransDFA(T,I) = \bigcup_{S \in T} TransNFA(S,I).$$

If there is a transition $TransDFA(T,I)=Q$ such that Q does not exist yet in the DFA, a new state Q is added. The procedure ends when there are no new states in the DFA. It ultimately ends because the set of all possible DFA states is finite. The DFA for our example is given in Figure 7.

The obtained DFA is a directed cyclic graph that might consist of several strongly connected components (SCCs) that may be topologically sorted. Because an execution of a loop is treated to be an infinite sequence of DFA transitions, and any sequence of IF outcomes may occur, the DFA might eventually end in one of the SCCs that have no successors (other than \emptyset) in the topological order. These SCCs will be called the tailing SCCs. Therefore, all the SCCs other than

one of the tailing may be ignored. Consequently, the final DFA consists of a single tailing SCC (plus possibly \emptyset). The final DFA for the example consists of the states AE, CG, and BDFH, corresponding to the kernels A, C, and B, respectively in the final control flow graph in Figure 1c.

The last step is the construction of the acyclic control flow graph for each kernel. The procedure is quite similar to the reverse if-conversion, but modified for the PSP IR. The algorithm maintains the leaf node (basic block) set, which initially contains the starting node only. Each basic block is assigned a set of "active" NFA states $NFAS$ that corresponds to the allowable predicate set in the reverse if-conversion. The starting block is assigned an $NFAS$ that is equal to the DFA state of the kernel being constructed. The algorithm processes operation instances from the schedule, one at a time. An operation instance with predicate matrix pm_{opi} is added to those leaf basic blocks which have an active NFA state $S \in NFAS$ in which the operation should be executed ($pm_S \subseteq pm_{opi}$). When an IF operation instance is encountered, two new basic blocks are added as successors to all the blocks the operation has been placed in. The leaf node set is updated accordingly. Each of the two new basic blocks for the two outcomes has the active NFA state set updated according to the same rule as in the procedure of the NFA construction: if this IF operation instance computes the predicate instance (i, j) and is to be executed in an NFA state $S \in NFAS$, but the predicate matrix element $pm_S(i, j)$ has the opposite value of the outcome for the branch, this state S is deleted from the new basic block's $NFAS$. The transition edge (loopback edge) is constructed for each final leaf basic block according to the sequencing rule for the states of its $NFAS$. The intermediate representation does not support predicate merge operations yet, which is a weakness of the approach. The active $NFAS$ sets are also shown in Figure 1c.

Preloop and Postloop

A real, finite loop execution path may be viewed as a subpath of an infinite path, whereby the iterations before and after the considered finite path are virtual. The virtual iterations execute no operations, but produce some assumed (arbitrary) outcomes of IFs. For example, consider a path of the sample loop in Figure 8a, with the sequences of NFA and DFA transitions in Figure 8b.

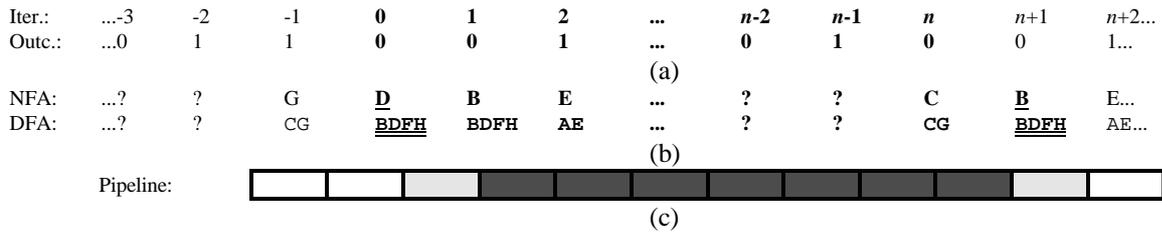


Figure 8: Extending a real execution of a loop to a virtual infinite execution. Iteration 0 is the first, and n is the last actual iteration. (a) A supposed path for the original sample loop. The bolded outcomes represent the actually executed outcomes as a subpath of an infinite path. Other outcomes belong to the virtual iterations with no executed operations, and are arbitrarily assumed. (b) The sequences of the NFA and DFA state transitions that correspond to the supposed path. Bolded are those states that should be executed to ensure that all the operations from the supposed original path are executed. Underlined are the states that constitute the preloop and postloop. (c) The model of the preloop, body, and postloop execution is the model of filling, executing, and draining a pipeline, where the pipeline stages interchange conditionally.

