

A Formal Model of Software Pipelining Loops with Conditions

Dragan Milicev and Zoran Jovanovic
University of Belgrade
E-mail: emiliced@etf.bg.ac.yu

Abstract

This paper addresses the problem of parallelizing loops with conditional branches in the context of software pipelining. A new formal approach to this problem is proposed in the form of Predicated Software Pipelining (PSP) model. The PSP model represents execution of a loop with conditional branches as transitions of a finite state machine. Each node of the state machine is composed of operations of one parallelized loop iteration. The rules for operation movements between nodes in the PSP model are described. The model represents a new theoretical framework for further investigation of inherent properties of these loops.

Keywords: instruction level parallelism, loops with conditional branches, software pipelining.

1 Introduction

Applications that call for efficient execution generally spend most of their running time in loops. Hence, loops are extremely interesting for parallelization. Loops that do not contain conditional branches have been both theoretically and practically analyzed in detail [1,2]. There are few theoretical and empirical results for loops with conditions [3], and several effective methods for their optimization [4,5,6,7]. This work addresses the problem of modeling such loops in the context of software pipelining, in order to enlighten the way towards their better understanding. A more detailed version of this paper can be found at <http://ubbg.etf.bg.ac.yu/~emiliced>.

2 Problem analysis

This section describes the conclusions of our analysis of the problem. Some inherent properties of the loops and the motivation for our model are explained.

2.1 Inherent properties of the loops

Every specific execution of a loop with conditional branches is characterized with the number of iterations, and the outcomes of the branches (conditions) during the execution. It can be described with the data dependency graph (DDG) that incorporates all operations that have been executed according to the outcomes of the conditions, and dependencies between them. The graph has its critical path(s), and this path dictates the shortest possible execution time, again for that specific execution.

The heart of our problem is that the described DDG alters its shape from one execution to another, due to various outcomes of conditions. In some boundary cases, the DDG of a loop may have a shape of a graph of a DOALL loop, with all iterations being independent. For some other executions of the same loop, the graph may turn into a chain of dependencies of a DOACROSS loop, with little or no opportunities for parallelization. Since we cannot *predict* in advance the outcomes of all the branches, we cannot *adjust* the target code to every single execution. We continue our analysis by observing the effects that outcomes of the conditions produce on the DDG of the loop.

Our analysis suggests separating the meanings of *conditional operation* (or *condition* for short) and *outcome* of the condition. An outcome of a condition is a Boolean value that actually affects the shape of the DDG. It is an abstract term that we will also refer to as *predicate*. A conditional operation is an operation that performs some computation, thus occupying some resources, and defines the actual value of the predicate. This operation is connected with other operations by data dependencies. Such a separation provides better focus on the problem of defining the schedule of an iteration, determined by a combination of predicates.

We first have to semantically decouple these two entities and adjust the code according to various values of the predicates. As a result of this adjustment, which incorporates operation movements guarded by the values

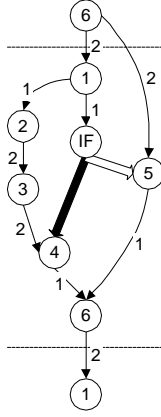


Figure 1: DDG of a loop iteration. Dotted lines represent the iteration boundaries

of the predicates, we get some *predicated* code. Some operations are executed only if a certain predicate has a certain value. Finally, we recouple the two entities by observing at what time a predicate is actually computed (when its own conditional operation has been scheduled). Two cases may occur. First, the conditional operation may be computed before the moment its predicate is used for the first time, perhaps several iterations away. This case suggests recording of the outcomes of several conditional operation instances from previous iterations. Second, the outcome may be used before its conditional operation has been computed. This is the occurrence of *speculative execution*. Both features can (but need not) be supported by hardware.

2.2 Motivation

As a result of different outcomes of the conditions, some operations in the DDG of the unrolled loop that are control dependent upon the conditions do not exist. Consequently, the data dependency edges that come into or go out from them do not exist. This is the primary effect of changing the shape of the DDG that opens possibilities for parallelization. For example, consider the loop DDG in Figure 1. Control dependencies are represented by bold edges, with *True* denoted by the filled edge, and *False* by the hollow one. Data dependencies are marked with operation latencies expressed in clock cycles.

