

Sources of Parallelism in Software Pipelining Loops with Conditional Branches

Dragan Milicev and Zoran Jovanovic
University of Belgrade
School of Electrical Engineering
Department of Computer Engineering and Science
E-mail: emiliced@etf.bg.ac.yu

Abstract: *The subject of this paper is the instruction-level parallelism and the process of software pipelining loops with conditional branches. First, preconditions for treating such loops are introduced, and some effects of existence of conditional instructions and their outcomes that are important for parallelization are analyzed. These effects are emphasized and systematized.*

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

1 Introduction

Modern high performance computers with RISC, VLIW, or superscalar processors base their power on the ability to execute several operations in parallel. As a contrast to the higher level of parallelism, explicitly stated by high level programming language concepts, these machines use *instruction level parallelism*, which means that they extract parallelism from the executable (or intermediate) code. This extraction can be either *static*, when the compiler reorganizes initial sequence of instructions in order to group those operations that can be executed simultaneously [5], or *dynamic*, when the processor itself detects independent operations. Whichever the approach, a parallelizing compiler can make significant optimizations by reorganizing instructions of the initial program.

The basic precondition for parallel execution of several operations is their data-independence. The factors that affect the performance of a parallel execution of a program on a processor are:

- (1) the existence of data-independent operations in the program,
- (2) the capability of the compiler to discover these independent operations and to group them, and
- (3) the availability of processing elements and other resources in the processor to perform all currently independent operations.

The first factor is predominantly an inherent property of the algorithm, and the compiler can hardly affect it. The third factor has a steadily decreasing influence, due to the ever increasing power of processors, but will never vanish as a limiting factor in compilation. We are specially interested in investigating the second factor.

Applications that call for efficient execution generally spend most of their running time in loops. This is why loops are extremely interesting for parallelization. Loops that do not contain conditional branches (IF-THEN-ELSE constructs) have been both theoretically and practically analyzed in detail [2, 5, 9, 11, 20]. In contrast, there are few theoretical and empirical results for loops with conditional branches [10, 12, 22, 27], and several effective methods for their optimization [6, 7, 14, 16, 17, 23, 24, 25, 26, 28, 30]. This work addresses the problem of parallelization of such loops. Besides, we concentrate here on the innermost loops, since they iterate the most, and since there are techniques that optimize outer loops after transforming the inner ones [16].

The rest of the paper has the following outline. The problem is described in greater detail in Section 2, where some conditions and assumptions are stated. Section 3 describes some inherent properties of loops with conditional branches. Section 4 seeks sources of parallelism in such loops, in order to enlighten the way towards new software pipelining approaches. Section 5 is a conclusion.

2 Problem Statement

2.1 Problem Description

Consider a loop with a conditional statement as in Figure 1. Suppose that the target processor has enough resources to perform in parallel all operations that are independent, and that the given intermediate code can be directly mapped into the processor instructions and registers. Let the latency of addition be one, and of multiplication two clock cycles. For the purpose of this example, we will initially neglect the latency of the conditional operation (IF), assuming that there is a kind of hardware mechanism that supports this assumption.

```

DO I=1,N
O1:    D[I]=A[I-1]+B
O2:    E[I]=D[I]*C
O3:    F[I]=E[I]*A[I-1]
      IF (D[I]>0) THEN
O4:    G[I]=F[I]+5
      ELSE
O5:    J[I]=A[I-1]-2
      ENDIF
O6:    A[I]=G[I]*J[I]
END

```

Figure 1: Example of a loop with a conditional statement

With all other assumptions stated in the next subsection taken into account, we can see that the latency of one iteration of this initial loop is eight clock cycles for both branches of IF. If we apply a simple and greedy scheduling with speculative execution and renaming [5], but constrained to the code of a single iteration, we can get the transformed loop in Figure 2, with the *initiation interval (II)* of eight cycles for one, and five cycles for the other branch.

```

DO I=1,N
O1, O5':    D[I]=A[I-1]+B;   JR=A[I-1]-2
            IF (D[I]>0) THEN
O2:         E[I]=D[I]*C
O3:         F[I]=E[I]*A[I-1]
O4:         G[I]=F[I]+5
O6:         A[I]=G[I]*J[I]
            ELSE
O2, O6:     E[I]=D[I]*C;      A[I]=G[I]*JR
O3, O5":    F[I]=E[I]*A[I-1]; J[I]=JR
            ENDIF
END

```

Figure 2: Transformed loop from Figure 1, with greedy scheduling applied. Operations that can be performed in parallel are in the same line

We have moved the operation 5 upwards, above the conditional operation. It is now executed *speculatively*, and its destination variable J[I] has been temporarily renamed. Consequently, another

compensational operation was required to restore the value of $J[I]$ in the ELSE branch. We have also scheduled the operations 2 and 6 together in the ELSE branch, since the operation 4 does not exist and the operation 6 is dependent on the operation 5 only.

