The Partial Reverse If-Conversion Framework for Balancing Control Flow and Predication

David I. August Wen-mei W. Hwu

Center for Reliable and High-Performance Computing
University of Illinois
Urbana-Champaign, IL 61801
Email: {august, hwu}@crhc.uiuc.edu

Scott A. Mahlke

Hewlett-Packard Laboratories
Hewlett-Packard
Palo Alto, CA 94304
Email: mahlke@hpl.hp.com

June 30, 1999

Abstract

Predicated execution is a promising architectural feature for exploiting instruction-level parallelism in the presence of control flow. Compiling for predicated execution involves converting program control flow into conditional, or predicated, instructions. This process is known as if-conversion. In order to apply if-conversion effectively, one must address two major issues: what should be if-converted and when the if-conversion should be performed. A compiler's use of predication as a representation is most effective when large amounts of code are if-converted and when if-conversion is performed early in the compilation procedure. On the other hand, efficient execution of code generated for a processor with predicated execution requires a delicate balance between control flow and predication. The appropriate balance is tightly coupled with scheduling decisions and detailed processor characteristics. This paper presents a compilation framework based on partial reverse if-conversion *that allows the compiler to maximize the benefits of predication as a compiler representation while delaying the final balancing of control flow and predication to schedule time.*

Key words: control flow, if-conversion, instruction scheduling, predicated execution, reverse if-conversion

1 Introduction

The performance of modern processors depends on their ability to execute multiple instructions per cycle, requiring increasing levels of Instruction-Level Parallelism (ILP) to be exposed in programs. One of the major challenges to increasing the available ILP is overcoming the limitations imposed by branch instructions.

ILP is limited by branches for several reasons. First, branches impose control dependences which often sequentialize the execution of surrounding instructions. Second, the uncertainty of branch outcomes forces compiler and hardware schedulers to make conservative decisions. Branch prediction is generally employed along with speculative execution to overcome these limitations [1][2], but branch misprediction detracts significantly from the potential performance gain. Third, traditional techniques, constrained to work with branching control structure, only facilitate exploiting ILP along a single trajectory of control, although the concurrent execution of instructions from multiple trajectories could dramatically increase ILP. Finally, branches often interfere with or complicate aggressive compiler transformations, such as optimization and scheduling.

Predication is a model in which instruction execution conditions are not solely determined by branches. This characteristic allows predication to form the basis for many techniques which deal with control effectively in both the compilation and execution of codes. It provides benefits both in compilers as a representation and in ILP processors as an architectural feature.

The *predicated representation* is a compiler N-address program representation in which each instruction is guarded by a Boolean source operand whose value, as supplied from a register, determines whether the instruction is executed or nullified. This guarding Boolean source operand is referred to as the *predicate*. The values of predicate registers can be manipulated by a predefined set of predicate define instructions.

The predicated representation provides a useful and efficient model for compiler optimization and scheduling. The use of predicates to guard instruction execution can reduce or even completely eliminate the need for branch control dependences, since when all instructions control dependent on a branch are predicated using the same condition as the branch, that branch can legally be removed. Control dependences between branches and other instructions are thus converted into data dependences between predicate computation instructions and predicated instructions. In the predicated representation, then, control flow transformations can then be performed as traditional data flow optimizations. In the same way, the predicated representation allows scheduling among branches to be performed as a simple reordering of sequential instructions. Removal of control dependences increases scheduling scope and affords new freedom to the scheduler [3]. The process of replacing branches with appropriate predicate computations and guards is known as *if-conversion* [4][5].

Predicated execution is an architectural model which supports direct execution of the predicated representation [6][7][8]. With respect to a conventional instruction set architecture, the new features are an additional Boolean source operand guarding each instruction and a set of compare instructions used to compute predicates. Predicated execution benefits directly from the advantages of compilation using the predicated representation. In addition, the removal of branches yields performance benefits in the executed code, the most notable of which is the elimination of branch misprediction penalties. In particular, the removal of frequently mispredicted branches yields large performance gains [9][10][11]. Predicated execution also provides an efficient mechanism by which a compiler can overlap the execution of multiple control paths on the hardware. In this manner, processor performance may be increased by exploiting ILP across multiple program paths. Another, more subtle, benefit of predicated execution is that it allows control height reduction along important program paths [12].

Supporting predicated execution introduces two central issues for the compiler: what should be if-converted and when in the compilation procedure if-conversion should be applied. The first question to address is what should be if-converted or, more specifically, which branches should be removed via if-conversion. If-conversion increases ILP by combining multiple paths of control into a single path of conditional instructions. Execution of the combined path exhibits higher resource utilization than execution of the paths individually. Furthermore, inclusion of incompatible paths together in a hyperblock can result in degraded optimization or execution. If the compiler produces a path whose execution requirements exceed those available or in which the included paths interfere, performance can be negatively impacted. Traditionally, full if-conversion, in which all branches are removed from inner loop bodies, has yielded positive results for compilation of numerical applications [13]. In compilation of non-numerical codes which typically exhibit more complicated control flow, the central focus of this work, if-conversion must be applied much more selectively if performance is to be improved [14].

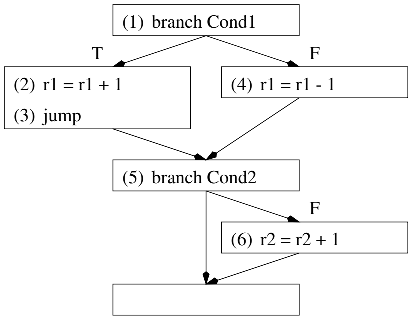
The second question that must be addressed is when in the compilation procedure to apply if-conversion. In the two extremes, if-conversion may be applied very early in the back-end compilation procedure or may be delayed to occur in conjunction with scheduling. Applying if-conversion early enables the compiler to make full use of the predicated representation, facilitating enhanced ILP optimizations and scheduling. In addition, this approach allows complex control flow transformations to be recast into the data dependence domain to make them practical and profitable. On the other hand, delaying if-conversion to as late as possible makes answering the first question much more practical. Since many of the if-conversion decisions are tightly coupled to the scheduler and its knowledge of target processor characteristics, it seems most natural to apply if-conversion at schedule time. Delaying if-conversion until scheduling also alleviates the need to make the entire compiler back-end cognizant of the predicated representation, reducing the effort necessary to accommodate some form of predication support into a traditional compiler.

An effective compiler strategy for supporting predicated execution must address the "what" and "when" questions of if-conversion. The purpose of this paper is to present a flexible framework for if-conversion in ILP compilers. The framework enables the compiler to extract the full benefits of the predicated representation by applying aggressive if-conversion early in the compilation procedure. A novel mechanism called *partial reverse if-conversion* then operates at schedule time to balance the amount of control flow and predication present in the generated code, based on the characteristics of the target processor.