If the outcome is *True*, operation 4 is executed, and operation 5 does not exist. There is a dependency chain 6-1-2-3-4-6 that limits the latency of an iteration to at least 8 cycles. If the outcome is *False*, operation 4 does not exist, and operation 3 becomes bottom-free, so it can be moved downwards into the next iteration. However, there is a chain 6-5-6 that dictates the execution of the iteration in at least 3 cycles. Consequently, the chain 1-2-3 should

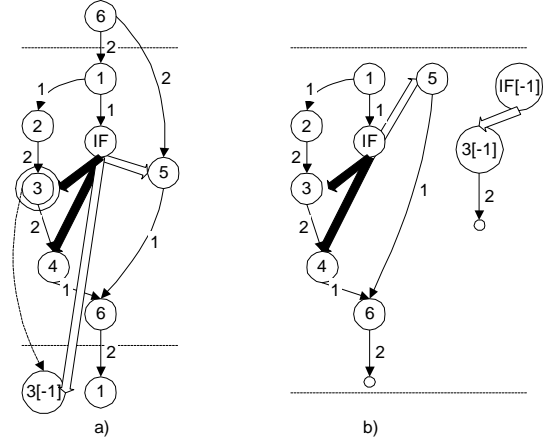


Figure 2: Operation movement

be adjusted to this latency of 3 cycles, to prevent an increase in latency of the iteration. This can be done by moving operation 3 into the next iteration, as in Figure 2a. This movement of operation 3 does not affect the latency of the next iteration, since it also lasts for at least 3 cycles.

We may conclude that operation 3 is moved into the next iteration only if the outcome is *False*. Thus, operation 3 from the considered iteration exists in this iteration under the condition *True*, as shown in Figure 2a. Otherwise, this operation exists in the next iteration, as an instance of operation 3 from the previous iteration, denoted by 3[-1]. Figure 2b shows the net effect of the described movement on the code of a single iteration. Since software pipelining must produce a "periodic" code, every single iteration can have the instance 3[-1] controlled by the instance $IF[-1] = False$, and the instance 3[0] controlled by $IF[0] = True$.

Two things can be noticed. First, each iteration may have from null up to two instances of the same operation 3, controlled by different instances of the same condition. The combination of their outcomes determines the actual number of the instances of operation 3. This way, we can get the predicated code for each iteration, i.e., a set of operations that should be executed under certain composed condition ($IF[0]IF[-1]$ in the example). Second, we can see that an instance may depend on an outcome of a condition from previous iteration, which raises the need for recording the outcomes from previous iterations.

3 The model

The proposed model for treating loops with conditional branches is referred to as Predicated Software Pipelining (PSP). To simplify, we first consider loops with non-nested IF statements. We will discuss nested IFs later.



Figure 3: General form of the predicate matrix. Non- b elements are shaded, and the column with index 0 is bordered

3.1 The PSP model

Our analysis has pointed out that the code of a transformed iteration consists of different instances of operations, controlled by different instances of predicates (Figure 2b). In other words, different combinations of predicate instances dictate the contents of an iteration. The predicate instances come from different iterations. Consequently, a loop iteration can be described by the *predicate matrix*, having m rows and n columns, where m is the number of conditional operations in the loop, and n is an arbitrary number that determines the maximum scope (number of adjacent iterations) of a predicate. For example, suppose that a loop has two conditions, denoted by $p1$ and $p2$. The predicate matrix may be:

$$\begin{bmatrix} p1[-1] & p1[0] & p1[1] \\ b & p2[0] & p2[1] \end{bmatrix}$$

This matrix describes that the contents of an iteration are (or may be) dictated by predicate instances $p1[-1]$, $p1[0]$, $p1[1]$, $p2[0]$, and $p2[1]$, where $p[0]$ denotes the instance from current iteration, $p[-1]$ from previous, etc. The symbol b serves only to fill in an empty element of the matrix, because $p2[-1]$ is of no interest.

