Predicated Software Pipelining Technique for Loops with Conditions

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

Abstract

An effort to formalize the process of software pipelining loops with conditions is presented in this paper. A formal framework for scheduling such loops, based on representing sets of paths by matrices of predicates, has been proposed. Usual set operations and relationships may then be applied to such matrices. Operations of a loop body are placed into a single schedule with the flow of control implicitly encoded in predicate matrices. An algorithm that generates loop code from such an encoded schedule has been described. The framework is supported by a concrete proposed technique that iteratively parallelize loops, as well as with heuristics driven by data dependencies to efficiently shorten loop execution. Preliminary experimental results of our prototype implementation prove that the proposed framework, technique, and heuristics produce efficient code at acceptable cost.

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

1 Introduction

Software pipelining [8] is one of the most commonly used frameworks for parallelizing loops on the instruction level. Its idea is in overlapping parts of adjacent iterations, so the next original iteration can start before the preceding one has been completed. As a result, machine resources are used by operations from different initial iterations, sometimes notably increasing parallelism.

Loops with conditional branches have irregular flow of execution, and just a few effective scheduling techniques exist for these loops. This work introduces a formal framework for scheduling these loops, and then proposes a new technique in the context of software pipelining. The technique is inspired by a recently proposed model of pipelining loops with conditions [4] named "Predicated Software Pipelining" (PSP). This section continues with a more precise problem statement, and a brief overview of the related work, including a short description of the underlying PSP model. Section 2 presents the approach in an intuitive manner. Section 3 briefly describes our current implementation. The paper ends with conclusions. A more detailed version of this paper can be found at [5].

1.1 Problem Statement

As an example to demonstrate various aspects of the approach throughout the paper, we will consider a simple loop with one conditional statement that finds the vector minimum:

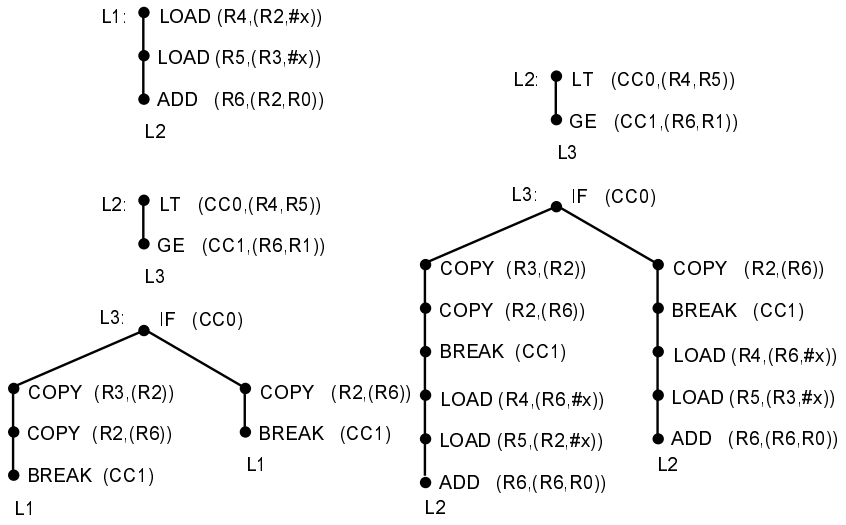
```
for (k=0; k<n; k++)
  if (x[k]<x[m]) m=k;
```

To set up the environment, we assume the underlying IBM VLIW architecture as described in the series of papers [2][6]. The architecture allows execution of tree VLIW instructions that consist of ALU and LOAD/STORE operations, as well as IF operations that determine the choice of one path down the tree instruction, in a single cycle. Second, due to the reasons that will be clear later, we introduce BREAK operation that tests a conditional register (CC) just as IF operation, but has one control flow branch that exits the loop; IF operation has both branches that stay inside the loop body. Assuming also the following register allocation: R0←1, R1←n, R2←k, R3←m, we get the initial assembly code in Figure 1a (the while-do loop has been transformed into a do-while loop with an initial test in front that we do not consider further).

If executed on a sequential machine, the latency of one loop iteration, called *initiation interval* (*II*), is 7 and 8 clock cycles for the two paths. Applying local scheduling with renaming, without moving operations across loop boundaries, and assuming sufficient parallelism in the hardware, *II* of 3 cycles can be obtained as shown in Figure 1b. VLIW instructions are depicted as tree structures of operations executed in the same cycle. Each instruction is labeled and has an explicit pointer to the successor instruction at each leaf.