The remainder of this paper is organized as follows: Section 2 discusses the characteristics of good and bad predicated code and details the issues and challenges associated with compiling for predicated execution. Section 3 introduces our proposed compilation framework which takes full advantage of the predicated representation and which achieves an efficient balance between branching and predication in the final code. The essential component in this framework, partial reverse if-conversion, is described in detail in Section 4. The effectiveness of this framework in the context of our prototype compiler for ILP processors is demonstrated in Section 5.

2 Challenges of predicated compilation

Effective use of predicated execution presents a difficult challenge for ILP compilers. Although predication offers the potential for large performance gains when efficiently utilized, an imbalance of predication and control flow in the generated code can lead to dramatic performance losses. The determining factor is whether the compiler can select the appropriate sets of paths for inclusion in predicated regions. The flexibility of the *hyperblock* makes it a natural choice for the basic unit of predicated code used in this work. The *hyperblock* is a structure created to facilitate optimization and scheduling



(a)

1	(1) p1, $\overline{p2}$ = Cond1	(5) p3, $\overline{p4}$ = Cond2	
2	(2) r1 = r1 + 1 <p1>	(4) r1 = r1 - 1 <p2>	(6) r2 = r2 + 1 <p4>

(b)

Figure 1: Hyperblock formation of consecutive if-then-else constructs.

for predicated architectures [14]. A hyperblock is a set of predicated basic blocks in which control may enter only from the top, but may exit from one or more locations. Hyperblocks are formed by applying tail duplication and if-conversion over a selected set of paths.

This section begins with a characterization of hyperblock quality, which provides valuable insight into the selection of an appropriate hyperblock formation mechanism and compiler phase ordering. Next discussed is the motivation for performing if-conversion early in the compilation process. The section concludes with an exposition of the problems that arise in developing a heuristic for performing early (pre-schedule-time) hyperblock formation. The desire to extract high levels of ILP through the effective use of both predicated execution and the predicated representation despite the complications presented in this section motivates the predication compilation framework presented in the remainder of the paper.

The principles developed in the following section are demonstrated using specifically designed, hand-crafted code examples. For clarity, these examples are necessarily smaller than hyperblocks typically created during compilation. The ideas illustrated, however, are culled from years of experience compiling real benchmarks in a hyperblock compilation framework. Unless noted otherwise, all schedules are for a three-issue, uniform functional unit machine with unit latencies.

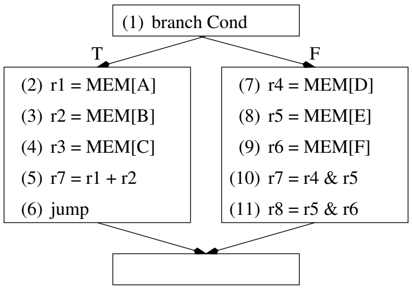
2.1 Beneficial path inclusion

Building good hyperblocks entails selecting the appropriate execution paths for inclusion. As might be expected, the scope of hyperblock-based optimizations and the benefit to be derived from run-time effects of predication increase with hyperblock size, as long as included paths are well selected. The following example demonstrates some of the characteristics of a well-formed hyperblock.

Figure 1 demonstrates the if-conversion of two consecutive if-then-else constructs, or hammocks. Figure 1a shows a very simple control flow graph before hyperblock formation. It consists of six instructions, three of which are branches. The instructions in Figure 1a are given numbers which uniquely identify them and which indicate in what order the code would be laid out. The static schedule for this code on a traditional machine cannot have a height of less than four cycles due to its dependence height. Parallel execution of this code segment is limited solely by control dependences. One subset of these control dependencies, which sets a lower bound on the dependence height, consists of the control dependences $1 \rightarrow 2$, $2 \rightarrow 5$, and $5 \rightarrow 6$. Other properties worth noting in the original code are related to the number and types of instructions. There is a total of six instructions: two conditional branches, one unconditional branch, and three computation instructions. The preponderance of branches in this code segment has a parasitic effect on performance, the severity of which depends on the branch-handling capabilities of the target architecture.

Figure 1b shows the same code segment after hyperblock formation including all paths. Conditional branches 1 and 5 have been converted to predicate define instructions. Branch 3 has been completely removed by combining the four original control flow paths into a single path. Elimination of branch 3 reduces the raw instruction count of this code from six to five. The removal of all branches eliminates their corresponding control dependencies. While the resulting hyperblock contains no control dependences, it does contain new data dependencies $1 \rightarrow 2$, $1 \rightarrow 4$, and $5 \rightarrow 6$ created by the sourcing and sinking of predicate registers. However, these data dependences are more desirable than the original control dependencies in that they do not force the resultant schedule to consume more than two cycles. In addition, the new data dependences do not prohibit the overlapping of consecutive hammocks as the control dependences did. This ability to overlap non-complementary paths can be a great source of ILP. Assuming the three-issue machine described earlier, neither the dependence height nor the resource subscription force the schedule to more than two cycles. The net result is a code schedule that cuts the execution time in half even assuming perfect branch prediction.

The benefits of the resulting hyperblock do not stop at the static code schedule. The removal of two conditional branches eliminates their mispredictions and associated misprediction penalties. Any taken branch penalties which may have existed are also removed. The hyperblock's straight-line code segment can be a great improvement over the disjoint blocks of the original code for instruction fetch mechanisms.



(a)

1	(1) p1, $\overline{p2}$ = Cond	(2) r1 = MEM[A]	(7) r4 = MEM[D]
2	(3) r2 = MEM[B] <p1>	(8) r5 = MEM[E] <p2>	(4) r3 = MEM[C] <p1>
3	(9) r6 = MEM[F] <p2>	(5) r7 = r1 + r2 <p1>	(10) r7 = r4 & r5 <p2>
4	(11) r8 = r5 & r6 <p2>		

(b)

Figure 2: Hyperblock formation saturating processor resources.

2.2 Harmful path inclusion

As suggested by the previous section, the benefits of hyperblock formation increase with the inclusion of more paths. However, for practical machines, compilers, and code characteristics there are many constraints on how large hyperblocks can be made without reducing their effectiveness.

One reason including a path can be detrimental to good hyperblock performance is that it causes excessive resource consumption. Figure 2 illustrates this problem. Figure 2a is a single hammock with two control flow paths controlled by branch instruction 1. The taken path consists of five instructions and has a dependence height of 2 cycles. The fall-through path also consists of five instructions and a dependence height of two. Since the machine model assumed for the examples has uniform functional units, instruction count can be an accurate measure of the subscription to machine resources. Scheduling the code of Figure 2a for this machine model results in each path fitting into two cycles.

Figure 2b shows the hyperblock formed by the if-conversion of these paths together. As expected, the resulting number of instructions is roughly equal to the sum of the instruction counts of the original paths. Scheduling for the three-issue machine model shows that this hyperblock requires four cycles for completion, compared with the two cycles needed for the original code. This doesn't take into account the branch and cache penalties eliminated by if-conversion. However, in some machines these benefits would be stripped by the increase in schedule height. Certainly, in actual code where path instruction counts number in the hundreds, resource over-subscription has the potential to negate all benefits of hyperblock compilation.