When a current iteration $-i$ ($i > 0$, absolute numbering) is virtual, preceding the actual subpath, it should execute only those operations that have indices greater than or equal to i . These might be the operations that have been moved from the actual iterations. Therefore, the schedules for the NFA states for this iteration are partial, including only those operations. The excluded IF instances are virtual and are assumed to produce arbitrary outcomes. The next iteration $-(i-1)$ should include only those operations with indices greater than or equal to $i-1$, etc. Ultimately, the iteration schedule completes, and the loop reaches the steady state. This is the principle of preloop construction. It starts from an "early enough" absolute iteration $-i$ so it does not skip any actual operation. Partial schedules are constructed for the preloop iteration by iteration, until a full kernel schedule is reached. For the given example, this occurs for the iteration 1, because the first actual iteration 0 should not execute the operations with indices -1 . Since an arbitrary virtual subpath may be assumed to precede the actual subpath, it may be one that leads the DFA into one of its tailing SCCs. The DFA then stays in it forever. That is why only one tailing SCC is important.

In our model, we distinguish conditional loop exit operations, denoted with BREAK, and IF operations considered so far. Executing a BREAK means a signal to exit the loop, while IF operations direct the control flow inside the loop. Their difference is only conceptual, but their implementation is the same. The postloop construction follows the similar idea as for the preloop.

State	[0b0]	[1b0]	[0b1]	[1b1]	IFs
A[0000]	AI	AI	∅	∅	IF(+1)
B[1000]	AI	∅	∅	∅	IF(-1), IF(+1)
C[0100]	BJ	BJ	∅	∅	IF(+1)
D[1100]	∅	BJ	∅	∅	IF(-1), IF(+1)
E[0010]	CK	CK	CK	CK	
F[1010]	CK	∅	CK	∅	IF(-1)
G[0110]	DL	DL	DL	DL	
H[1110]	∅	DL	∅	DL	IF(-1)
I[0001]	∅	∅	EM	EM	IF(+1)
J[1001]	∅	∅	EM	∅	IF(-1), IF(+1)
K[0101]	∅	∅	FN	FN	IF(+1)
L[1101]	∅	∅	∅	FN	IF(-1), IF(+1)
M[0011]	GP	GP	GP	GP	
N[1011]	GP	∅	GP	∅	IF(-1)
P[0111]	HQ	HQ	HQ	HQ	
Q[1111]	∅	HQ	∅	HQ	IF(-1)

(a)

State	[0b0]	[1b0]	[0b1]	[1b1]
AI[000]	AI	AI	EM	EM
EM[001]	CGKP	CGKP	CGKP	CGKP
CGKP[01b]	BDHJLQ	BDHJLQ	DFHLNQ	DFHLNQ
BDHJLQ[1b0] ∪ [111]	AI	BDHJLQ	EM	DFHLNQ
DFHLNQ[1b1] ∪ [110]	CGKP	BDHJLQ	CGKP	DFHLNQ

(b)

Figure 9: Non-speculative split, speculative move of an IF. The final schedule is: IF(+1)[0]; IF(-1)[1bb]. (a) The NFA. (b) The final DFA.

The postloop is constructed for each BREAK instance, i.e., for its loop-exit branch. The active NFA state sets are constructed for each basic block following this branch as previously, but the pipeline is being drained for those paths that execute the BREAK, by constructing partial schedules. The draining is performed by considering indices of the operations and the index of the BREAK, which defines the index of the broken actual iteration, relative to the current iteration.

We have only shown the principles, while the details and algorithms can be found in [8]. For our example, assuming that *op5* is a conditional BREAK, and that the state BDFH is chosen as the loop entry state, the final control flow graph of the transformed loop is given in Figure 1c.

5 Case Study

To illustrate the general applicability of the approach, we show some interesting cases that existing techniques cannot deal with. The example we have studied so far is simple, where the IF operation instance (which is of major interest because it affects the control flow) was split non-speculatively (it was split on the predicate instance from the previous iteration), and moved also non-speculatively (it was moved down). The result was a simple final DFA without the error state.

