The Benefit of Predicated Execution for Software Pipelining

Nancy J. Warter Daniel M. Lavery Wen-mei W. Hwu

Center for Reliable and High-Performance Computing
University of Illinois at Urbana/Champaign
Urbana, IL 61801

Abstract

Software pipelining is a compile-time scheduling technique that overlaps successive loop iterations to expose operation-level parallelism. An important problem with the development of effective software pipelining algorithms is how to handle loops with conditional branches. Conditional branches increase the complexity and decrease the effectiveness of software pipelining algorithms by introducing many possible execution paths into the scheduling scope. This paper presents an empirical study of the importance of an architectural support, referred to as predicated execution, on the effectiveness of software pipelining. In order to perform an in-depth analysis, we focus on Rau's modulo scheduling algorithm for software pipelining. Three versions of the modulo scheduling algorithm, one with and two without predicated execution support, are implemented in a prototype compiler. Experiments based on important loops from numeric applications show that predicated execution support substantially improves the effectiveness of the modulo scheduling algorithm.

1 Introduction

Software pipelining has been shown to be an effective scheduling method for exploiting operation-level parallelism [1] [2] [3] [4]. The basic idea behind software pipelining is to overlap the iterations of a loop body and thus expose sufficient parallelism to utilize the underlying hardware. There are two fundamental problems that must be solved for software pipelining. The first is to ensure that overlapping lifetimes of the same virtual register are allocated to unique physical registers. The second problem is enabling loop iterations with conditional branches to be overlapped.

These problems have been solved in the compiler implementations for the Warp [5] [3] and Cydra 5 [6] machines. Both implementations are based on the modulo scheduling techniques proposed by Rau and Glaeser [7]. The Cydra 5 has special hardware support for modulo scheduling in the form of a rotating register file and predicated operations [8]. In the absence of special hardware support in the Warp, Lam proposes modulo variable expansion and hierarchical reduction for handling register allocation and conditional constructs respectively [3]. Rau has studied the benefits of hardware support for register allocation [9]. In this paper we analyze the implications and benefits of predicated operations for software pipelining loops with conditional branches. In particular, the analysis presented in this paper is designed to help future microprocessor designers to determine if predicated execution support is worthwhile given their own estimation of the increased hardware cost.

Although software pipelining has been shown to be effective for many different architectures, in this paper we limit the discussion to VLIW and superscalar processors. For clarity, we use VLIW terminology. Thus, an instruction refers to a very long instruction word which contains multiple operations.

2 Modulo Scheduling

In a modulo scheduled software pipeline, a loop iteration is initiated every *II* cycles, where *II* is the *Initiation Interval* [7] [3] [10] [11]. The *II* is constrained by the most heavily utilized resource and the worst case recurrence for the loop. These constraints each form a lower bound for *II*. A tight lower bound is the maximum of these lower bounds. In this paper, this tight lower bound is referred to as the *minimum II*. If an iteration uses a resource r for c_r cycles and there are n_r copies of this resource, then the minimum *II*

Algorithm	Loop Data Dependence Graph	Modulo Resource Reservation Table
1. Generate data dependence graph 2. Determine Iteration Interval (II) - maximum of resource constraints 3. List schedule using modulo resource reservation table	LD <f, 4> ADD <f, 1> MUL <f, 1> ST - 2 functional units - uniform FU's - II = 2	functional units cycle mod II: FU1, FU2 0: LD {s = 0}, ADD {s = 4} 1: MUL {s = 5}, ST {s = 7} s = start time

Figure 1: Modulo scheduling a loop without recurrences.

due to resource constraints, RII, is

$$RII = \max_{r \in R} \left\lceil \frac{c_r}{n_r} \right\rceil,$$

where R is the set of all resources. If a dependence edge, e, in a cycle has latency l_e and connects operations that are d_e iterations apart, then the minimum II due to dependence cycles, CII, is

$$CII = \max_{c \in C} \left\lceil \frac{\sum_{e \in E_c} l_e}{\sum_{e \in E_c} d_e} \right\rceil,$$

where C is the set of all dependence cycles and E_c is the set of edges in dependence cycle c.

Figure 1 shows how modulo scheduling is applied to a loop without recurrences. The loop has four operations with dependences shown in the data dependence graph. The arcs on the dependence graph show the type of the dependence (flow) and latency. In this example, a 2-issue VLIW with uniform function units is assumed. The load operation has a 4-cycle latency and the add and multiply have unit latencies. Since there are no recurrences, the minimum II depends on the maximum of the resource constraints. Since there are two uniform function units and four operations in the loop, the minimum II is $\lceil \frac{4}{2} \rceil = 2$. With no recurrences, the operations can be list scheduled using the modulo resource reservation table [12]. The modulo resource reservation table has a column for each function unit and a row for each cycle in II. Note that after the load, add, and multiply are scheduled, the store operation can be scheduled in cycle 6. However, there are no available resources at cycle 0 (6 mod 2) in the reservation table and thus the store operation is delayed a cycle and scheduled in cycle 7.