The problem of predicting the resource consumption due to including a path is not easily solved. In our uniform functional unit machine, the expected resource height is simply the sum of all instructions in each path included divided by the issue width after accounting for the removal of branches and creation of predicate define instructions. However, in practical machines, the types and number of instructions which can be issued together complicate this calculation.

The example of Figure 2 clearly illustrates that resources available in the target machine must be considered when choosing whether or not to include a path. While resource height is important, it is not the only consideration. Dependence height can also cause harmful path interaction. Consider a simple hammock of two paths, one with a dependence height of two cycles and the other with a dependence height of eight cycles. A hyperblock which contains both paths cannot complete until all of its constituent paths have completed. Therefore, the overall height of the hyperblock must be the maximum of all the original path's dependence heights. This means that every time the short path would have been executed in two cycles it is now executed in eight cycles–four times slower–resulting in poor hyperblock performance.

The problem is more complicated than solving for resource consumption and dependence height independently in the hyperblock formation decision process. There can exist many interactions between dependence height and resource consumption in a code schedule. For instance, the number of instructions available for issue in the first cycle may be much greater than the number of instructions available for issue later in the schedule due to flow dependencies among the instructions. This is the case in Figure 6, as will be discussed later. Considering this issue alone suggests that hyperblock formation decisions may belong in the scheduling phase.

2.3 Phase ordering considerations

Having established a concept of what make good and bad paths for inclusion in hyperblocks, or the "what to if-convert" question, we now turn our attention to the other question, of when it is most profitable to perform if-conversion in an ILP compiler. Although the previous section concluded with a strong justification for delaying hyperblock formation until schedule time, there are many compelling reasons for wanting to incorporate predication as early as possible in the compiler back-end. The benefits of performing significant optimization subsequent to if-conversion fall into two general categories. The first category consists of optimizations which could be performed in a non-predicated environment, but which can be effected much more naturally and with less compiler complexity in the predicated domain. The second category consists of optimizations which take particular advantage of the features of predicated execution, and which cannot be implemented without it.

Applying if-conversion to create hyperblocks early enables the full use of the predicated representation by the compiler to facilitate ILP optimizations and scheduling. The formation of relatively large hyperblocks early in the compilation process allows many local transformations to have global scope. Consider the local scheduling of a hyperblock for example, in which instructions from various paths can be scheduled freely with respect to each other. To achieve the same effect in a non-predicated environment, a global scheduler would have to be employed to move code among the constituent basic blocks. The formation of hyperblocks early in the compilation process also enables local data flow optimizations to optimize across dependences which were converted from control dependences. This can have a powerful effect on the quality of code generated.

Furthermore, when traditional data flow optimizations are applied to predicate defines themselves, powerful control optimizations can result. For example, common sub-expression elimination when applied to predicate defines can have the same effect as branch merging. Put another way, branch merging and many other control flow transformations can be performed by simple data flow transformations which already exist in most compiler infrastructures. The need to implement optimizations which perform control flow transformations can be eliminated. While these benefits of early hyperblock formation are important in simplifying compiler design, a compiler could be built which performs all the necessary optimizations to achieve the same level of code quality.

Finally, there exists a large and growing body of optimizations whose effectiveness depends on being able to generate a predicated result, or which have no practical control flow equivalent. The predicated representation allows complex control flow transformations to be recast into the data dependence domain, making them practical and profitable. Examples of such transformations include branch reordering, control height reduction [12], and branch combining [15], and optimization of program decision logic [16]. Some outcomes of these and other optimizations can only be represented effectively to the hardware using predication. The introduction of predication after most optimizations are complete would make these optimizations impossible or at least would reduce their effectiveness. Work continues in exploring the full potential of predication and the predicated representation and we expect the case for early hyperblock formation continue to strengthen.

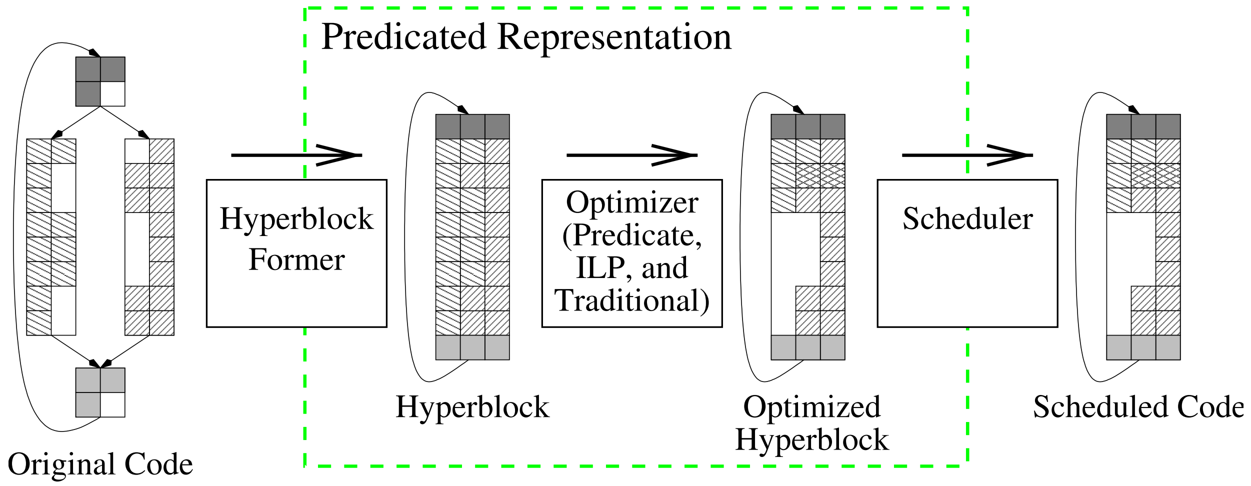
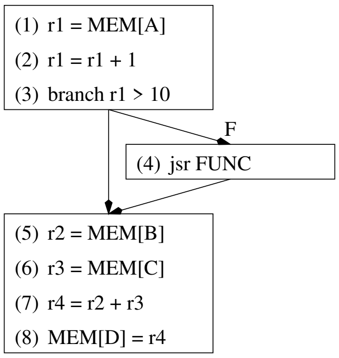


Figure 3: Early heuristic hyperblock formation estimates final code characteristics. Optimization changes a good hyperblock decision into a poor one.

2.4 The Hyperblock Compilation Framework

Having examined the rationale for performing if-conversion early, we finally turn to a consideration of the means by which paths could be selected for inclusion in hyperblocks. Balancing control flow and predication in hyperblocks is a relatively easy process when performed after all other code transformations are complete. However, to take full advantage of predication, optimization and scheduling passes must be performed on the hyperblocks. The following discusses various mechanisms for selecting traces, and problems with performing hyperblock path selection before schedule time.