The example in Figure 9 shows a non-speculative split, but a speculative move of an IF. The initial instance IF(0)[*b*] was split on (1,-1), producing two instances: IF(0)[0*b*] and IF(0)[1*b*]. Then, the former was moved up, and the latter was moved down one iteration, producing the

State	[00]	[10]	[01]	[11]	IFs
A[000]	AE	AE	∅	∅	IF(0)
B[100]	AE	AE	∅	∅	IF(0)
C[010]	∅	∅	BF	∅	IF(-1), IF(0)
D[110]	∅	∅	∅	BF	IF(-1), IF(0)
E[001]	CG	CG	CG	CG	
F[101]	CG	CG	CG	CG	
G[011]	DH	∅	DH	∅	IF(-1)
H[111]	∅	DH	∅	DH	IF(-1)

(a)

State	[00]	[10]	[01]	[11]
CG[01]	DH	∅	BDFH	∅
DH[11]	∅	DH	∅	BDFH
ACEG[0b]	ACDEGH	ACEG	BCDFGH	CG
BDFH[1b]	ACEG	ACDEGH	CG	BCDFGH
ACDEGH[b1] ∪ [00]	ACDEGH	ACDEGH	BCDFGH	BCDFGH
BCDFGH[b1] ∪ [10]	ACDEGH	ACDEGH	BCDFGH	BCDFGH

(b)

Figure 10: Speculative split on a future IF. The final schedule is: IF(0)[b0]; IF(-1)[1]. (a) The NFA. (b) The states of SCCs of the DFA; those of the tailing SCC are shaded.

schedule IF(+1)[0]; IF(-1)[1b]. The schedule involves a speculative execution of an IF under certain condition, which is rather difficult to explain and comprehend intuitively.

Even more peculiar are the following two examples. The example in Figure 10 shows a speculative split on a future predicate instance. IF(0)[b] was split on (1,1), producing IF(0)[b0] and IF(0)[b1], and the latter was moved down, producing the schedule: IF(0)[b0]; IF(-1)[1]. The DFA has two SCCs. The states of the tailing SCC are shaded in Figure 10, while the states of the other have transitions to the error state.

Finally, the example in Figure 11 shows an IF speculatively split on itself. First, IF(0)[b] was split to IF(0)[0] and IF(0)[1], and the former was moved up speculatively, producing IF(0)[1]; IF(+1)[b0]. The final DFA has two states, one of them leading to the error state.

Not only are these examples hard to comprehend, but they can unlikely occur in a real scheduling technique. However, they prove the general applicability of the approach, which might be regarded as a complete formal model of loop execution, whereby the execution of separate paths may be viewed as separate threads of control that are canceled or approved by executed IFs. The difference of the paths is limited to the scope of predicate instances described by predicate matrices. At the same time, the execution is encoded by a static, conventional code in the form of

State	[00]	[10]	[01]	[11]	IFs
A[00]	AC	AC	∅	∅	IF(+1)
B[10]	∅	AC	∅	∅	IF(0), IF(+1)
C[01]	BD	BD	BD	BD	
D[11]	∅	BD	∅	BD	IF(0)

(a)

State	[00]	[10]	[01]	[11]
BD[1]	∅	ABCD	∅	BD
ABCD[b]	ABCD	ABCD	BD	BD

(b)

Figure 11: Speculative split of an IF on itself. The final schedule is: IF(0)[1]; IF(+1)[b0]. (a) The NFA. (b) The final DFA.

the control flow graph. Moreover, the model might improve our understandings of the phenomenon. For example, an interesting issue is the reachability of the error state in the final DFA. We speculate that it is reachable only if there is an IF that was split speculatively. This property might be understood intuitively, but needs a formal proof.

6 Implementational Implications

The code generation algorithm has actually two tasks. First, it has to find out which kernels (DFA states) comprise the transformed loop body and which paths (NFA states) start each kernel. Second, it has to reconstruct the acyclic control flow graph for each kernel. The latter is very similar to the reverse if-conversion. Its complexity much depends on the schedule and is difficult to describe. The former is the main contribution of the approach and its core step, because it must balance the two needs: to distinguish paths by separate kernels if possible (if IFs are computed early enough), and to merge paths that cannot be distinguished due to speculation. The separation of paths leads to the kernels and transitions that "remember the history" of execution. For example, if the schedule for the current iteration depends on an outcome from k iterations before, an order of 2^k kernels are needed to "remember the history" of outcomes. This issue is inherent in our approach, because NFA states are generated for all variations of predicate instances between columns $-k$ (on which an operation is control dependent) and 0 (which is computed by the IF). Basically, our algorithm inherently resolves the overlapped predicate lifetimes, but only for those predicates for which this is necessary (a predicate matrix may have different non- b elements in different rows), without explicit need for unrolling, which would expand all predicates equally.