If a schedule is not found for a given II, then II is incremented and the loop body is rescheduled. This process repeats until an II is found that satisfies the resource and dependence constraints. If the loop does not have any recurrences, a schedule can usually be found for the minimum II [7]. In the presence of recurrences, heuristics were developed in the Cydra 5 compiler to generate near-optimal schedules. The basic principle is to schedule the recurrence node first and then the nodes not constrained by recurrences [12] [11].

After II is scheduled, the code for the software pipelined loop is generated. Since not every instruction is issued in the first II cycles, a *prologue* is required to "fill" the pipeline. There are $p = \left(\lceil \frac{latest\ issue\ time}{II} \rceil - 1 \right)$ II's or *stages* in the prologue. In the example in Figure 1, $p \left(= \lceil \frac{7+1}{2} \rceil - 1 \right) = 3$. The *kernel* of the software pipeline corresponds to the portion of the code that is iterated. Since the loop body spans multiple II's, a register lifetime may overlap itself. If this happens, the registers for each lifetime must be renamed. This can be done in hardware using a *rotating register file* as in the Cydra 5 [6]. With this support, the kernel has one stage. Without special hardware support, *modulo variable expansion* can be used to unroll the kernel and rename the registers [3]. The number of times the kernel is unrolled, u, is determined by the maximum register lifetime modulo II. The last part of the pipeline is the *epilogue* which consists of p stages needed to complete executing the last p iterations of the loop. In total, the software pipeline has $2p + u$ stages.

3 Scheduling Loops with Conditional Branches

In order to apply modulo scheduling to loops with conditional branches, the loop body must first be converted into straight-line code. Two techniques, *hierarchical reduction* [13] [3] and *if-conversion* [14] [15], have been developed for converting conditional constructs (e.g., if-then-else constructs) into straight-line code for the Warp [5] and Cydra 5 [6] machines respectively. In this section, we analyze the impact of these two conversion techniques on the scheduling constraints.

3.1 Hierarchical Reduction

Hierarchical reduction is a technique that converts code with conditional constructs into straight-line code by collapsing each conditional construct into a pseudo operation [13] [16] [3]. In this section, these

pseudo operations will be referred to as *reduct_op's*. It is a hierarchical technique since nested conditional constructs are reduced by collapsing from the innermost to the outermost. After hierarchical reduction, reduct_op's can be scheduled with other operations in the loop. Hierarchical reduction assumes no special hardware support. Thus, the conditional constructs are regenerated after modulo scheduling, and all operations that have been scheduled with a reduct_op are duplicated to both paths of the conditional construct.

A reduct_op is formed by first list scheduling both paths of the conditional construct. The resource usage of the reduct_op is determined by the union of the resource usages of both paths after list scheduling. The dependences between operations within the conditional construct and those outside are replaced by dependences between the reduct_op and the outside operations. A dependence between two operations is characterized by the type (flow, anti, output, and control), distance, and latency. The distance is the number of loop iterations the dependence spans. The latency is the minimum number of cycles that must be between the operations to guarantee the dependence is met. While the type and distance of the dependence does not change, the latency for a given dependence is modified to take into account when the original operation is scheduled with respect to the reduct_op. Consider two operations op_i and op_j where where op_j is an operation from the conditional construct that has been list scheduled at time t_j. A dependence arc with latency d, source op_i, and destination op_j, is replaced by a dependence arc between op_i and the reduct_op, op_r, with latency $d' = d - t_j$.[1] If instead, op_j is the source and op_i is the destination, the latency for the dependence between op_r and op_j is $d + t_j$.

Figure 2 shows a C code segment of a loop and the corresponding assembly code. Figures 3 and 4 show how the hierarchical reduction technique reduces the control construct in this code segment. The machine being scheduled in this example is a VLIW with two uniform function units. All operations have a one cycle latency. Figure 3 shows the dependence graph for the loop body. The dependence arcs are marked with the type, distance and latency. The types are abbreviated as follows: flow (f), anti (a), and control (c). Figure 4a shows how the reduction operation is formed by list scheduling the two paths *op5, op6, op7* and *op5, op8*. The resultant resource usage of the reduct_op *op5'* is also shown. Figure 4b shows the modified loop

[1] It is possible to have a dependence with a negative latency. After op_i is scheduled, op_r can be scheduled d' cycles earlier if there are no resource conflicts and all other dependences are satisfied.

```
for (i = 0; i < max; i++) {
   if(i == 0) {
      k = c1 * A[i];
   } else {
      k = c2 * A[i];
   }
   A[i] = k;
}
```

a: C code segment.

```
op1:          r1 <- 0
op2:          r2 <- label_A
op3:          r3 <- max*8
op4:    L1:   r4 <- mem[r2+r1]
op5:          bne r1, 0, L2
op6:          r5 <- r4 * c1
op7:          jump L3
op8:    L2:   r5 <- r4 * c2
op9:    L3:   mem[r2+r1] <- r5
op10:         r1 <- r1 + 8
op11:         bne r1, r3, L1
```

b: Assembly language segment

Figure 2: Example C loop segment with assembly corresponding assembly code.

dependence graph. Note that the latency for the flow dependence between operations *op4* and *op6* (and also between *op4* and *op8*) is originally one. These dependences are replaced by a flow dependence between *op4* and *op5'* with zero latency since *op6* and *op8* are scheduled one cycle later than *op5'*. Note that there are no control dependences after the graph has been reduced. This code can now be modulo scheduled. After scheduling, the reduct_op's must be expanded. Any operations scheduled within a reduct_op must be copied to both paths of the regenerated conditional construct.