For the purpose of operations movements, we pose the following restrictions to predicate matrix construction:

1. $(\forall p \in P)(\forall i, j \in \mathbb{Z})(i < j \wedge PM(p, i) \neq b \wedge PM(p, j) \neq b \Rightarrow (\forall k \in \mathbb{Z})(i \leq k \leq j \Rightarrow PM(p, k) \neq b))$, where P is the set of predicates, and $PM(p, i)$ denotes the element of the predicate matrix in the row p and column i . Simply said, every row in the matrix must not have "b-holes:" these would be b elements between non- b elements.

2. The predicate matrix must have a column with index 0, and all elements of this column must be non- b .

Consequently, a general form of the predicate matrix is as in Figure 3. It is important to emphasize that the rows of the matrix may have different non- b elements, so that different instances of different conditions are taken into account. Consequently, we can enlarge the scopes of only those predicates that provide a better schedule. On the other side, loop unrolling multiplies the scope of each

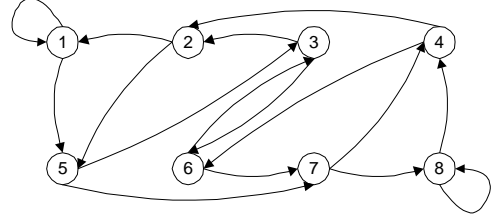


Figure 4: State transition diagram of a PSP model with 8 states

predicate the same way. This is one of the advantages of PSP over loop unrolling.

Every specific execution of the loop consists of iterations, and each iteration can be described with a matrix containing concrete values of the predicate matrix elements. The matrix with concrete values is called the *state matrix*. For the same example, if the execution sequence (the sequence of the predicates' outcomes) is (1 stands for *True* and 0 for *False*):

$p1$: ..., 1, 1, 0, 0, 1, ...

$p2$: ..., 0, 0, 0, 1, 0, ...

then the corresponding sequence of the state matrices is:

$$\dots, \begin{bmatrix} 1 & 1 & 0 \\ b & 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ b & 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 1 \\ b & 1 & 0 \end{bmatrix}, \dots$$

The way this sequence has been obtained is obvious: the next state matrix is derived from the previous one by shifting all its values one place left, discarding the left-edge elements, and filling the right-edge elements with the "arriving" predicate outcomes. An edge element is the one that is on the border of the matrix, or that is adjacent to a b element.

This approach leads to the idea of representing an execution of a loop by transitions in a finite state machine. The machine is determined by the predicate matrix. It has 2^k states, where k is the number of non- b elements in the predicate matrix. Every state is determined by a state matrix. A transition from state A to state B exists if state B can follow A in a legal execution sequence. In other words, the successor states of the state A are determined by shifting its state matrix one place left, discarding the left-edge elements, and filling the right-edge elements with all 2^m (m is the number of conditional operations) variations of the set $\{0, 1\}$. Hence, every state has 2^m predecessors and the same number of successors.

For example, consider a loop with one conditional operation and the predicate matrix $[p[-2] \ p[-1] \ p[0]]$. The states can be numbered as follows:

1. [000] 2. [100] 3. [010] 4. [110]
5. [001] 6. [101] 7. [011] 8. [111]

1: [000]	2: [100]	3: [010]	4: [110]
1[0]	1[0]	1[0]	1[0]
2[0]	2[0]	2[0]	2[0]
3[0]	3[0]	3[0]	3[0]

5: [001]	6: [101]	7: [011]	8: [111]
1[0]	1[0]	1[0]	1[0]
2[0]	2[0]	2[0]	2[0]
3[0]	3[0]	3[0]	3[0]
4[0]	4[0]	4[0]	4[0]

Figure 5: An example of initial schedule

The state transition diagram is shown in Figure 4.

An interesting property of the diagram can be easily proved for general case: exactly 2^m nodes have the same set of 2^m successors, and these 2^m successors have only the considered 2^m nodes as predecessors. Hence, the transitions group the states in clusters: the set of nodes that have the same set of successors will be called the *source cluster*, and the set of the successors will be called the *destination cluster*. For example in Figure 4, the nodes 1 and 2 form a source cluster, with the corresponding destination cluster consisting of the nodes 1 and 5.