The previous transformation did not move any operation across the boundaries of the initial loop iteration. By applying this kind of movement, which leads to software pipelining, we can achieve II of eight and three cycles for the two branches. The transformed loop body is given in Figure 3.

```

                                Preloop
                                DO I=2,N-1
                                IF (D[I-1]<=0) THEN
O1,O5',O3[-1]:                D[I]=A[I-1]+B; JR=A[I-1]-2; F[I-1]=E[I-1]*A[I-2]
                                IF (D[I]>0) THEN
O2:                            E[I]=D[I]*C
O3:                            F[I]=E[I]*A[I-1]
O4:                            G[I]=F[I]+5
O6:                            A[I]=G[I]*J[I]
                                ELSE
O2,O6,O5":                    E[I]=D[I]*C; A[I]=G[I]*JR; J[I]=JR
                                ENDIF
                                ELSE
O1,O5':                        D[I]=A[I-1]+B; JR=A[I-1]-2
                                IF (D[I]>0) THEN
O2:                            E[I]=D[I]*C
O3:                            F[I]=E[I]*A[I-1]
O4:                            G[I]=F[I]+5
O6:                            A[I]=G[I]*J[I]
                                ELSE
O2,O6,O5":                    E[I]=D[I]*C; A[I]=G[I]*JR; J[I]=JR
                                ENDIF
                                ENDIF
                                ENDDO
                                Postloop

```

Figure 3: Transformed loop from Figure 1, with operations moved across the boundaries of the initial iteration. Operation O3[-1] is the instance of the operation O3 from the previous iteration

We will describe later the technique that has produced this schedule. Here we just want to draw the reader's attention to the occurrence of several instances of the same operation 3 from two adjacent iterations in a single new iteration of the transformed loop. The instance from the previous iteration is indexed with [-1]. Thus, there is a set of conditions from two adjacent iterations that dictates the execution of the transformed iteration. The set increases the number of basic blocks in the transformed iteration.

To conclude, the problem of loop parallelization is to find out a technique for applying movements of operations across initial iteration boundaries, in order to achieve a transformed loop with II s as short as possible.

2.2 Conditions and Assumptions

We impose the following conditions:

1. We will perform the problem analysis in the context of software pipelining. Software pipelining [9, 11] has proven to be an effective technique for optimizing loops in general. Informally, we define software pipelining as a technique that performs movements of operations across boundaries of iterations, while preserving periodicity of the loop body. The transformed loop consists of a *preloop* (or

prologue), that initializes the first p iterations, a transformed loop body, and a *postloop* (or *epilogue*), that completes the last p iterations.

Several formal definitions of software pipelining were proposed by other authors. However, we cannot accept the definition in [9] because it is too broad and allows even DOALL loops, with all iterations of the initial loop executed simultaneously, to be considered as pipelined. Furthermore, a definition which would require that only one instance of each operation exists in the transformed iteration is not appropriate, because in loops with conditional branches, as outlined in the previous subsection, the existence of several instances in one iteration may be useful. Finally, we find that the only common property of all existing software pipelining techniques, which allows existence of several instances of an operation in one iteration, is the property of *preserving the number of executed iterations*. More formally, we define a necessary condition for software pipelining in the following way: A transformed loop P' is achieved by software pipelining from the initial loop P only if for any execution I of P with a sufficiently large number n of iterations, and the semantically equivalent execution I' of P' with n' iterations is: $n = n' + p$, where p is the number of iterations initialized by the preloop, and completed by the postloop, and is independent on n .

2. We will search for code transformations that produce a variable II . The inherent property of loops with conditional branches is a variable latency of their iterations due to various outcomes of the conditions. We try to exploit this property, so we allow a variable II , which means that the execution time of an iteration of the transformed loop can last differently, depending on outcomes of the conditions.