While hierarchical reduction allows a loop with conditional constructs to be modulo scheduled, it places some artificial scheduling constraints on the loop by first list scheduling the operations of the conditional construct. The list schedule causes the reduct_op to have a complex resource usage which may conflict with already scheduled operations during modulo scheduling. Figure 4 illustrates the complex resource usage patterns formed by hierarchical reduction. In addition, if a reduct_op spans more than one II, it may conflict with itself and thus no schedule for that II can be found. Since a reduct_op can be overlapped with other reduct_op's, including itself, there is a potential for large code expansion. If a reduct_op is overlapped n times, there are 2^n possible execution paths. Other operations scheduled with the overlapping reduct_op must be copied to each of these paths.

3.2 If-conversion

If-conversion is another technique that can be used to convert loops with conditional constructs into straight-line code [14]. The basic concept behind if-conversion is to replace conditional branch operations with equivalent compare operations which set a flag.

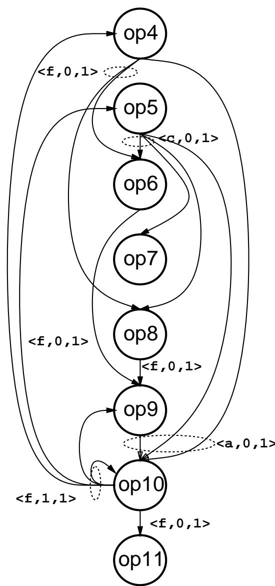


Figure 3: Dependence graph of example code segment.

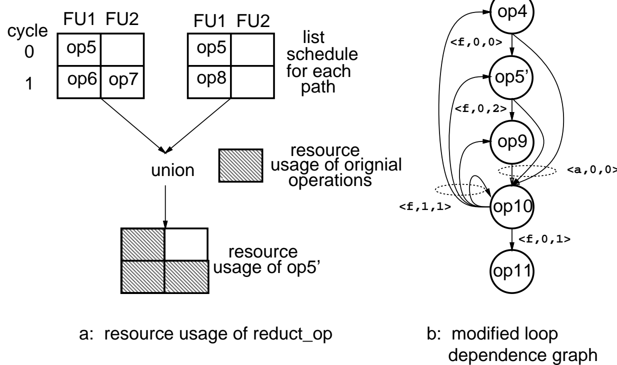
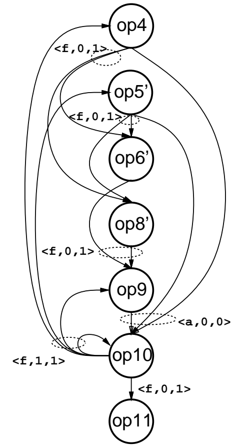


Figure 4: Applying hierarchical reduction to example code segment.

```
op1:         r1 <- 0
op2:         r2 <- label_A
op3:         r3 <- max*8
op4:   L1:   r4 <- mem[r2+r1]
op5':        pred_ne r1, 0, p1
op6':        r5 <- r4 * c1      <p1, F>
op8':        r5 <- r4 * c2      <p1, T>
op9:         mem[r2+r1] <- r5
op10:        r1 <- r1 + 8
op11:        bne r1, r3, L1
```



a: assembly code segment after if-conversion

b: modified loop dependence graph

Figure 5: Assembly code segment after if-conversion.

Operations that are control dependent on the conditional branch are converted into operations that only execute if the flag is set properly. In this way, control dependences are converted into data dependences [15]. To execute if-converted code, hardware support must be available for setting a conditional execution flag and to allow conditional execution of operations. In a vector processor, if-conversion is supported by a mask vector. In a superscalar or VLIW processor, if-conversion can be supported by predicated execution. The architecture support for predicated execution is presented in Section 4.