There are some other properties of the algorithm that may simplify its implementation. First, the NFA definition does not need the full transition table with 2^q columns, where q is the number of IF instances in the final schedule, because a successor of an NFA state N may only be either \emptyset or the set of NFA states S defined by the sequencing rule. A compact representation of the set of "input symbols" (variations of outcomes of all IF instances) that lead to the non- \emptyset successor S is a *predicate vector*. Each predicate vector has one element from $\{0, 1, b\}$ for each

IF instance in the final schedule. A primitive predicate vector is one that has no b elements; it represents one "input symbol" of the automata. Each NFA state N is thus defined only by its predicate matrix, the set S , and a predicate vector. Its predicate vector is defined as follows. If an IF instance $opif$ is to be executed in N ($pm_N \subseteq pm_{opif}$) and its outcome contradicts to the N 's matrix element computed by $opif$, N will not contribute to the set of successor states of a leaf basic block in a kernel (DFA state), because it will be canceled. Therefore, only if the outcome of the IF is $x(\neq b)$ that coincides with this N 's matrix element, N will contribute to the successor state set with S . That is why x stands in N 's predicate vector element for $opif$. If $opif$ instance is not executed in N , b stands in N 's vector element for $opif$, meaning that N contributes regardless to the $opif$'s outcome. For the sample loop, predicate vectors are given in the rightmost column in Figure 6.

Furthermore, instead of constructing the full DFA transition table, the algorithm may only record the possible successors of each DFA state T . Instead of varying all elements of the predicate vector, which would mean filling all columns of the transition table for T , the algorithm varies only the elements of the vector at places where there is at least one non- b element in the vectors of the NFA states that constitute T . This property may significantly reduce the searching space. Consequently, the algorithm constructs a new successor set Q of NFA states for each such vector v : for each NFA state $N \in T$ with vector u , if $v \subseteq u$, then N 's successors S are added to Q . The meaning is that for outcomes defined by v , N contributes to the T 's successor Q only if $v \subseteq u$.

Figure 12 gives the complexity estimation of the NFA and DFA construction procedures. Although it is very difficult to give the precise total complexity because some important parameters depend heavily on the final schedule, it is obvious that the NFA construction is an exponential algorithm with the complexity $O(2^N)$, where N is the number of non- b elements in the predicate matrix format. The exponentiality is an inherent property of the problem due to the described effect that non- b elements in a matrix row mean a need to remember the history of outcomes. Moreover, it may be assumed that the DFA construction algorithm is predominantly influenced by the factor D (the number of DFA states), because D is a factor that might be expected to be large and out of the direct control of the scheduler. It is also very difficult to

Symbol	Meaning
$nops$	Number of operation instances in the final schedule
$nifs$	Number of IF operation instances in the final schedule
m	Number of IF operations in the initial loop; number of rows in the predicate matrix
n	Number of columns in the predicate matrix
N	Number of non- b elements of the NFA predicate matrix format
D	Number of DFA states
M	Mean number of NFA states in a DFA state
nv	Mean number of non- b elements in a predicate vector format of a DFA state

(a)

Proc.	Step	Complexity
NFA constr.	Create predicate matrix format	$O(nops \cdot m \cdot n)$
	Create NFA states by the matrix format; For each NFA state:	$O(2^N)$
	Create the state's predicate matrix	$O(m \cdot n)$
	Create the state's successor set	$O(2^m \cdot m \cdot n)$
	Create the state's predicate vector	$O(nifs \cdot m \cdot n)$
DFA constr.	Create initial DFA states	$O(2^N)$
	For each DFA state T :	$O(D)$
	Create predicate vector format vf	$O(M \cdot nifs)$
	For each predicate vector v as a variation of vf	$O(2^{nv})$
	Create an empty successor set Q	$O(1)$
	For each NFA state $N \in T$ with predicate vector u	$O(M)$
	if $v \subseteq u$ then	$O(nifs)$
	add N 's successors to Q	$O(2^m \cdot M)$
	if Q does not exist, add it to the DFA	$O(D \cdot M)$

(b)

Figure 12: The complexity of the NFA and DFA construction algorithms. (a) Notation. (b) Complexity of the main steps of the algorithms. It is assumed that a set is implemented as a sorted array of included elements.

estimate how D depends on other parameters, such as N . However, it is expected that D is much less than its theoretical upper bound 2^L , where $L=2^N$. Experiments prove this expectation.