The original approach used in the IMPACT compiler to support predicated execution is called the *Hyperblock Compilation Framework* and is shown in Figure 3. In this framework hyperblocks are formed using a path selection heuristic in an early compilation phase. The early phase if-conversion exposes the predicated representation throughout all the back-end compilation phases. Inclusion of a particular path into a hyperblock is done by considering its profitability. Profitability is determined by five pieces of information: resource utilization, dependence height, hazard presence, execution frequency, and branch behavior. Ideally, a hyperblock contains paths which together fully utilize machine resources, which are balanced in height, and which do not contain hazards. Further, in a model hyperblock most instructions are not nullified often and only well-behaved branches remain. Heuristic hyperblock formation has been shown to perform well for relatively regular machine models [15]. In these machines, resource consumption, dependence height, and hazards are managed effectively by the carefully crafted heuristics.



(a)

1	(1) r1 = MEM[A]		
2	(2) r1 = r1 + 1		
3	(3) p1, $\overline{\text{p2}}$ = r1 > 10		
4	(4) jsr FUNC <p2>	(5) r2 = MEM[B]	(6) r3 = MEM[C]
5	(7) r4 = r2 + r3		
6	(8) MEM[D] = r4		

(b)

1	(1) r1 = MEM[A]	(5) r2 = MEM[B]	(6) r3 = MEM[C]
2	(2) r1 = r1 + 1	(7) r4 = r2 + r3	
3	(3) branch r1 > 10	(8) MEM[D] = r4	

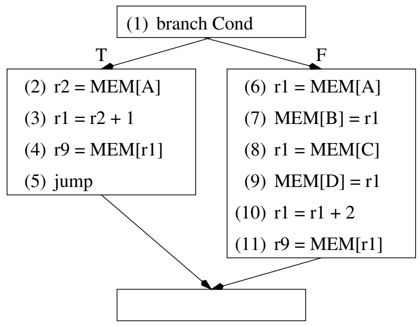
(c)

Figure 4: Hyperblock formation in the presence of hazards.

Hyperblock formation is a complex decision making process which requires a carefully crafted heuristic. Clearly, the heuristic must be sophisticated enough to consider the resource utilization, dependence height, hazard presence, and execution frequency for each path. Additionally, the heuristic must understand how an included path will interact with other included paths. For example, the inclusion of two paths which share a branch allows for that branch's removal, so the heuristic must also consider the run-time behavior of shared branches in candidate path sets. Path selection is further complicated by the fact that the inclusion or exclusion of particular paths can have a dramatic influence on the effectiveness of further optimizations and scheduling, as well as on how well the code after optimization will utilize the target machine's resources.

Over-subscription to resources and imbalances in dependence height are the two most common causes of poor hyperblock performance. However, other less obvious factors play a role in a hyperblock's effectiveness. One such factor is the presence of hazards. A hazard is any instruction or set of instructions which hinders efficient scheduling of paths other than its own. Since hazards affect other paths by definition, their negative effects can be eliminated by keeping them separate from other paths. This can be done by excluding them through tail duplication.

Figure 4a shows the control flow graph of a code segment which contains a hazard. Since the fall-through path contains one instruction and has a dependence height of one, it seems that including it would not adversely affect the hyperblock's dependence height and resource consumption characteristics. Figure 4b shows the hyperblock formed by including this fall-through path. In this schedule, it is assumed that the jsr has an unknown pointer store. Since the called procedure could potentially store to locations B or C, memory dependences $4 \rightarrow 5$ and $4 \rightarrow 6$ need to be created and respected. The result of this hyperblock schedule is that six cycles are required regardless of branch 3's direction. Figure 4c shows a



(a)

1	(1) p1, $\overline{\text{p2}}$ = Cond	(2,6) r2 = MEM[A]	(8) r3 = MEM[C]
2	(3) r1 = r2 + 1 <p1>	(7) MEM[B] = r2 <p2>	(10) r1 = r3 + 2 <p2>
3	(4,11) r9 = MEM[r1]	(9) MEM[D] = r3 <p2>	

(b)

Figure 5: Hyperblock formation of seemingly incompatible paths with positive results due to code transformations. The T and F annotations in (a) indicate the taken and fall-through path for the conditional branch. r2 is not referenced outside the T block.

hyperblock formed by excluding the fall-through path. Here the schedule is only three cycles when the branch is not taken. The destination of instruction 3 is a block containing instruction 4 as well as a copy, or tail duplication, of instructions 5 through 8. This tail duplication makes it possible to obtain the short schedule on the fall-through path. It is important to note that the path containing the jsr still has a six cycle schedule, three cycles in the Figure 4c hyperblock and three cycles in the tail-duplicated code.

As demonstrated, identifying and excluding hazards is very important to resulting code performance. Unfortunately, identifying hazards is a difficult problem. Consider the same example if we had assumed that all of instruction 4's memory operations were known to be independent with respect to instructions 5 and 6. In that case, the resulting hyperblock would have been four cycles for both paths. Clearly, the jsr is only a hazard when it creates dependences between itself and instructions in other paths. Hazards can take the form of instructions referencing registers which cannot be renamed or of ambiguous stores. The type and number of hazards is often a function of the capabilities of the compiler. For example, the level of memory disambiguation in a compiler determines which store instructions are hazards to efficient code generation.

2.5 Challenges posed by post-if-conversion optimization

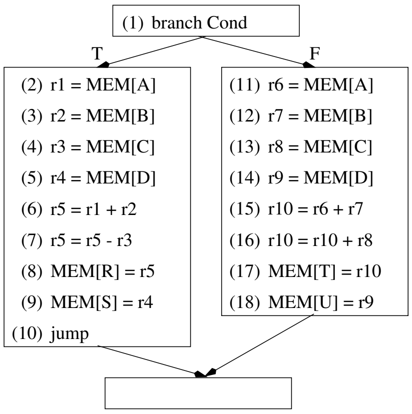
While it might seem reasonable that a heuristic could be designed to take all these effects into account, the following example will show that optimization subsequent to hyperblock formation may render early heuristic-based selection decisions incorrect.

Figure 5a shows another simple hammock to be considered for if-conversion. The taken path consists of a dependence height of two and a resource consumption of three instructions after if-conversion. The fall-through path consists of a dependence height of six and a resource consumption of six instructions. A simple estimate would indicate that combining these paths together would result in a four-cycle penalty for the taken path due to the fall-through path's long dependence height. Figure 5b shows this code segment after hyperblock formation and further optimizations. The first optimization performed is renaming to eliminate the false dependences $7 \rightarrow 8$ and $8 \rightarrow 10$. This reduces the dependence height of the hyperblock to three cycles.

If a heuristic could foresee that dependence height would no longer be an issue, it might still choose not to form this hyperblock due to its level of resource consumption. Based on an inspection of Figure 5a, a heuristic might estimate a hyperblock size of ten instructions. Unfortunately, execution of ten instructions requires at least four cycles on a three-issue machine, penalizing the taken path by one cycle. Thus a heuristic might decide that the combination of these paths is not beneficial. After an instruction merging optimization in which instructions 2 and 6 are combined and 4 and 11 are combined, the instruction count becomes eight. The final schedule takes only three cycles, a win for both paths when the control dependence from instruction 1 is considered.