Figure 5a shows the assembly code after if-conversion for the example in Figure 2. A predicate has an *id* and *type*, represented as $< id, \; type >$, where type is either true or false. The predicate compare operation $op5'$ sets (clears) the predicate $<\mathbf{p1},\mathbf{T}>$ $(<\mathbf{p1},\mathbf{F}>)$ if $\mathbf{r1}$ is not equal to zero. Otherwise, it will clear (set) $<\mathbf{p1},\mathbf{T}>$ $(<\mathbf{p1},\mathbf{F}>)$. Operation $op6'$ executes if $<\mathbf{p1},\mathbf{F}>$ is set and operation $op8'$ executes if $<\mathbf{p1},\mathbf{T}>$ is set. Note that while only one of these operations executes, both will be fetched. Thus, with if-conversion, the resource constraints along both paths are summed. Therefore, if-conversion will usually require more resources than hierarchical reduction but operation placement is more flexible since these operations are not list scheduled prior to modulo scheduling. Also note that if-conversion removes the need for the jump operation $op7$. However, if a predicate compare operation is not guaranteed to be executed during the execution of the loop body, then the predicate must be invalidated at the beginning of the loop. In the example in Figure 5a,

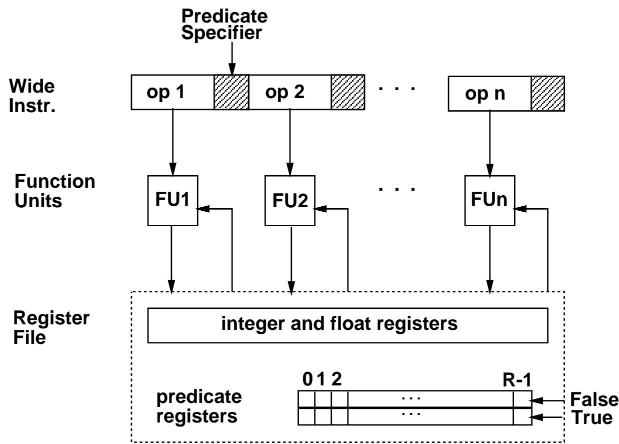


Figure 6: Architecture support for predicated execution.

the predicate compare operation *op5'* will always be executed and thus no predicate invalidate operation is required. The modified dependence graph in Figure 5b illustrates that the control dependences in Figure 3 have been converted into data dependences.

4 Architecture Support for Predicated Execution

The architecture support required for predicated execution consists of: 1) predicate compare operations, 2) predicate invalidate operations, 3) a predicate register file, and 4) predicated operations. An example of an architecture similar to the Cydra 5 is shown in Figure 6 [6] [8] [11]. There are 14 types of predicate compare operations: integer, single precision floating-point; double precision floating-point versions of equal, not equal, greater than, and greater than equal; and unsigned integer versions for greater than and greater than equal. These operations compare two source operands and set the value of the destination predicate register accordingly. A predicate invalidate operation is used to invalidate the specified predicate register.

The predicate register file has $\mathbf{R}$ 2-bit registers. The low-order bit corresponds to the predicate being false and the high-order bit corresponds to the predicate being true. Both bits should never be set at the same time. If both are cleared, the predicate is invalid. A predicate compare operation will set the low-order bit and clear the high-order bit if the result of the comparison is false. Likewise, it will set the high-order and will clear the low-order bit if the result is true. The predicate invalidate operation clears both bits of a predicate register. Each operation within the wide instruction word has a predicate register specifier of width $\log_2 \mathbf{R}+1$, where $\log_2 \mathbf{R}$ bits specify the predicate register and the remaining bit specifies the type (true or false). The width can be reduced if the predicate register file is implemented as a rotating register file as in the Cydra 5 [11].

All operations within the wide instruction are executed. After the predicate register file access delay, an operation in the execution pipeline that references a cleared predicate register will be squashed.

5 Compiler Support

Both hierarchical reduction and if-conversion, as well as modulo scheduling, have been implemented in the IMPACT C compiler. These techniques are applied to the appropriate loops after classical code optimizations have been performed [17] and after they have been translated into the target machine assembly code, but before register allocation. In our current implementation, we apply software pipelining to inner loops that do not have function calls or early exits from the loop.

After a loop is determined to be appropriate for software pipelining, either hierarchical reduction or if-conversion is applied to remove conditional constructs. Then the loop is modulo scheduled. Next, modulo variable expansion is applied to rename overlapping register lifetimes. The resultant kernel code is used to generate the prologue and epilogue where operations that are not issued in a given cycle are replaced by no-op's. If hierarchical reduction has been performed, the reduct_op's are expanded. Since there can be no early exits from the software pipeline[2], the loop must execute $p + k * u$ times, where p is the number of stages in the prologue, k is an integer greater than or equal to one, and u is the number of stages in the kernel determined by modulo variable expansion. A non-software pipelined version of the loop is required to execute the remaining number of iterations. If the loop trip count is greater than $p + u$, the remaining number of iterations is $(trip\ count - p) \bmod u$. If the trip count is less than $p + u$, only the non-software pipelined loop is executed. If the trip count is known to be less than $p + u$ at compile time, the software pipeline is not generated.

[2] Early exits require a special epilogue for each stage in the prologue and kernel which increases the code generation complexity and code expansion considerably.

```
          op1:          r1 <- -8   <------- (2)
          op2:          r2 <- label_A
          op3:          r3 <- max*8
        / op10:   L1:   r1 <- r1 + 8
       |  op4:          r4 <- mem[r2+r1]
       |  op5:          bne r1, 0, L2
   (1) |  op6:          r5 <- r4 * c1
       |  op7:          jump L3
       |  op8:    L2:   r5 <- r4 * c2
        \ op9:    L3:   mem[r2+r1] <- r5
          op11:         bne r1, r3, L1
```

Figure 7: Assembly code segment after induction variable reversal.