3.2 Operation moves

In the PSP model, every node of the state machine comprises a set of operations that are executed if the execution sequence passes through that node. This set of operations can be represented by an acyclic DDG which is an extract of the whole DDG of the (infinitely) unrolled loop. This DDG has its own critical path(s).

It might happen that the DDGs of two adjacent nodes, when the nodes are concatenated in an execution, have a common critical path shorter than the sum of their separate critical paths. This is the opportunity for loop parallelization: some operations should be moved from one of the iterations into the other, in order to adjust the sum of the separate critical paths to the common one.

In the PSP model, an operation move should be performed between two adjacent states. The move should preserve the semantics of the execution. Consider the diagram in Figure 4. Suppose that an operation should be moved from state 3 to state 6. Preservation of semantics requires that if that operation is to be executed in state 3, it should be executed in every path that passes through that state. If it is moved into state 6, it should be also moved into state 2. After this move, the operation will be executed in every path that passes through the states 2 and 6, including the paths that come from the node 4. Consequently, the requirement for the move was that the

1: [000]	2: [100]	3: [010]	4: [110]
1[0]	1[0]	1[0]	1[0]
2[0]	2[0]	2[0]	2[0]
3[-1]		3[0]	3[0]

5: [001]	6: [101]	7: [011]	8: [111]
1[0]	1[0]	1[0]	1[0]
2[0]	2[0]	2[0]	2[0]
3[0]	3[0]	3[0]	3[0]
4[0]	4[0]	4[0]	4[0]

Figure 6: The schedule after a move of operation 3 downwards

operation existed in both states 3 and 4, and it was bottom-free in both states (none of the other operations in these states was dependent on it).

To sum up, we define a move of operation $Op[i]$ *downwards* from all the states of a source cluster into all the states of its corresponding destination cluster, if $Op[i]$ exists in all these states and is bottom-free in all of them. The operation becomes $Op[i-1]$ in the destination cluster, which indicates that the operation origin is the previous iteration, and that the index adjustment should be made. In a similar way we define a move *upwards*, from a destination cluster to its appropriate source cluster: the operation should be top-free and becomes $Op[i+1]$.

For example, consider a simple loop and the initial schedule as in Figure 5. The model is initialized so that every state consists of the operations that are to be executed under condition $p[0]$. The tables in Figure 5 represent the contents of the states, with the operations scheduled according to their dependencies.

For example, the operation 3[0] exists in both states 1 and 2 of a source cluster and is bottom-free. It can be moved into the states 1 and 5 of the destination cluster. The schedules (DDGs) of the states 1 and 5 are updated according to possible dependencies from the incoming operation 3[-1] towards the existing operations in the states 1 and 5. The result is as in Figure 6.

As a result, we obtain shorter schedule of the states 1 and 2, without lengthening the schedules of the states 1 and 5, since there are no dependencies on 3[-1].

We underline that this discussion represents only an approach to treating loops with conditional branches and defines the rules for operation moves. Concrete techniques, i.e., heuristics for selecting moves and scheduling could be defined separately.

1: [0]	2: [1]
1[0]	1[0]
2[0]	2[0]
3[0]	3[0]
	4[0]

Figure 7: Initial schedule for 2-state PSP

1: [0]	2: [1]
2[0] 1[+1]	2[0]
3[0]	3[0]
	4[0]
	1[+1]

Figure 8: The schedule for 2-state PSP after operation 1/1 moved upwards

1: [00]	2: [10]	3: [01]	4: [11]
2[0] 1[1]	2[0]	2[0] 1[1]	2[0]
3[0]	3[0]	3[0]	3[0]
	4[0]		4[0]
	1[1]		1[1]

Figure 9: Expanded schedule for a 4-state PSP

3.3 Model expansion

Consider the same loop in Figure 5, but with the predicate matrix $[p[0]]$. There are only two states 1 and 2 with the initial contents as in Figure 7.