By introducing software pipelining even shorter *II* can be achieved. All operations from the instruction L1 can be

```
//if (x[k]<x[m])m=k;
LOAD (R4, (R2,#x))
LOAD (R5, (R3,#x))
LT   (CC0, (R4,R5))
IF   (CC0) {
  COPY  (R3, (R2))
}
Continue:
// k++
ADD (R2, (R2,R0))
//for (...; k<n;
GE  (CC1, (R1,R1))
BREAK (CC1)
```



a) b) c)

Figure 1

scheduled together with the operations of L3 from the previous iteration. Some true data dependencies are eliminated by combining [6]. The transformed loop body with *II* of 2 cycles is shown in Figure 1c. Proper startup of the loop is provided by the operations in the preloop; in this example, these are the operations from L1 that are pushed into the previous iteration (not shown in the picture).

To sum up, the problem is to find out a technique that will define rules for scheduling operations of initial loop body with overlapping operations from different iterations, taking into account limitations of hardware resources, to obtain the smallest possible *II*. We will search for the solution in the context of software pipelining with variable *II*, meaning that *II* may differ from one execution path to another.

1.2 Overview of the Related Work

Techniques that schedule loops with conditions can produce single or multiple *II* [2][6][7][9][10][11][12][13]. Dealing with multiple *II* is more complex and many techniques produce a single fixed *II* [10][11][12].

Although the essentiality of the notion of paths in loops with conditions has been understood for long, there has been no attempts to formally define this concept in a way that can be used in scheduling. Moreover, most of the techniques use the term *path* only to refer to one path of control in a single iteration [9]. However, the efficiency of software pipelining is due to utilizing absence of data dependencies between operations from different iterations. For loops with conditions, the problem is hard because there is an unlimited and unpredictable number of paths through the whole loop.

A recent work [4] proposed a model named Predicated Software Pipelining (PSP) as a new viewpoint to execution of loops with conditions. Skipping the details, we interpret the way PSP encodes paths of execution in such loops. First, PSP defines a *predicate* as a Boolean variable that represents possible outcomes of one IF operation in one iteration. The concrete value (True–1 or False–0) of a predicate controls execution of some operations of the loop that originally belong to the branches of the corresponding IF. The model separates the notions of predicate and IF operation: a predicate controls scheduling, while an IF operation computes its outcome. If an operation is controlled by a predicate, but scheduled before the corresponding IF that computes its outcome, that operation is executed speculatively.

Paths that span over several iterations are represented by *predicate matrices* with *m* rows, where *m* is the number of IFs in the loop, and *n* columns, where *n* is an arbitrary number that limits the scope of predicates. The columns are indexed with integer numbers, and the column *i* refers to the *i*-th iteration starting from the current. The column 0 refers to the current iteration; its elements will be underlined for clearance in our notation. A predicate matrix may contain special *b* symbols; *b* stands for both outcomes. For example, the matrix $\begin{bmatrix} 1 & \underline{1} & 0 \\ b & \underline{0} & 1 \end{bmatrix}$ denotes the paths that go over the following outcomes of the first IF operation (the first row): True for the previous (column -1), True for the current, and False for the next iteration (column 1); and for the second IF operation (the second row): False for the current and True for the next iteration.

2 The Idea of the PSP Technique

We will here describe the idea of the proposed technique in an informal, intuitive way, using the given simple loop. More formal descriptions exist but are beyond the scope of this paper [5]. Basically, each operation in each iteration of the loop is controlled by a combination (or cross-product) of none to several outcomes of IFs. Initially, these are IFs from the same iteration. After operation moves across loop boundaries, these can be outcomes from other iterations than the current [4]. In order to formally control the operation, we attach a predicate matrix to each operation. For the given sample loop with only one IF (BREAK is not counted), predicate matrices have one row and arbitrary number of columns. The initial assignment is:

```
LOAD  (R4, (R2,#x))   [b]
LOAD  (R5, (R3,#x))   [b]
LT    (CC0, (R4,R5))  [b]
IF    (CC0)           [b]
COPY  (R3, (R2))      [1]
ADD   (R2, (R2,R0))   [b]
GE    (CC1, (R2,R1))  [b]
BREAK (CC1)           [b]
```