We are introducing the following assumptions in order to make our conclusions architecture-independent:

(a) The input to our analysis is a loop body represented by the data dependency graph (DDG) of the intermediate code [1]. We assume that all dependencies have been detected, their iteration distances are known, and that the latency of the operations is deterministic.

(b) We assume that all usual transformations such as renaming (anti- and output dependencies elimination) and induction variable elimination [1, 5] have already been done, and we do not take them into account.

(c) We neglect the loop-overhead operations that increment the loop index variable and test the loop exit condition. These operations can be attached to the transformed loop body afterwards. Incrementing the loop-index variable is usually a data independent operation, so it can be attached anywhere in the transformed body, while all references to it (in vector indices) can be adjusted according to their new position relative to this operation.

(d) The problem of the preloop and postloop generation is beyond the scope of this paper, because we concentrate on the loop body only.

(e) We neglect resource constraints, assuming sufficient hardware parallelism.

(f) When vector operations are treated, we assume that there is a kind of indirect addressing supported by the machine. Moving such an operation in an adjacent iteration does not cause any additional overhead. For instance, if the operation $a[i] = \dots$ is moved to the previous iteration, it becomes $a[i+1] = \dots$. As the access to the vector a can be performed via an index register, there is no additional operation for computing $i+1$. (Note that this does not mean the operation itself does not imply latency, but only that its movement does not add any extra latency.)

3 Inherent Properties of the Loops

Every specific execution of a loop with conditional branches is characterized by the number of iterations, and the outcomes of the branches (conditions) during the execution. It can be described with the DDG that incorporates all operations that have been executed according to the outcomes of the conditions, and dependencies between them. This 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 loop execution to another, due to various outcomes of conditions. In some boundary cases, the DDG of a loop can 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 can 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 the branches, we cannot *adjust* the target code to every single execution. All we can do is produce a code that is "near-optimal." 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 a *conditional operation* (or *condition* for short) and an *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 a *predicate*. A conditional operation is an operation that performs some computation, thus occupying some resources, and determines the actual value of the predicate. This operation is connected with other operations by data dependencies. This is a physical term, and can be treated as any other operation. Such a separation provides a better focus on the problem of defining the schedule of an iteration, determined by a combination of predicates.

We first have to decouple these two entities and adjust our code according to various values of the predicates. As a result of this adjustment, which incorporates operation movements guarded by the values of the predicates, we get some *predicated* code, like in the *if-conversion* [4, 29], but without introducing unnecessary data dependencies from control dependencies. 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 (where its own conditional operation was scheduled).

Two cases can occur then. First, the conditional operation can be computed before the moment its predicate is used for the first time, perhaps several iterations away. This case suggests the recording of the outcomes of several conditional operation instances from previous iterations. This feature can be implemented in hardware, as in Cydra 5 [18]. Second, the outcome can be used before its conditional operation has been computed. For example, one operation can be control-dependent on a predicate, and scheduled before the conditional operation that defines that predicate. This is the occurrence of speculative execution, which again can (but need not) be supported by hardware [21].

4 Sources of Parallelism Specific for Loops with Conditional Branches

This section summarizes some of the effects of outcomes on a loop DDG. These effects represent important sources of parallelism that every ambitious loop parallelization technique must exploit.

4.1 Disappearance of operations

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 4.

Control dependencies are represented by thick edges, with *True* denoted by the bold edge, and *False* by the hollow one. Data dependencies are marked with operation latencies expressed in clock cycles.

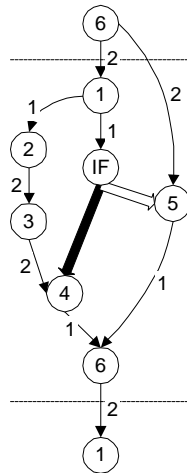


Figure 4: DDG of a loop iteration. Dotted lines represent the iteration boundaries

If the outcome is *True*, the operation 4 is executed, and the 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*, the operation 4 does not exist, and the operation 3 becomes "bottom-free." There is a dependency chain 6-5-6 of 3 cycles. As we will see later, if we allow speculative execution, this disappearance of the operation 4 allows execution of this iteration in 3 cycles. Therefore, we may conclude:

One effect of conditional operations and a source of parallelism is the "disappearance" of some operations from the DDG, as a consequence of their different outcomes.

4.2 Variable iteration distance

The second important effect is a *variable iteration distance*. Consider the loop in Figure 5.