In order to estimate the complexity, we have generated a set of final schedules and ran the algorithm with them. Each final schedule consisted of one IF operation that was split at adjacent predicate instances in columns $-splitWidth/2 + splitOffset..splitWidth/2 + splitOffset$. The obtained instances were then moved up and/or down, so their predicate matrices were shifted in the range $-shiftWidth/2 + shiftOffset..shiftWidth/2 + shiftOffset$. Values $shiftWidth$ and $splitWidth$ varied in the range 1..3, and $shiftOffset$ and $splitOffset$ in the range -3..3, producing a set of 429 samples.

Figure 13a shows the algorithm's total execution time at a 300 MHz Pentium II processor, in relation with the value of D . The results are grouped in two series according to whether samples included a speculative split of IF or not. It may be easily noticed that all the experiments without speculative splits ran in 50 ms or less, regardless to the value of D , while those with speculative splits have shown large variations of execution time for a same D , and produced runtime explosion in some cases. Figure 13b shows the value of D in relation with the value of 2^N . Again, for the samples without speculative splits, the correlation was very close to linear with a slope less than 1, meaning that the size of the DFA is expected to be smaller than that of the NFA. For the samples

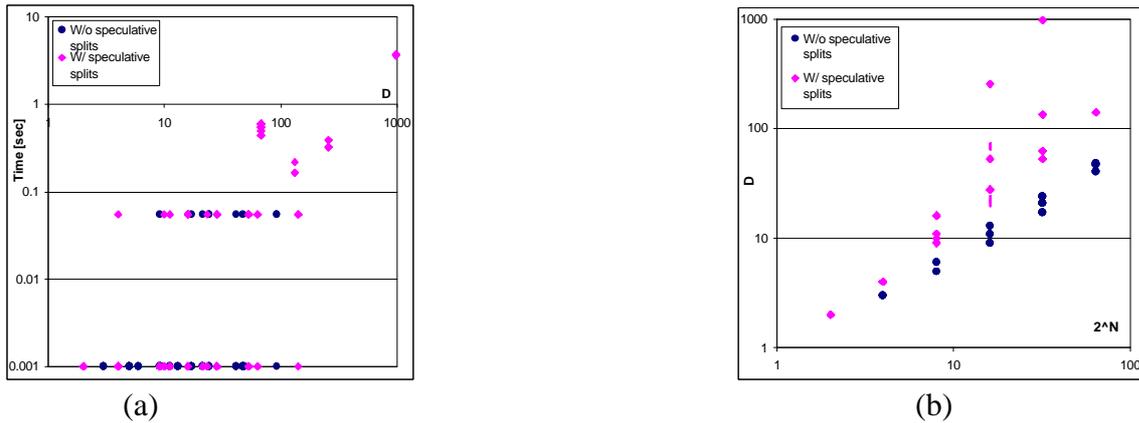


Figure 13: Experimental results. (a) The algorithm's execution time in relation with the number of DFA states (logarithmic scale). (b) The number of DFA states in relation with the number of NFA states (logarithmic scale).

with speculative splits, the correlation is again undeterminable and the values vary in large ranges. It may be concluded that for the schedules without IFs speculatively split, the algorithm is expected to run with an acceptable time and space complexity. When speculative splits exist, however, time and space explosion might be expected. Note once again that this does not include *speculatively moved* IFs, which are quite "harmless" for this technique, unlike for many other techniques. This precaution is directed to *speculatively split* IFs, which, fortunately, are of theoretical significance only and may not be expected to occur in real scheduling techniques.

7 Conclusions

We have proposed a new intermediate representation that may be used in modulo scheduling or other techniques for software pipelining loops with conditions. It allows separation of operations from different paths and their conditional movement. We have also defined an algorithm that transforms the representation into the executable code. The algorithm uses the notion of finite automata to represent the execution of separate paths as threads of control that are canceled or approved by executed IF operations. Our future work may follow two directions. First, we will try to use the formalism to investigate further how some important aspects of scheduling, such as code size, are affected by parameters such as predicate scope or level of speculation. Second, we will try to understand better how some existing scheduling techniques

work by using our model and to improve them, or to search for novel techniques which would benefit from our model.