As all of the operations except the COPY operation are control-independent, their matrices consist of *b* elements, meaning that they should be executed on both paths of the first and only IF from the current iteration. The COPY operation should be executed only on the True path of the only IF. Note the use of the word "should," since it represents the separation of control-dependence and computation of outcomes by IF operations. For example, the following order of operations in a possible schedule:

```
COPY  (R3, (R2))      [1]
IF    (CC0)           [b]
```

determines that the COPY operation will be executed speculatively, since it *should* be executed on the True path of IF, but this outcome has not been computed yet when COPY is to be executed.

In other words, a predicate matrix attached to an operation defines *a set of formal paths* on which the operation should be executed. These paths correspond to the execution paths of the initial loop. As a predicate matrix has (theoretically) an infinite number of columns, and the default value of its elements is *b*, each predicate matrix represents an infinite set of paths. A single predicate matrix is necessary and sufficient to describe the set of formal paths of an operation.

During scheduling, a control-dependent operation may be scheduled before its controlling IF. The operation still preserves its set of formal paths. Yet, it must be executed before the IF operation, and thus it is to be executed on both paths of the IF. Consequently, a *set of actual paths* can be derived for each scheduled operation. This set consists of all the paths that the operation is to be actually executed, according to the given schedule. Simply said, an operation that has different sets of formal and actual paths is executed speculatively.

A set of actual paths cannot be generally defined by a single predicate matrix. For example, one operation may be speculative on one path, and not speculative on another. The set of actual paths for such an operation can be defined with a set of predicate matrices. For example, two matrices that describe an actual set: [1 *b*] and [0 1] denote that the operation is to be executed on both paths of the IF from the current iteration, if the outcome of the previous iteration was True; otherwise, it is to be executed on the True path of the current iteration only. Obviously, the operation is control-dependent on the IF from the current iteration, and is scheduled speculatively only on the True path of the IF from the previous iteration.

In order to provide the link between IF operations and the predicates that they compute, we assign an *operation index* to each operation in the schedule. It denotes the original iteration that the operation belongs to, relatively to the current transformed iteration. Consequently, when an operation is moved into the previous iteration, its index is incremented, and its predicate matrix is shifted one place right, in order to preserve the relative reference. For example, if the COPY operation is moved into the previous iteration, it becomes: COPY (...) (+1) [b1]. This is a formal representation for an *operation instance* that performs a COPY operation, has its origin in the next iteration (+1), and should be executed under the outcome True of the IF instance from the next iteration, and both outcomes of the IF instance from the current and all other iterations.

The purpose of the operation index is twofold. First, it provides a link between an IF operation instance and a predicate. For example, the operation instance IF (CC0) (+1) [0] computes the predicate $p(+1)$, but the IF operation itself should be executed under the outcome of the same IF but from the current initial iteration ($p(0)$). Second, it can be used for vector index adjustment when a vector operation is moved across the iteration boundary.

Our scheduling technique is based on elementary transformations applied on the schedule. The schedule is defined as a list of rows, where each row consists of operation instances that are allocated in the same cycle. One important transformation is *moveup*. It is applied on an operation instance that is moved up from its original schedule row (cycle) into another schedule row. A move can scroll across the schedule, producing software pipelining. For example, moving up the first four operations of the initial sample loop schedule across the loop boundary, we can obtain the schedule in Figure 2.

While moving an operation upwards, several cases can occur. First, true data dependencies can prohibit further

Cycle1: COPY (R3,(R2)) (0)[1]
Cycle2: ADD (R2,(R2,R0)) (0)[b]
Cycle3: GE (CC1,(R2,R1))(0)[b]
Cycle4: BREAK (CC1) (0)[b]
Cycle5: LOAD (R4,(R2,#x)) (1)[b]
LOAD (R5,(R3,#x)) (1)[b]
Cycle6: LT (CC0,(R4,R5))(1)[b]
Cycle7: IF (CC0) (1)[b]