In the presence of recurrences, Dehnert et. al., discuss the importance of compiler optimizations such as back-substitution and load-store removal to reduce *CII* [8]. Another important optimization is needed to *remove* recurrences involving induction variables. When an induction variable is post-incremented (as in the example in Figure 2a), any operations that use the induction variable will cause a recurrence as shown in Figure 2b. This recurrence will force all of the operations in the cycle to be scheduled within one *II*. However, the recurrence can be removed by converting the induction variable to be pre-incremented. We refer to this optimization as *induction variable reversal*. To perform this optimization, the operation that increments the induction variable is moved to the beginning of the loop, and the initial value of the induction variable is decremented by the increment value.

The code after applying induction variable reversal to the assembly code segment in Figure 2[3] is shown in Figure 7. First *op10* is moved to the beginning of the loop and then 8 is subtracted from the initial value in *op1*. Note that there is still a recurrence due to *op10*. However, since only one operation is in the recurrence cycle, the cycle can always be scheduled. After this optimization, *op4*, *op5* and *op9* are no longer constrained to be scheduled within one *II*. When the induction variable spans more than one *II*, overlapping lifetimes are renamed using modulo variable expansion.

6 Experimental Results

In order to determine the benefits of predicated execution, we ran experiments for three forms of modulo scheduled loops: 1) with if-conversion, 2) with hierarchical reduction, and 3) with limited hierarchical reduction. The limited hierarchical reduction form does not allow any reduct_op to overlap itself (i.e., to span more than one *II*).

6.1 Machine Model

The machine model for these experiments is a VLIW processor with Cray-style interlocking. There are uniform resource constraints with the exception that only one branch can be issued per cycle. There are 14 branch operations (in order to be compatible with predicate compare operations). There are no branch delay slots. Other than the branch operation, we use the instruction set and operation latencies of the Intel i860[4]. Most integer operations take 1 cycle except for the integer load which takes 2 cycles. The integer multiply and divide and the floating-point divide are implemented using approximation algorithms [18]. The floating-point load, ALU, and single-precision multiply take 3 cycles, and the double precision multiply takes 4 cycles. For if-conversion, we assume the predicate execution support discussed in Section 4. The predicate compare operations have a one cycle latency. In order to measure the predicate register requirements, the size of the predicate register file is unlimited. The model has an infinite register file and we assume an ideal cache. The experiments were performed using machines with instruction widths or, equivalently, issue rates of 2, 4, and 8.

The base processor for these experiments is a RISC processor with an infinite register file and an ideal cache. The base schedule is a basic block schedule.

6.2 Benchmarks

To run our experiments, we collected a set of 28 loops with conditional branches from the Perfect and SPEC benchmarks. Since the focus of this study is to analyze the relative performance of conditional branch handling techniques, only *DOALL* loops (loops without cross-iteration memory dependencies) were included in the test suite.

6.3 Results

Performance

The speedup results presented are the harmonic mean of the speedup for each loop. We assume that the loop executes an infinite number of times. Thus, the effect of the prologue and epilogue are not accounted

[3] The code in Figure 2 has been optimized by the classical optimizations induction variable strength reduction and induction variable elimination [17].

[4] We assume that the load and floating-point pipelines are automatically advanced.

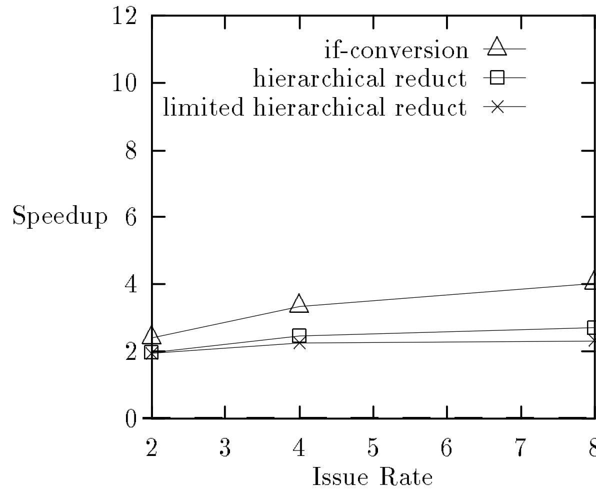


Figure 8: Speedup before induction variable reversal.