The example of Figure 5 shows that in even simple cases a heuristic which forms hyperblocks before some optimizations must anticipate the effectiveness of those optimizations in order to form hyperblocks effectively. In this example, some optimizations, such as renaming, could have been done before hyperblock formation. Others like operation merging, however, can be performed only after hyperblock formation. In addition, some optimizations may be done differently because of different trade-offs which would be made in the context of the different code characteristics.

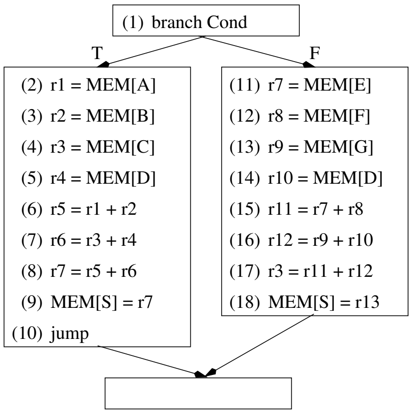
Anticipating exactly how effective future optimizations would be is an intractable problem, but perhaps empirical data could be used to estimate the aggregate effectiveness of optimization. Using this information, the hyperblock former might be able to make intelligent decisions. Assume that empirical data correctly suggests that the resource consumption of the resulting hyperblock will be 66% of the original, and that its dependence height will remain unchanged. Under these conditions the generated estimates may still be misleading, as the example of Figure 6 demonstrates. Figure 6a consists of two paths each with a dependence height of four. The resource consumption, or equivalently for this machine, the instruction count, is 18 instructions. Using the predicted optimization effects, the resulting hyperblock should contain 12 instructions and have a dependence height of four cycles, for a total cycle count of four cycles. Indeed, Figure 6b shows



(a)

1	(2,11) r1 = MEM[A]	(3,12) r2 = MEM[B]	(4,13) r3 = MEM[C]
2	(1) p1, $\overline{p2}$ = Cond	(5,14) r4 = MEM[D]	(6,15) r5 = r1 + r2
3	(7) r5 = r5 - r3	(16) r10 = r5 + r8	(9) MEM[S] = r4 <p1>
4	(8) MEM[R] = r5 <p1>	(17) MEM[T] = r10 <p2>	(18) MEM[U] = r4 <p2>

(b)

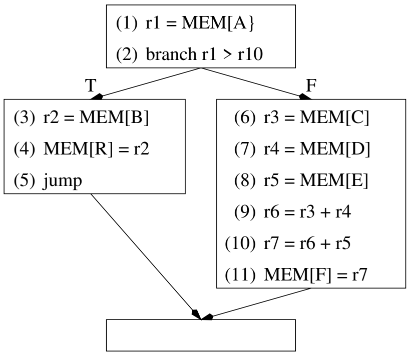


(c)

1	(1) p1, $\overline{p2}$ = Cond	(2) r1 = MEM[A]	(3) r2 = MEM[B]
2	(11) r1 = MEM[E] <p2>	(12) r2 = MEM[F] <p2>	(4) r3 = MEM[C] <p1>
3	(6,15) r5 = r1 + r2	(13) r3 = MEM[G] <p2>	(5,14) r4 = MEM[D]
4	(7,16) r6 = r3 + r4		
5	(8,17) r7 = r5 + r6		
6	(9,18) MEM[S] = r7		

(d)

Figure 6: Hyperblock formation of seemingly compatible paths with negative results.



(a)

1	(1) r1 = MEM[A]	(3) r2 = MEM[B]	(6) r3 = MEM[C]
2	(2') p1, $\overline{p2}$ = r1 > r10	(7) r4 = MEM[D]	(8) r5 = MEM[E]
3	(4) MEM[R] = r2 <p1>	(9) r6 = r3 + r4 <p2>	(2'') jump <p2>

(b)

1	(10) r7 = r6 + r5		
2	(11) MEM[F] = r7		

(c)

Figure 7: An efficient hyperblock formed through the inclusion of a partial path.

that the optimized code meets these expectations. Unfortunately, predicting the outcome of optimization in this way is inaccurate in the majority of cases. Consider the code segment shown in Figure 6c. Like the code in Figure 6a, this segment of code also has two paths with dependence heights of four and an instruction count of 18 instructions. Figure 6d shows this code after optimization. Here the predicted dependence height and resource count was obtained. However, the resulting schedule height is six cycles, 50% higher than the estimate. The discrepancy between the predicated schedule and the actual schedule is caused by the destructive interaction between dependences and resources. This effect demonstrates that to be useful the exact outcome of further optimization must be predicted not just at a high level, but in a very accurate and precise manner. Due to the complex interactions of different optimizations, the only reliable way to predict the exact outcome of optimizations is actually to perform them. Since these optimizations need to be applied to each candidate path selection set for each hyperblock, the optimization anticipation method is either cost-prohibitive or too inaccurate to be useful.

The challenge of forming efficient hyperblocks does not end there. Further complications result when we consider the fact that including part of a path may sometimes be more beneficial than including or excluding that entire path. This gives a hyperblock formation heuristic many more possibilities to consider. It also gives it the responsibility to be accurate at the instruction level, not just the path level. Taken together, this further complicates the already seemingly impossible task it must perform. If-conversion which can include parts of paths is referred to as *partial if-conversion*. Partial if-conversion is generally effective when the resource consumption or dependence height of an entire candidate path is too large to permit profitable if-conversion, but there is a performance gain to be had by overlapping a part of the candidate path with the other paths selected for inclusion in the hyperblock.

Figure 7 shows an example in which partial path inclusion results in better code than full inclusion or exclusion. Figure 7a shows two paths which are otherwise incompatible. However, by including all of the taken path and 4 instructions from the fall-through path an efficient hyperblock is created. This hyperblock is shown in Figure 7b. Notice that branch instruction 2 has been split into two instructions: the condition computation, labeled $2'$, and a branch based on that computation, labeled $2''$. The schedule did not benefit from the complete removal of branch instruction 2, as the branch instruction $2''$ has the same characteristics of the original. However, the schedule did benefit from the extra speculation drawn from both paths. The destination of branch instruction $2''$ is shown in Figure 7c.

The splitting of branch instruction 2 into a condition computation and a predicated jump has some interesting side effects. In this small example, the condition computation and branch have not been moved very far apart; however, in typical hyperblocks which are much larger these instructions may be scheduled many cycles apart. An architecture which can identify this situation may be able to eliminate the need to make a prediction on instruction $2''$ if the result of $2'$ has already been computed. Furthermore, one can reasonably expect that predicated architectures will not have a single instruction compare-and-branch of the form of instruction 2, but will only support separate predicate compares and predicated jumps of the form of instructions $2'$ and $2''$. Other ramifications of having only predicated jumps perform conditional branches are explored in [17].