Figure 5: An example of a variable iteration distance

In this example, the operation 2 depends on the operation 1 from the same iteration, if the condition `Test` is true. Otherwise, the operation 2 depends on the operation 1 from the previous iteration, if `Test` from that iteration was true, etc. As a result, there is a dependence of a variable iteration distance from the operation 1 to the operation 2, which can have all values starting from 0.

Figure 6a depicts the DDG of four adjacent iterations with the outcome *True* in the first, and *False* in the others. Figure 6b shows the same effect, but with a set of potential dependencies pointing to the operation 2, where only that one from the last executed operation 1 exists.

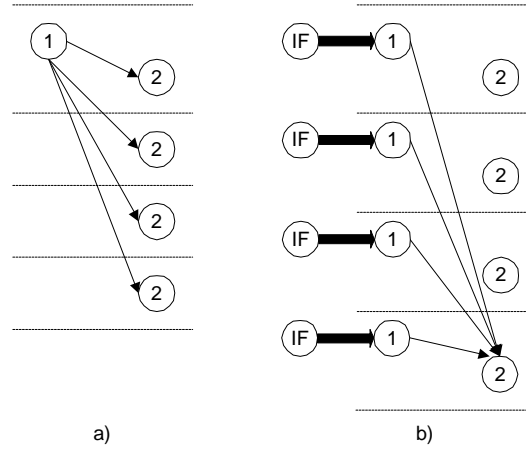


Figure 6: Variable iteration distance. Dotted lines represent iteration boundaries

This effect enables the later instances of the operation 2 to be moved upwards into the previous iterations (Figure 6a), while "stretching" the dependency edges that start from the moved operation 2. This stretching "spreads" the execution of the moved operation 2 over several iterations, causing shorter latencies of these subsequent iterations. (Moving later instances of the operation 2 upwards causes replication of instances of this operation in earlier iterations.)

Here we have the combined predicates that control the moved up operation, e.g. $op2[2]$: it is to be moved up only if $op1[0]=True$ and $op1[1]=False$ and $op1[2]=False$. Besides, if these conditional operations may be moved upwards and executed early enough, these predicates will be known when $op2$ is to be executed. Otherwise, we may execute the future instances of the operation 2 speculatively.

This is another source of parallelism. We therefore conclude:

Another effect of control dependencies is that under some conditions certain iteration distances of data dependencies are increased. This allows moving some operations upwards into previous iterations, while increasing the iteration distances of the dependencies that go out from these operations.

4.3 Replication of operation instances

Let us go back to the example in Figure 4. If the outcome is *False*, the operation 4 does not exist, and the operation 3 becomes bottom-free, so it can be moved downwards into the next iteration. However, there is a chain 6-5-6 that requires at least 3 cycles for the iteration. Consequently, the chain 1-2-3 should be adjusted to this latency of 3 cycles, to prevent an increase in the latency of the iteration. This can be done by moving the operation 3 into the next iteration, as in Figure 7a. This movement of the operation 3 does not affect the latency of the next iteration, since it also lasts for at least 3 cycles.

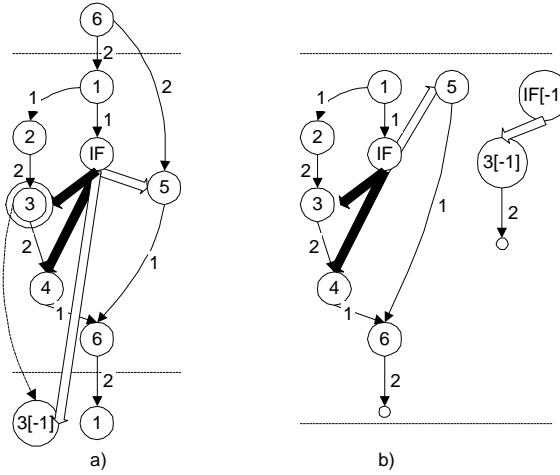


Figure 7: Operation movement

We may conclude that the operation 3 is moved into the next iteration only if the outcome is *False*. Thus, the operation 3 from the considered iteration exists in this iteration under the condition *True*, as shown in Figure 7a. Otherwise, this operation exists in the next iteration, as an instance of the operation 3 from the previous iteration, denoted with 3[-1]. Figure 7b 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 $IF[0] = True$. This is the graph of the example in Section 2, Figure 1, and the way we can achieve H of 8, or 3 cycles, as in Figure 3.