There are state transitions from states 1 and 2 into the same states. Note that not any operation can be moved downwards. Only the operation 1[0] can be moved upwards. After that move, we get the schedule in Figure 8. Now the operation 2[0] can be moved upwards, but this move does not bring any improvement (it is dependent on operation 1). An inherent property of the model inhibits further movements because the predicate matrix does not provide information about interference between conditions in adjacent iterations. If we *expand* the predicate matrix by adding $p[1]$, we get a 4-state model as in Figure 9.

This model allows further movements. It can be easily shown that this schedule could be obtained by starting from matrix $[p[0] \ p[1]]$ and the initial schedule. We conclude that expansion of the matrix can open possibilities for new movements and improvements of the schedule. Furthermore, without detailed justification, we state that the *left* expansion, i.e., adding less indexes into the predicate matrix, enables downward moves; similarly, *right* expansion enables upward moves.

5.4 Execution model

The basic principle of PSP schedule execution is simple. If we have, for example, the predicate matrix $[p[-2] \ p[-1] \ p[0]]$, the transformed loop can be represented by a CASE structure:

```

CASE (p[-2]p[-1]p[0]) OF
  000: ... // code for state 000
  001: ... // code for state 001
  ...    // etc.
END CASE

```

Several issues can be noticed. First, some of the states might have parts of code in common. These are the operations that do not depend on the predicates that distinguish these states. In this case, some of the code replication can be eliminated. This issue is correspondent to *unification* in other techniques [8].

Second, the combination $p[-2] \ p[-1] \ p[0]$ indicates the need for storing the results of conditional operations belonging to previous iterations. Third, it might happen that certain predicate is determined by a conditional operation that is scheduled after the use of the predicate. In this case, the PSP state machine is nondeterministic with the following meaning. There are several "current" states at any moment of execution. When a state is to be exited, the machine passes nondeterministically to its successors, because the corresponding predicate has not been defined yet. When a conditional operation that defines the predicate has been executed, some of the nondeterministic paths (i.e., current states) of execution are discarded.

Loops with nested IFs can easily fit in the PSP model. For example, if we denote the outer IF's predicate by $p1$, and the inner IF is in THEN branch, we can denote the inner predicate by $p2$. The predicate matrix will have two rows, and the initial schedule will contain the same operations in both states in which $p1[0]=0$ ($p2[0]$ is either 0 or 1, actually it does not exist).

4 Conclusion

We have proposed a new approach to treating the problem of software pipelining loops with conditions—the PSP model. The approach can serve as a new research direction in the domain, since the preliminary analysis

shows some interesting regularities. It can also be a framework for concrete optimizing techniques.

References

- [1] Bacon D. F., Graham S. L., Sharp O. J., "Compiler Transformations for High-Performance Computing," *ACM Computing Surveys*, Vol. 26, No. 4, Dec 1994
- [2] Gasperoni F., "Compilation Techniques for VLIW Architectures," Tech. Rep. #435, New York University, Comp. Sci. Dept., Mar 1989
- [3] Schwiegelshohn U., Gasperoni F., Ebcioğlu K., "On Optimal Parallelization of Arbitrary Loops," *J. Par. Distr. Computing* 11, 1991, pp. 130-134
- [4] Moon S.-M., Ebcioğlu K., "An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW processors," *Proc. 25th Annual Intl Symp Microarchitecture (MICRO-25)*, Dec 1992
- [5] Su B., Wang J., "GURPR*: A New Global Software Pipelining Algorithm," *Proc. 24th Annual Workshop on Microprogramming and Microarchitecture (MICRO-24)*, Nov 1991
- [6] Tang Z., Chen G., Zhang C., Zhang Y., Su B., Habib S., "GPMB-Software Pipelining Branch-Intensive Loops," *Proc. 26th Annual Intl Symp Microarchitecture (MICRO-26)*, Dec 1993
- [7] Warter N. J., Bockhaus J. W., Haab G. E., Subramanian K., "Enhanced Modulo Scheduling for Loops with Conditional Branches," *Proc 25th Annual Intl Symp Microarchitecture (MICRO-25)*, Dec 1992
- [8] Nikolau A., "Percolation Scheduling: A Parallel Compilation Technique," Tech. Rep. TR-85-678, Cornell University, 1985