3 Compilation frameworks

Section 2 described the challenges of compiling effectively for predicated execution. Efficient code must exhibit a delicate balance between control flow and predication. This balance is highly dependent on final code characteristics and the resource characteristics of the target processor. An effective compilation framework for predicated execution must provide a means of making intelligent tradeoffs between control flow and predication so the desired balance can be achieved. The best time to create an accurate balance is after all code transformations have been performed. An over-simplistic solution to this problem is to apply if-conversion at schedule time, after all other optimizations have been performed. While this phase ordering can use predication in a beneficial way, performance will fall short of predication's full potential.

As described in Section 2, if-conversion must be done before optimization in order to take full advantage of the predicated representation and of predicate optimizations. In the IMPACT compiler this has been accomplished in the context of the *Hyperblock Compilation Framework*, as described in Section 2.4. The remainder of this section examines two frame-

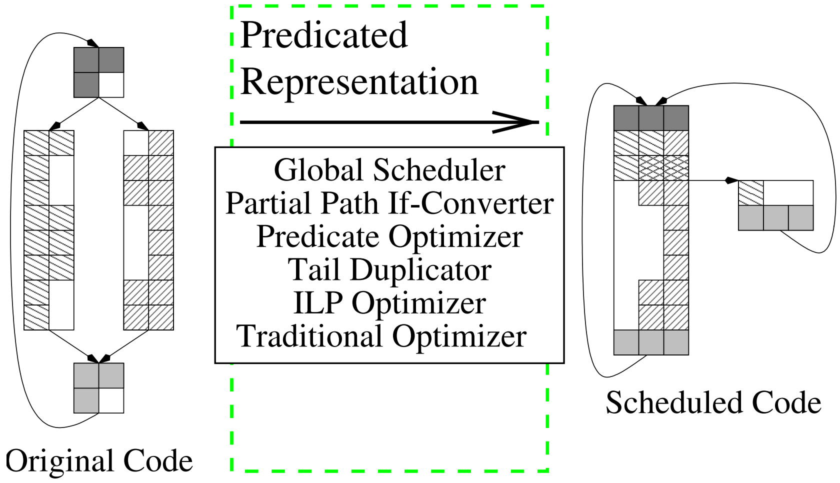


Figure 8: The If-Conversion During Scheduling Framework.

works, the *If-Conversion During Scheduling Framework* and the *Partial Reverse If-Conversion Framework*, which attempt to reconcile the need to perform optimization after if-conversion with the need to form carefully selected regions. The *Partial Reverse If-Conversion Framework*, the subject of this paper, is successful because it is designed to capture the desirable properties of both other frameworks.

3.1 The If-Conversion During Scheduling Framework

Given the difficulties with forming hyperblocks early in the back-end compilation process, a seemingly natural strategy is to perform if-conversion in conjunction with instruction scheduling. This can be achieved by integrating if-conversion into the scheduling process itself. The scheduler not only accurately models the detailed resource constraints of the processor but also understands the performance characteristics of the code. Therefore, the scheduler is ideally suited to making intelligent if-conversion decisions. In addition, all compiler optimizations are usually complete when scheduling is reached, thus the post-if-conversion path transformation problem is alleviated.

Delaying if-conversion until scheduling, however, prevents the compiler from using the predicated representation to perform control flow transformations or predicate-specific optimizations as shown in Section 2. One solution to this problem, the *If-Conversion During Scheduling Framework* shown in Figure 8, delays the optimizations which require predication to schedule time as well. Unfortunately, delaying only some optimizations until schedule time creates a problematic compiler phase ordering. Generally, most transforms have profound effects on one another and must be repeatedly applied in turn

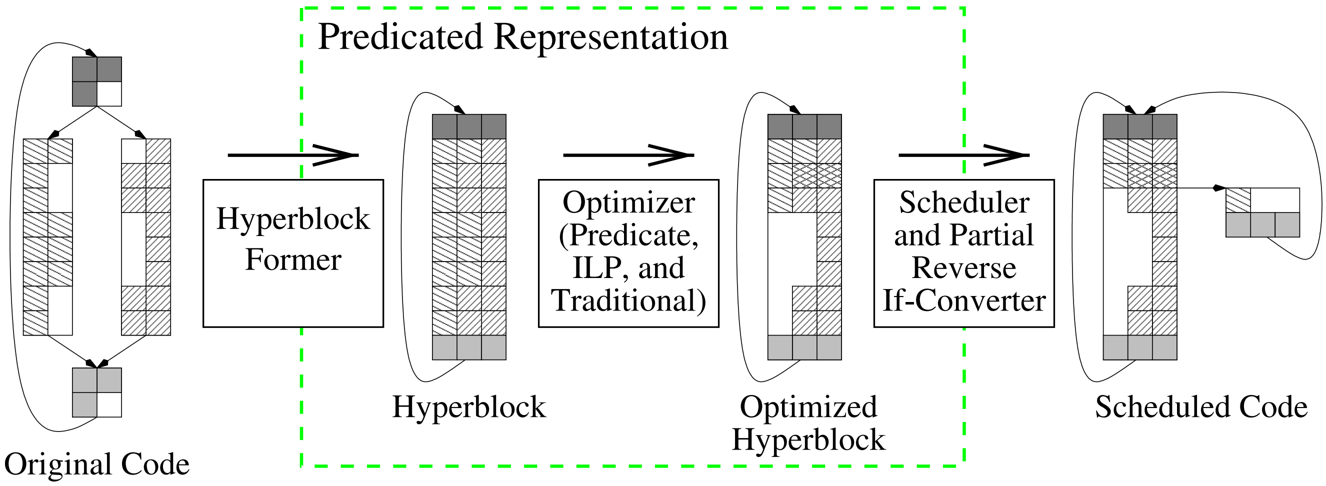


Figure 9: The Partial Reverse If-Conversion Predication Framework.

to achieve desirable results. For example, one predicate-based transformation, such as control height reduction [12], may subsequently expose a critical data dependence edge that should be broken by expression reformulation. Unless the control dependence height is reduced, however, the potential instruction-count cost of applying expression reformulation combined with the apparent lack of any benefit will prevent the compiler from applying the transform. Thus, for if-conversion during scheduling to yield code of quality comparable to that produced in an early formation scheme, traditional optimizations must be performed along with if-conversion and predicate-specific optimizations, in the scheduler.

For this framework to generate high-quality predicated code, the scheduler must simultaneously perform global scheduling, partial path if-conversion, predicate optimizations, tail duplication, ILP optimizations, and traditional optimizations. While studies have exhibited the possibility and profitability of performing optimizations at schedule time [18], in the *If-Conversion During Scheduling Framework* much more complexity is added to an ILP scheduler which must already consider many issues, including control speculation, data speculation, and register pressure. That this additional complexity is tolerable in a commercial instruction scheduler is highly doubtful, but for such a framework to produce quality code it is absolutely necessary.