Two things can be noticed. First, each iteration can have from zero 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 the operation 3. Second, we can see that an instance can depend on an outcome of a condition from previous iteration, which raises the need for recording the outcomes from previous iterations. We may conclude:

In order to achieve parallelism, one iteration might have several instances of the same operation from different iterations, which are control-dependent on different instances (predicates) of the same or different conditional operations. All, some, or none of the operation instances can be executed, depending on the combinations of the outcomes.

4.4 Interference of conditions

The described effects point out that there is a significant interference of outcomes from different iterations. At the first sight, it seems that different conditional operations from the same iteration, or adjacent iterations, have the major impact. This is not always the case. Consider the example of a loop whose cyclic DDG is depicted in Figure 8. Dependencies are marked with the pairs (latency, iteration distance). Only predicates are shown.

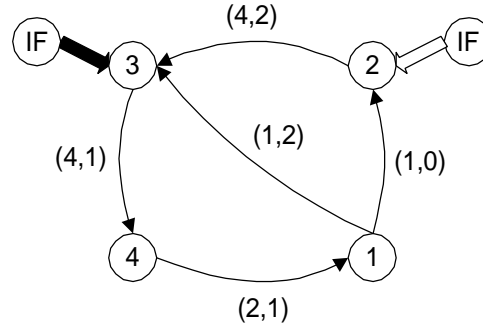


Figure 8: Cyclic DDG of a loop with a condition

Since the iteration distance of the edge 2-3 is 2, there is an important interference between the outcomes from iterations at that distance. Let the outcome of IF from the iteration $i+k$ be denoted by $p[k]$, where i is the considered iteration connected with the operation 3. The operation op from iteration $i+k$ is denoted similarly: $op[k]$. There are four cases:

- a) $p[0]=F, p[-2]=F$; the graph becomes the chain 4-1-2, because 3[0] does not exist;
- b) $p[0]=F, p[-2]=T$; the graph becomes the chain 4-1, because 3[0] and 2[-2] do not exist;
- c) $p[0]=T, p[-2]=F$; the graph is as shown, because both 3[0] and 2[-2] exist;
- d) $p[0]=T, p[-2]=T$; the graph becomes the cycle 4-1-3-4, because 2[-2] does not exist.

Finally, we conclude: *There are important interferences between those outcomes that are at the distance dictated by the data dependencies between the controlled operations, no matter whether these are the outcomes of instances of the same or different conditional operations from different iterations.*

This conclusion discourages every solution that is oriented towards loop unrolling, since loop unrolling treats all instances of all conditions equally: a loop is unrolled n times, and all conditions are represented with the same number n of instances.

5 Conclusion

We have provided a brief analysis of the problem of parallelizing loops with conditional branches in the context of software pipelining. Several important sources of parallelism have been identified. One of the most important is the need to allow several instances of the same operation from different initial iterations to exist in one transformed iteration. These instances are control-dependent on different instances of predicates. These conclusions have guided to the definitions of the *Predicated Software Pipelining* (PSP) loop execution model [13], the PSP scheduling technique [14], and the PSP intermediate representation and code generation algorithm [15].

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. on Programming Language Design and Implementation*, June 1988, pp. 308-317
- [3] Aiken, A., Nicolau, A., "Perfect Pipelining: A New Loop Optimization Technique," *Proc. 1988 European Symposium on Programming, Springer Verlag Lecture Notes in Computer Science*, #300, 1988, pp. 221-235
- [4] Allen, J. R., Kennedy, K., Porterfield, C., Warren, J., "Conversion of Control Dependence to Data Dependence," *Proc. 10th ACM Symp. Principles of Programming Languages*, 1983, pp. 177-189
- [5] Bacon, D. F., Graham, S. L., Sharp, O. J., "Compiler Transformations for High-Performance Computing," *ACM Computing Surveys*, Vol. 26, No. 4, December 1994, pp. 345-420