Figure 2

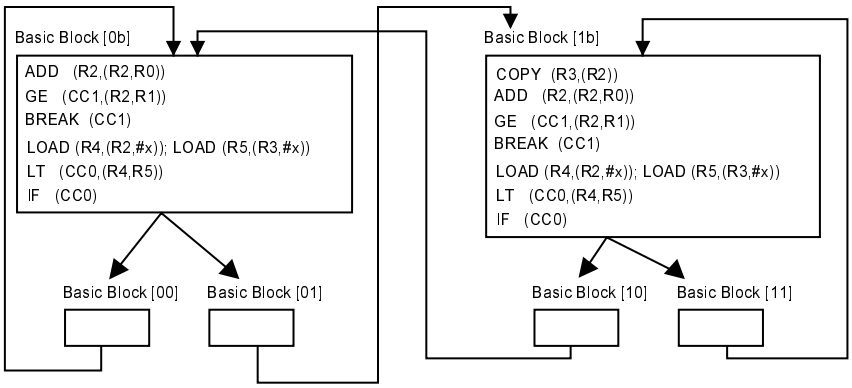


Figure 3

move, causing the transformation to fail. Some true dependencies can be eliminated by combining [6]. Output-, anti- and control dependencies are eliminated by renaming in a similar way as in [6]. The basic difference from other techniques that perform such transformations is that we do not explicitly store and update control flow paths, but represent them implicitly by the predicate matrices. Thus, if two operations belong to different formal paths, they will have at least one complementary predicate matrix element (1 and 0). Such predicate matrices are called *disjoined*. Operations with disjoined predicate matrices are not tested for data and control dependencies. Similarly, the *movedown* transformation is also defined.

Another important transformation is the *split* transformation. It splits one *b* predicate matrix element of an operation instance and creates two clone instances instead of the original one. This transformation corresponds to moving a joint point in the control flow graph downwards, except that it is applied on one operation only, which need not be the one that immediately follows the joint. The purpose of this transformation is to create operation instances that are disjoined with the operation instances from other paths. Such a transformation can enable further moves.

The set of actual paths is calculated for each operation after the scheduling, in the process of code generation. During scheduling, only predicate matrices for formal paths are considered. Our current framework does not support differences between formal and actual paths for IF operations, i.e., we do not allow speculative IF operations. This does not mean that we prohibit moving IFs across unrelated IFs, but only control-dependent IFs are not moved so.

A special auxiliary structure called *IFLog* traces the schedules of IF operations, in order to provide links between predicates and IF operations that compute their outcomes, i.e., to enable calculation of actual from formal path sets. Since an IF operation, as any other, may have non-*b* elements in its predicate matrix, i.e., may be conditionally executed, IFLog logs the schedules of IF operations on various paths.

The final step in the parallelization process is the loop code generation. We will here briefly describe the process of loop body generation. The process is actually the reconstruction of the control flow graph from the encoded form—operations with predicate matrices. The process constructs the graph composed of basic blocks. Each basic block is attached to a predicate matrix which now describes actual paths. The operation instances are placed in all basic blocks that have "compatible" predicate matrices. A basic block may end with an IF operation. At this moment, two new successor blocks are constructed, with the corresponding matrix element computed by the IF operation set to 0 and 1.

For the example in Figure 2, the process starts with two basic blocks [0 *b*] and [1 *b*] (Figure 3), since the schedule contains an operation instance that is control dependent on *p*(0), and the corresponding instance IF(0) is computed in the previous iteration (recorded in IFLog). Then, the COPY operation is placed only in the basic block [1 *b*]. Other operations are placed in both basic blocks, since their predicate matrices include both paths. Each basic block ends with an IF operation, that then defines the outcome *p*(1). The successors of the two blocks are constructed and the process of placing operations is continued. But since there are no more operations in the schedule, the new basic blocks are linked to the existing blocks by loop back edges. The linkage is defined by "superset" relationships between predicate matrix of the successor block and the left-shifted matrix of the predecessor block. Finally, the empty basic blocks are deleted.

3 Current Implementation

Our approach separates three concepts: the framework, the technique, and the heuristics [5]. The framework formally defines the notions of operation, predicate matrix, index, and relationships between matrices and operation instances. It also defines how executable code can be

obtained from a given schedule, and how resource constraints are handled.

The framework may be used in probably various contexts. Each context is defined by a concrete *technique* that determines how the final schedule is achieved. We have proposed a technique based on iterative application of elementary transformations on the schedule [5]. We speculate that even other approaches are possible. Our technique defines four elementary transformations: *split*, *unify*, *moveup*, and *movedown*. During this process, preloop and postloop code may be created.