3.2 The Proposed Compilation Framework

Given that performing if-conversion at schedule time limits the use of the predicated representation for optimization and given that if-conversion at an early stage is limited in its ability to estimate final code characteristics, it is logical to look to an alternative compilation framework. This paper presents such a framework. This framework overcomes limitations of other schemes by utilizing two phases of predicated code manipulation to support predicated execution. Aggressive if-conversion is applied in an early compilation phase to create the predicate representation and to allow flexible application of predicate optimizations throughout the back-end compilation procedure. Then, at schedule time, the compiler adjusts the final amount of predication to match the target architecture. The *Partial Reverse If-Conversion Predication Framework*, shown in Figure 9, consists of two phases of predicate manipulation surrounding classical, predicate specific, and ILP optimizations. The first predicate manipulation phase, hyperblock formation, has been addressed thoroughly in [14]. The second predicate manipulation phase, the adjustment of hyperblocks during scheduling termed *partial reverse if-conversion*, was proposed in [19], and is discussed in further detail here.

The first phase of the compilation framework consists of performing aggressive hyperblock formation. The hyperblock former does not need to compute exactly what paths, or parts of paths, will fit in the available resources and be completely compatible with each other. Instead, it forms hyperblocks which are larger than the target architecture can handle. The large hyperblocks increase the scope for optimization and scheduling, further enhancing the benefit derived from the predicated representation. In many cases, the hyperblock former will include almost all the paths. This is generally an aggressive decision because the resource height or dependence height of the resulting hyperblock is likely to be much greater than the corresponding heights of any of its component paths. However, the if-converter relies on later compilation phases to ensure that this hyperblock is efficient. One criterion that is still enforced in the first phase of hyperblock formation is that paths containing hazards are excluded. As was discussed in Section 2, hazards reduce the compiler's effectiveness for the entire hyperblock; thus, they should be avoided to facilitate more aggressive optimization.

The second phase of the compilation framework consists of adjusting the amount of predicated code in each hyperblock as the code is scheduled via partial reverse if-conversion. Partial reverse if-conversion is conceptually the application of reverse if-conversion to a particular predicate in a hyperblock for a chosen set of instructions [20]. Reverse if-conversion was originally proposed as the inverse process to if-conversion. In that process, branching code that contains no predicates is generated from a block of predicated code. This allows code to be compiled using a predicated representation, but to be executed on a processor without support for predicated execution.

The scheduler with partial reverse if-conversion operates by identifying the paths composing a hyperblock. Paths which overlap profitably remain unchanged. Conversely, a path that interacts poorly with the other paths is removed from the hyperblock. In particular, the partial reverse if-converter decides to eject certain paths, or parts of paths, to enhance the schedule. To do this, the reverse if-converter inserts a branch that is taken whenever the removed paths would have been executed. This has the effect of dividing the lower portion of the hyperblock into two parts, corresponding to the taken and fall-through paths of the inserted branch. The decision to reverse if-convert a particular path consists of three steps: First, the partial reverse if-converter determines the savings in execution time to be obtained by inserting control flow and applying the full resources of the machine to two hyperblocks instead of only one. Second, it computes the loss created by any penalty associated with the insertion of the branch. Finally, if the gain of the reverse if-conversion exceeds the cost, it is applied. Partial reverse if-conversion may be applied repeatedly to the same hyperblock until the resulting code is desirable.

A very important point is that regardless of the best possible balance of predication and control flow, the code has benefited from originally being predicated. Even if the partial reverse if-converter decides to completely restore control flow, the resulting code benefits from the freedom provided to optimizations and scheduling [20]. For example, portions of all paths exist and execute in parallel in various parts of the hyperblock due to the freedom provided to the scheduler.

The strategy used for this compilation framework can be viewed as analogous to the use of virtual registers in many compilers. With virtual registers, program variables are promoted from memory to reside in an infinite space of virtual registers early in the compilation procedure. The virtual register domain provides a more effective internal representation than do memory operations for compiler transformations. As a result, the compiler is able to perform more effective optimization and scheduling on the virtual register code. Then, at schedule time, virtual registers are assigned to a limited set of physical registers and memory operations are reintroduced as spill code when the number of physical registers was over-subscribed. The framework presented in this paper does for branches what virtual registers do for program variables. Branches are removed to provide a more effective internal representation for compiler transformations. At schedule time, branches are inserted according to the capabilities of the target processor. The branches reinserted have different conditions, targets, and predictabilities than the branches originally removed. The result is that the branches in the code are there for the benefit of performance for a particular processor, rather than as a consequence of code structure decisions made by the programmer.

The key to making this predication and control flow balancing framework effective is the partial reverse if-converter.

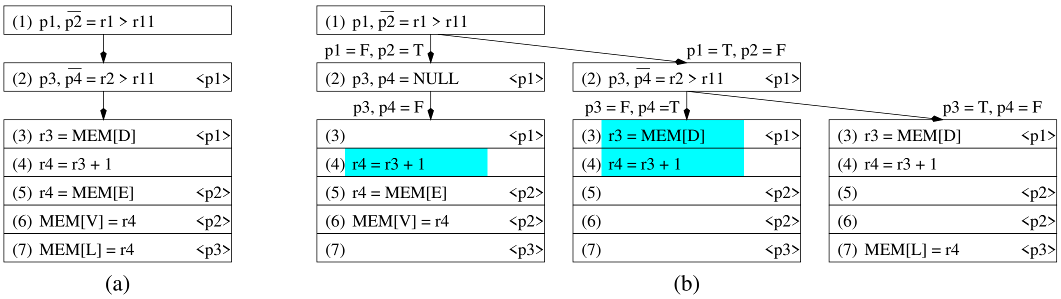


Figure 10: Predicate flow graph with partial dead code elimination given that r3 and r4 are not live out of this region.

The mechanics of performing partial reverse if-conversion, as well as a proposed policy used to guide partial reverse if-conversion, are presented in the next section.

4 Partial Reverse If-Conversion

The partial reverse if-conversion process consists of three components: analysis, transformation, and decision. These components are integrated into an instruction scheduler in the partial reverse if-conversion compilation framework. This section consists of a description of these components and their integration into a hyperblock scheduler. The section is concluded with a realistic code example to illustrate the operation of the framework.

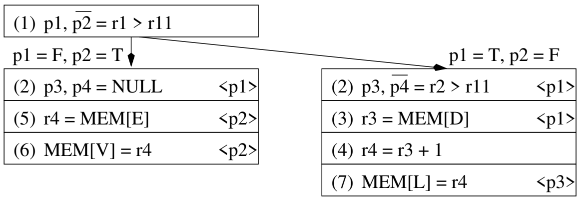
4.1 Analysis

Before any manipulation or analysis of execution paths can be performed, these paths must be identified in the predicated code. Execution paths in predicated code are referred to as predicate paths. Immediately after hyperblock formation, the structure of the predicate paths is identical to the control flow graph of the code before hyperblock formation. The structure of the predicate paths can be represented in a form called the *predicate flow graph* (PFG). The predicate flow graph is simply a control flow graph (CFG) in which predicate execution paths are also represented. After optimizations, the structure of the PFG can change dramatically. For reasons of efficiency and complexity, the compiler used in this work does not maintain the PFG across optimizations, instead it is generated from the resulting predicated N-address code.