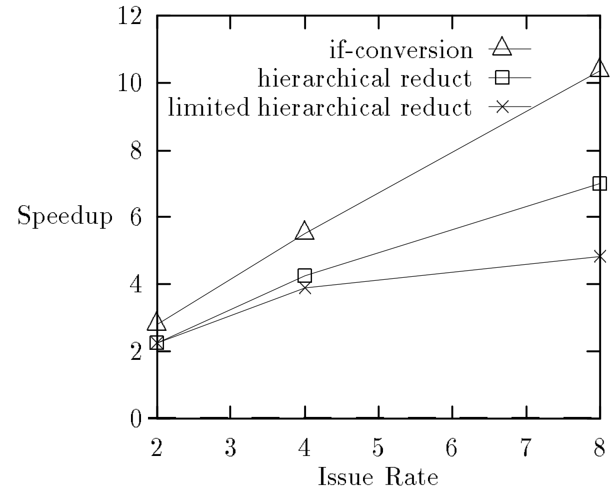


Figure 9: Speedup after induction variable reversal.

for. Figure 8 shows the performance of the three techniques before induction variable reversal. Comparing this graph with the graph of the performance after induction variable reversal shown in Figure 9, we see that the optimization improves the performance of hierarchical reduction by as much as 160% for an issue-8 machine and the performance of if-conversion by as much as 157% for an issue-8 machine.

In Figure 9 modulo scheduling with if-conversion performs approximately 25%, 29%, and 48% better than hierarchical reduction for issue rates 2, 4 and 8 respectively. For lower issue rates, hierarchical reduction performs worse since it is more difficult to schedule the reduct_op using fewer resources. The reason the relative performance increases for issue-8 is that the constraint of one branch per cycle becomes a limiting factor. For an issue-2 machine, limited hierarchical reduction has approximately the same performance as hierarchical reduction. This is due to the fact that the reduced resources increases II such that a conditional construct can usually be scheduled within one II. However, for issue 4 and 8, hierarchical reduction performs approximately 10% and 45% better than limited hierarchical reduction.

Code Expansion

Figure 10 shows the average code expansion of the three techniques. This is the code expansion for the software pipelined loop including the prologue, kernel, and epilogue[5]. It does not include the code expansion for the extra loop required to execute the remaining iterations that do not fit into the software pipeline. Figure 10 shows that hierarchical reduction has 5% and 27% more code expansion than if-conversion for a issue 2 and 4 respectively. As the issue rate increases, more conditional constructs overlap and thus the code expansion increases. For an issue-8 machine the code expansion for hierarchical reduction is approximatedly 78% greater than that for if-conversion. If the loop has a high trip count, the working set is the kernel code. In this case, if-conversion will have a much smaller working set than hierarchical reduction. Also, after if-conversion there is straight-line code which increases the data locality. While limited hierarchical reduction has comparable code expansion to if-conversion, it's performance is too poor to offset this benefit.

[5]Explicit prologue and epilogue code is not required with predicated execution. However, if they are generated, the compiler can overlap their execution with operations outside the loop.

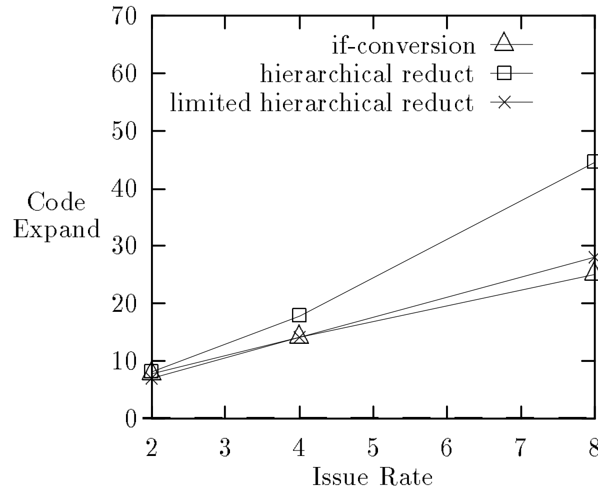


Figure 10: Code expansion.

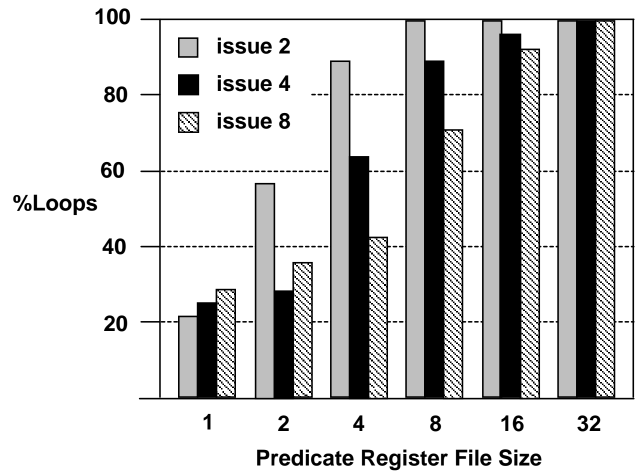


Figure 11: Number of predicated registers.

Predicate Register File Size

Figure 11 gives the percentage of the loops that can fit into the specified predicate register file size. As expected, the predicate register requirements increase as the issue rate increases and more iterations are overlapped. A predicate register file with R registers requires $\log_2 R + 1$ bits in the predicate register specifier. To schedule all of the loops for an issue-8 machine would require a 6-bit predicate register specifier. However, the majority of the loops scheduled have only one conditional construct per loop. Thus, if a rotating register file is used [8], a 1-bit predicate register specifier would be adequate.