For our technique, various *heuristics* can be applied for choosing transformations in the presence of resource constraints. Our current implementation is iterative. Each step first generates a set of candidate transformations. The candidate transformations are heuristically evaluated. The best transformation is applied. The process continues until there are no candidate transformations, or all candidate transformations fail due to resource conflicts. The algorithm has no backtracking. Generation of candidate transformations is directed towards shortening *II*.

To support the work with practical results, we have implemented a simple prototype scheduler. The prototype scheduler produces loop body code for the described VLIW architecture. Preliminary experimental results on our prototype implementation prove that the proposed framework, technique, and heuristics produce efficient code at acceptable cost [5].

4 Conclusion

We have proposed a new formal framework for software pipelining loops with conditions. The framework is based on the notion of formal and actual path sets that are efficiently represented by predicate matrices, which allows for defining mathematical set operations and possibly further investigation of properties of the loops. In contrast to the PSP *model* presented in [4], which is only a formal view of the loop execution, the framework presented here is directly oriented to scheduling and code generation. To illustrate this, we have outlined the algorithm for loop code generation from a given schedule that uses the notion of paths. We have implemented also a concrete scheduling technique with efficient heuristics. The heuristics are driven directly by data dependencies, which is very easy to obtain in the proposed framework.

The framework's main worth might be in possibility to incorporate other unknown techniques and heuristics. For example, heuristics driven by dynamic probabilities of path sets may be defined. The heuristics could force moves that shorten more probable paths while possibly ignoring less probable ones due to the resource constraints. The probabilities could be obtained by profiling, and a mapping from path sets (represented by predicate matrices) to their probabilities would enable exact calculation of estimated mean (dynamic) *II* of each intermediate schedule. We are currently investigating this option.

References

[1] Aiken A., Nikolau A., "Optimal Loop Parallelization," *Proc. SIGPLAN 1988 Conf Prog Lang Design and Implementation*, 1988

[2] Ebcioglu K., "A Compilation Technique for Software Pipelining of Loops With Conditional Jumps," *Proc 20th Annl Workshop on Microprog (MICRO-20)*, 1987

[3] Gasperoni F., "Compilation Techniques for VLIW Architectures," Tech. rep. #435, New York Univ., 1989

[4] Milicev D., Jovanovic Z., "A Formal Model of Software Pipelining Loops with Conditions," *Proc. 11th Intl Par Proc Symp (IPPS '97)*, 1997

[5] Milicev D., Jovanovic Z., "Predicated Software Pipelining Technique for Loops with Conditions," a detailed version of this paper, http://ubbg.etf.bg.ac.yu/~emiliced

[6] Moon S.-M., Ebcioglu K., "An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW processors," *Proc 25th Annl Intl Symp Microarch (MICRO-25)*, 1992

[7] Nikolau A., "Percolation Scheduling: A Parallel Compilation Technique," TR-85-678, Cornell Univ., 1985

[8] Rau B., Fisher J. A., "Instruction-level Parallel Processing: History, Overview, and Perspective," *Journal on Supercomputing*, Vol. 7, No. 1/2, May 1993

[9] Stoodley M., Lee C., "Software Pipelining Loops with Conditional Branches," *Proc 29th Annl Intl Symposium on Microarch (MICRO-29)*, 1996

[10] Su B., Wang J., "GURPR*: A New Global Software Pipelining Algorithm," *Proc 24th Annl Workshop on Microprog and Microarch (MICRO-24)*, 1991

[11] Tang Z., Chen G., Zhang C., Zhang Y., Su B., Habib S., "GPMB–Software Pipelining Branch-Intensive Loops," *Proc. 26th Annl Intl Symp on Microarchitecture (MICRO-26)*, 1993

[12] Warter N. J., Bockhaus J. W., Haab G. E., Subramanian K., "Enhanced Modulo Scheduling for Loops with Conditional Branches," *Proc. 25th Annl Intl Symp. on Microarchitecture (MICRO-25)*, 1992

[13] Warter-Perez N. J., Partamian N., "Modulo Scheduling with Multiple Initiation Intervals," *Proc. 28th Annl Intl Symp on Microarch (MICRO-28)*, 1995