- [6] Ebcioglu, K., "A Compilation Technique for Software Pipelining of Loops With Conditional Jumps," *Proc. 20th Ann'l Workshop on Microprogramming (MICRO-20)*, December 1987, pp. 69-79
- [7] Ebcioglu, K., Nakatani, T., "A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture," *Second Workshop on Languages and Compilers for Parallel Computing*, 1989
- [8] Ferrante, J., Ottenstein, K. J., Warren, J. D., "The Program Dependence Graph and its Use in Optimization," *ACM Trans. Programming Languages and Systems*, Vol. 9, July 1987, pp. 310-349
- [9] Gasperoni, F., "Compilation Techniques for VLIW Architectures," Technical Report #435, New York University, Computer Science Dept., NY, USA, March 1989
- [10] Lam, M. S., Wilson, R. P., "Limits of Control Flow on Parallelism," *Proc. 19th Ann'l Int'l Symposium on Computer Architecture (ISCA-92)*, May 1992, *SIGARCH Comp. Arch. News*, Vol. 20, No. 2, pp.46-57
- [11] Lam, M., "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," *Proc. ACM SIGPLAN 1988 Conf. on Programming Language Design and Implementation*, June 1988, pp. 318-328
- [12] Lee, G., "Parallelizing Iterative Loops with Conditional Branching," *IEEE Trans. Parallel and Distributed Systems*, Vol. 6, No. 2, February 1995, pp. 185-189
- [13] 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
- [14] Milicev, D., Jovanovic, Z., "Predicated Software Pipelining Technique for Loops with Conditions," *Proc. 12th Int'l Parallel Processing Symp. (IPPS '98)*, 1998
- [15] Milicev, D., Jovanovic, Z., "Code Generation for Software Pipelined Loops with Conditions," Technical Report TI-ETF-RTI-99-041, University of Belgrade, Faculty of Electrical Engineering, July 1999
- [16] Moon, S.-M., Ebcioglu, K., "An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW processors," *Proc. 25th Ann'l Int'l Symp. on Microarchitecture (MICRO-25)*, December 1992, pp. 55-71
- [17] Nikolau, A., "Percolation Scheduling: A Parallel Compilation Technique," Technical Report TR-85-678, Cornell University, 1985
- [18] Rau, B. R., Yen, D. W. L., Yen, W., Towle, R. A., "The Cydra 5 Departmental Supercomputer," *IEEE Computer*, January 1989, pp. 12-35
- [19] Rau, B. R., "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops," *Proc. 27th Ann'l Int'l Symp. on Microarchitecture (MICRO-27)*, 1994, pp. 63-74
- [20] Rau, B., Fisher, J. A., "Instruction-level Parallel Processing: History, Overview, and Perspective," *Journal of Supercomputing*, Vol. 7, No. 1/2, May 1993, pp. 9-50
- [21] Rau, B. R., Schlansker, M. S., Tirumalai, P. P., "Code Generation Schema for Modulo Scheduled Loops," *Proc. 25th Ann'l Int'l Symp. on Microarchitecture (MICRO-25)*, 1992, pp. 158-169
- [22] Schwiegelshohn, U., Gasperoni, F., Ebcioglu, K., "On Optimal Parallelization of Arbitrary Loops," *Journal of Parallel and Distributed Computing* 11, 1991, pp. 130-134
- [23] Shim, S., Moon, S.-M., "Split-Path Enhanced Pipeline Scheduling for Loops with Control Flows," *Proc. 31st Ann'l Int'l Symp. on Microarchitecture (MICRO-31)*, 1998
- [24] Stoodley, M., Lee, C., "Software Pipelining Loops with Conditional Branches," *Proc. 29th Ann'l Int'l Symp. on Microarchitecture (MICRO-29)*, 1996
- [25] Su, B., Wang, J., "GURPR*: A New Global Software Pipelining Algorithm," *Proc. 24th Ann'l Workshop on Microprogramming and Microarchitecture (MICRO-24)*, November 1991, pp.212-216
- [26] 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. on Microarchitecture (MICRO-26)*, December 1993, pp. 21-29
- [27] Uht, A. K., "Requirements for Optimal Execution of Loops with Tests," *IEEE Trans. Parallel and Distributed Systems*, Vol. 3, No. 5, September 1992, pp. 573-581
- [28] 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. on Microarchitecture (MICRO-25)*, December 1992, pp.170-179
- [29] Warter, N. J., Mahlke, S. A., Hwu, W.-M. W., Rau, B. R., "Reverse If-Conversion," *Proc. ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation*, 1993, pp. 290-299
- [30] Warter-Perez, N. J., Partamian, N., "Modulo Scheduling with Multiple Initiation Intervals," *Proc. 28th Ann'l Int'l Symp. on Microarchitecture (MICRO-28)*, 1995, pp. 111-118