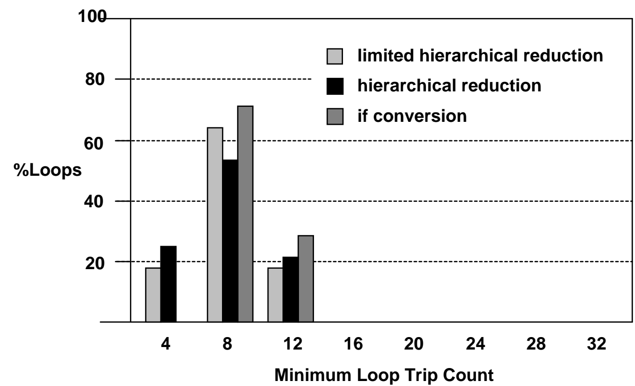


Figure 12: Minimum loop trip count for issue-2.

Minimum Loop Trip Count

The minimum loop trip count for a loop to be software pipelined is the number of stages in the prologue and epilogue. This metric can be used to determined the magnitude of the trip count required to see a benefit from software pipelining. Figures 12-14 show the distributions of the minimum loop trip count for issue 2, 4, and 8. The numbers on the x-axis represent the upper bound of a given range (e.g., 8 refers to the range 5-8). As expected, as the issue rate increases the minimum loop trip count increases because the II decreases, and thus the loop body is scheduled across more stages. For the same reason, the minimum trip count is the greatest for if-conversion.

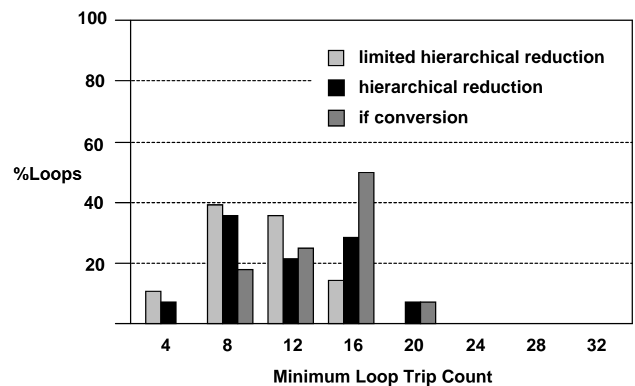


Figure 13: Minimum loop trip count for issue-4.

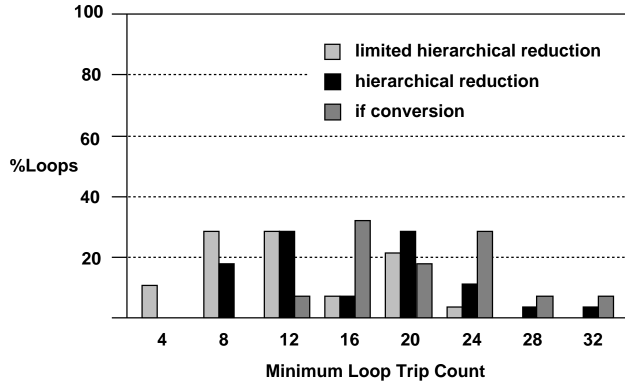


Figure 14: Minimum loop trip count for issue-8.

7 Related Work

Modulo scheduling schedules operations for a given iteration interval by delaying operations to eliminate resource conflicts [19]. Other global software pipelining techniques rely on global compaction where operations are scheduled as early as possible. Aiken and Nicolau's perfect pipelining technique compacts and unrolls the loop body until a pattern is reached [20]. Su and Wang have developed a similar technique, GURPR*, that does not require the computational complexity of determining a repeating pattern and that reduces the code expansion overhead [21]. Ebcioglu and Nakatani's enhanced pipeline scheduling also compacts the code by moving operations upward until an instruction is filled. Once an instruction is filled, it is moved across the backedge and is again available for scheduling. In this fashion, operations are scheduled across iteration boundaries enabling software pipelining [22].

Enhanced pipeline scheduling also requires special hardware support for a decision tree which allows operations from the same conditional constructs in different iterations of the loop to overlap. Jones and Allan showed that modulo scheduling performs slightly better than their implementation of enhanced pipeline scheduling for loops without conditional statements [23]. They also predicted that enhanced pipeline scheduling would perform better than Lam's implementation of modulo scheduling due to the constraints imposed by hierarchical reduction. The drawback to modulo scheduling with both if-conversion and hierarchical reduction is that the *II* remains constant for every iteration [24]. Thus, a loop that has a conditional construct with an infrequently executed path much longer than the frequently executed path will take longer to execute than techniques which have a variable *II* such as enhanced pipeline scheduling [25] [22].

Since predicated instructions remove the conditional branches, operations can be moved without worrying about code duplication to guarantee that the proper operations are executed along both paths of a branch. Perfect pipelining, enhanced pipeline scheduling, and GURPR* all require code duplication and thus do not have the code space efficiency afforded by scheduling with predicated execution.