References

- [1] Aho, A., Sethi, R., Ullman, J., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986
- [2] Aiken, A., Nicolau, A., "Optimal Loop Parallelization," *Proc. ACM SIGPLAN 1988 Conf. Prog. Lang. Design and Implementation*, 1988, pp. 308-317
- [3] Allen, J. R., Kennedy, K., Porterfield, C., Warren, J., "Conversion of Control Dependence to Data Dependence," *Proc. 10th ACM Symp. Principles of Prog. Lang.*, 1983, pp. 177-189
- [4] Ebcioğlu, K., "A Compilation Technique for Software Pipelining of Loops With Conditional Jumps," *Proc. 20th Ann'l Workshop on Microprogramming (MICRO-20)*, 1987, pp. 69-79
- [5] Ferrante, J., Ottenstein, K. J., Warren, J. D., "The Program Dependence Graph and its Use in Optimization," *ACM Trans. Prog. Lang. and Systems*, Vol. 9, July 1987, pp. 310-349
- [6] Milicev, D., Jovanovic, Z., "A Formal Model of Software Pipelining Loops with Conditions," *Proc. 11th Int'l Parallel Processing Symp. (IPPS '97)*, 1997, pp. 554-558
- [7] Milicev, D., Jovanovic, Z., "Predicated Software Pipelining Technique for Loops with Conditions," *Proc. 12th Int'l Parallel Processing Symp. (IPPS '98)*, 1998
- [8] Milicev, D., Jovanovic, Z., "Code Generation for Software Pipelined Loops with Conditions," Technical Report TI-RTI-99-0041, University of Belgrade, Faculty of Electrical Engineering, 1999
- [9] Moon, S.-M., Ebcioğlu, K., "An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW processors," *Proc. 25th Ann'l Int'l Symp. Microarch. (MICRO-25)*, 1992, pp. 55-71
- [10] Nikolau, A., "Percolation Scheduling: A Parallel Compilation Technique," TR-85-678, Cornell Univ., 1985
- [11] Rau, B. R., "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops," *Proc. 27th Ann'l Int'l Symp. Microarch. (MICRO-27)*, 1994, pp. 63-74
- [12] Rau, B. R., Schlansker, M. S., Tirumalai, P. P., "Code Generation Schema for Modulo Scheduled Loops," *Proc. 25th Ann'l Int'l Symp. Microarch. (MICRO-25)*, 1992, pp. 158-69
- [13] Schwiegelshohn, U., Gasperoni, F., Ebcioğlu, K., "On Optimal Parallelization of Arbitrary Loops," *J. of Parallel and Distributed Computing* 11, 1991, pp. 130-134
- [14] Shim, S., Moon, S.-M., "Split-Path Enhanced Pipeline Scheduling for Loops with Control Flows," *Proc. 31st Ann'l Int'l Symp. Microarch. (MICRO-31)*, 1998
- [15] Stoodley, M., Lee, C., "Software Pipelining Loops with Conditional Branches," *Proc. 29th Ann'l Int'l Symp. Microarch. (MICRO-29)*, 1996
- [16] Su, B., Wang, J., "GURPR*: A New Global Software Pipelining Algorithm," *Proc. 24th Ann'l Workshop Microprog. and Microarch. (MICRO-24)*, 1991, pp.212-216
- [17] Tang, Z., Chen, G., Zhang, C., Zhang, Y., Su, B., Habib, S., "GPMB-Software Pipelining Branch-Intensive Loops," *Proc. 26th Ann'l Int'l Symp. Microarch. (MICRO-26)*, 1993
- [18] Warter, N. J., Bockhaus, J. W., Haab, G. E., Subramanian, K., "Enhanced Modulo Scheduling for Loops with Conditional Branches," *Proc. 25th Ann'l Int'l Symp. Microarch. (MICRO-25)*, 1992, pp.170-179

- [19] Warter, N. J., Mahlke, S. A., Hwu, W-M. W., Rau, B. R., "Reverse If-Conversion," *Proc. ACM SIGPLAN 1993 Conf. Prog. Lang. Design and Implementation*, 1993, pp. 290-299
- [20] Warter-Perez, N. J., Partamian, N., "Modulo Scheduling with Multiple Initiation Intervals," *Proc. 28th Ann'l Int'l Symp. Microarch. (MICRO-28)*, 1995, pp. 111-118