8 Conclusion

In order to study the benefit of predicated execution for software pipelining, we have implemented hierarchical reduction, if-conversion and modulo scheduling in a prototype compiler. The implementation effort allows us to gain insight into the complexity and constraints imposed by conditional branches on the modulo scheduling technique.

With this implementation, we are able to quantify the impact of architectural support for predicated execution on the performance of loops with conditional branches. For our benchmarks, software pipelined loops with predicated execution support execute on the average 34% faster than without the support. The primary reason for this improvement is that the pre-pass list scheduling in the hierarchical reduction method creates pseudo operations with complex resource usage patterns. These pseudo operations limit the effectiveness of software pipelining for many loops. On the other hand, the if-conversion method does not need pre-pass list scheduling. This increased flexibility accounts for the superior performance results with predicated execution support.

Acknowledgements

The authors would like to acknowledge all members of the IMPACT research group for their support, particularly Scott Mahlke, David Lin, and William Chen for their contributions of compiler support for predicated execution. Special thanks to the anonymous referees whose comments and suggestions helped to improve the quality of this paper significantly.

This research has been supported by the Joint Services Engineering Programs (JSEP) under Contract N00014-90-J-1270, Dr. Lee Hoevel at NCR, the AMD 29K Advanced Processor Development Division, Matsushita Electric Industrial Corporation Ltd., Hewlett-Packard, and the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-

613 in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS).

References

[1] A. Charlesworth, "An approach to scientific array processing: The architectural design of the AP-120B/FPS-164 family," in *IEEE Computer*, September 1981.

[2] R. Touzeau, "A Fortran compiler for the FPS-164 Scientific Computer," in *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, June 1984.

[3] M. S. Lam, "Software pipelining: An effective scheduling technique for VLIW machines," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 318–328, June 1988.

[4] R. L. Lee, A. Kwok, and F. Briggs, "The floating point performance of a superscalar SPARC processor," in *Proceedings of the 4th International Conference on Architecture Support for Programming Languages and Operating Systems*, pp. 28–37, April 1989.

[5] M. Annaratone, E. Amould, T. Gross, H. Kung, M. Lam, O. Menzilcioglu, and J. Webb, "The Warp compiler: Architecture, implementation and performance," *IEEE Transactions on Computers*, vol. 12, December 1987.

[6] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *IEEE Computer*, pp. 12–35, January 1989.

[7] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," in *Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture*, pp. 183–198, October 1981.

[8] J. C. Dehnert, P. Y. Hsu, and J. P. Bratt, "Overlapped loop support in the Cydra 5," in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 26–38, April 1989.

[9] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker, "Register allocation for software pipelined loops," in *Proceedings of the ACM SIGPLAN 92 Conference on Programming Language Design and Implementation*, pp. 283–299, June 1992.

[10] C. Eisenbeis, "Optimization of horizontal microcode generation for loop structures," in *International Conference on Supercomputing*, pp. 453–465, July 1988.

[11] P. Tirumalai, M. Lee, and M. Schlansker, "Parallelization of loops with exits on pipelined architectures," in *Supercomputing*, November 1990.

[12] P. Y. T. Hsu, *Highly Concurrent Scalar Processing*. PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1986.

[13] G. Wood, "Global optimization of microprograms through modular control constructs," in *Proceedings of the 12th Annual Workshop on Microprogramming*, pp. 1–6, 1979.

[14] R. Towle, *Control and Data Dependence for Program Transformations*. PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1976.

[15] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pp. 177–189, January 1983.

[16] M. Lam, *A Systolic Array Optimizing Compiler*. PhD thesis, Carnegie Mellon University, Pittsburg, PA, 1987.

[17] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.

[18] Intel, *i860 64-Bit Microprocessor*. Santa Clara, CA, 1989.

[19] J. H. Patel and E. S. Davidson, "Improving the throughput of a pipeline by insertion of delays," in *Proceedings of the 3rd International Symposium on Computer Architecture*, pp. 159–164, 1976.

[20] A. Aiken and A. Nicolau, "Optimal loop parallelization," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pp. 308–317, June 1988.

[21] B. Su and J. Wang, "GURPR*: A new global software pipelining algorithm," in *Proceedings of the 24th Annual Workshop on Microprogramming and Microarchitecture*, pp. 212–216, November 1991.

[22] K. Ebcioglu and T. Nakatani, "A new compilation technique for parallelizing loops with unpredictable branches on a VLIW architecture," in *Second Workshop on Languages and Compilers for Parallel Computing*, August 1989.

[23] R. B. Jones and V. H. Allan, "Software pipelining: An evaluation of Enhanced Pipelining," in *Proceedings of the 24th International Workshop on Microprogramming and Microarchitecture*, pp. 82–92, November 1991.

[24] F. Gasperoni, "Compilation techniques for VLIW architectures," Tech. Rep. 66741, IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY 10598, August 1989.

[25] K. Ebcioglu, "A compilation technique for software pipelining of loops with conditional jumps," in *Proceedings of the 20th Annual Workshop on Microprogramming and Microarchitecture*, pp. 69–79, December 1987.