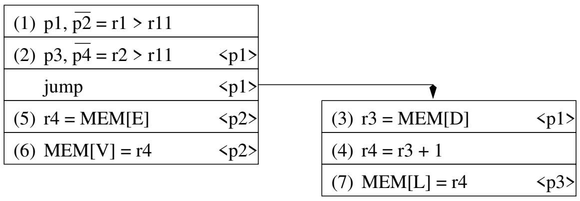
The synthesis of a PFG from predicated N-address code is analogous to creating a CFG from N-address code. A simple example is presented to provide some insight into how this is done. Figure 10 shows a predicated code segment and its predicate flow graph. The predicate flow graph shown in Figure 10b is created in the following manner. The first instruction in Figure 10a is a predicate definition. At this definition, p1 can assume TRUE or FALSE. A path is created for each of these possibilities. The complement of p1, p2, shares these paths because it does not independently create new conditional outcomes. The predicate define instruction 2 also creates another path. In this case, the predicates p3 and p4 can only be TRUE if p1 is TRUE because their defining instructions is predicated on p1, so only one more path is created. The creation of paths is determined by the interrelations of predicates, which are provided by mechanisms addressed in other work [21][22]. For the rest of the instructions, the paths that contain these instructions are determined by the predicate guarding their execution. For example, instruction 3 is based on predicate p1 and is therefore only placed in paths where p1 is TRUE. Instruction 4 is not predicated and therefore exists in all paths. The type of predicate defines used in all figures in this paper are unconditional, meaning they always write a value [8]. Since they write some value regardless of their predicate, their predicate can be ignored, and the instruction's destinations must be placed in all paths.

Paths in a PFG can be merged when a predicate is no longer used and does not affect any other predicate later in the code. However, this merging of paths may not be sufficient to solve all potential path explosion problems in the PFG. This is because the number of paths in a PFG grows exponentially with respect to the number of independent predicates whose live ranges overlap. Fortunately, in practice this never becomes prohibitively expensive except during code scheduling. After code scheduling, a complete PFG will have a large number of paths and may be costly. A description of how the partial reverse if-converter overcomes this problem is located in Section 4.2. A more general solution to the path explosion problem for other aspects of predicated code analysis is currently being constructed by the authors.

With a PFG, the compiler has the information necessary to know which instructions exist in which paths. In Figure 10, if the path in which p1 and p3 are TRUE is to be extracted, the instructions which would be placed into this path would be 3, 4 and 7. The instructions that remain in the other two paths seem to be 3, 4, 5, and 6. However, inspection of the dataflow characteristics of these remaining paths reveals that the results of instructions 3 and 4 are not used, given that r3 and r4 are not live out of this region. This fact makes these instructions dead code in the context of these paths. Performing traditional dead code removal on the PFG, instead of the CFG, determines which parts of these operations are dead. Since this application of dead code removal indicates that these instructions are dead only under certain predicate conditions, this process is termed *predicate partial dead code removal* and is related to other types of partial dead code removal [23]. The result of partial dead code removal indicates that instructions 3 and 4 would generate correct code and would not execute



(a)



(b)

Figure 11: Predicate flow graph (a) and a partial reverse if-conversion of predicate p1 located after instructions 1 and 2 (b).

unnecessarily if they were predicated on p3.

At this point, all the paths have been identified and unnecessary code has been removed by partial dead code removal. The analysis and possible ejection of these paths now becomes possible.

4.2 Transformation

Once predicate analysis and partial dead code elimination have been completed, performing reverse if-conversion at any point and for any predicate requires a small amount of additional processing. This processing determines whether each instruction belongs in the original hyperblock, the new block formed by reverse if-conversion, or both. Figure 11 is used to aid this discussion.

The partial reverse if-converted code can be subdivided into three code segments. These are: the code before the reverse if-converting branch, the code ejected from the hyperblock by reverse if-conversion, and the code which remains in the hyperblock below the reverse if-converting branch. Instructions before the location of the partial reverse if-converting branch are left untouched in the hyperblock. Figure 11b shows the partial reverse if-conversion created for p1 after instructions 1 and 2. This means that instructions 1 and 2 are left in their originally scheduled location and the reverse if-converting branch, predicated on p1, is scheduled immediately following them. The location of instructions after the branch is determined by the PFG. To use the PFG without experiencing a path explosion problem, the PFG's generated during scheduling are done

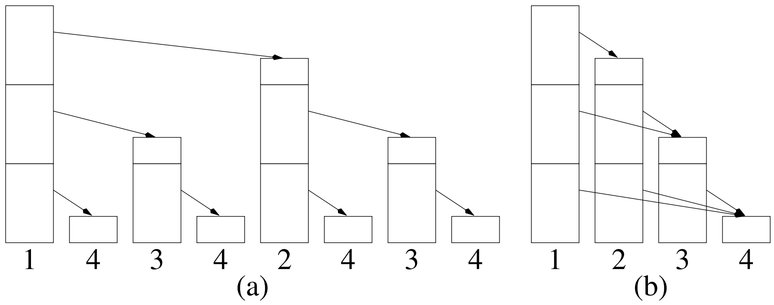


Figure 12: Simple code size reduction on multiple partial reverse if-conversions applied to an unrolled loop. Each square represents an unroll of the original loop.

only with respect to the predicate which is being reverse if-converted. This keeps the number of paths under control since the single-predicate PFG can contain no more than two paths. Figure 11a shows the PFG created for the predicate to be reverse if-converted, p1. Note that the partial dead code has already been removed as described in the previous section. Instructions which exist solely in the “p1 is FALSE” path, such as 5 and 6, remain in the original block. Instructions which exist solely in the “p1 is TRUE” path, such as 3, 4, and 7, are moved from the original block to the newly formed region. An instruction which exists in both paths must be placed in both regions.

Notice that the hyperblock conditionally jumps to the code removed from the hyperblock but there is no branch from this code back into the original hyperblock. While this is possible, it was not implemented in this work. Branching back into the hyperblock would violate the hyperblock semantics since it would no longer be a single entry region. Violating hyperblock semantics may not be problematic since the benefits of the hyperblock have already been realized by the optimizer and prepass scheduler. However, the postpass hyperblock scheduler may experience reduced scheduling freedom since each side entry into the hyperblock effectively divides the region into two smaller hyperblocks.

The advantage of branching back into the original hyperblock is a significant reduction in code size through elimination of unnecessarily duplicated instructions. However, as will be shown in the experimental section, code size was generally not a problem. One code size optimization which was performed merges targets of partial reverse if-conversion branches if the target blocks are identical. This resulted in a significant code size reduction in codes where loop unrolling was performed. If a loop in an unrolled hyperblock needed to be reverse if-converted, it is likely that all iterations needed to be reverse if-converted. This creates many identical copies of the loop body subsequent to the loop being reverse if-converted. Figure 12a shows the original result of repeated reverse if-conversions on an unrolled loop. Figure 12b shows the result obtained by combining identical targets. While this simple method works well in reducing code growth, it does not eliminate all unnecessary code growth. Eliminating all unnecessary code growth would require a mechanism capable of inserting a re-entry target at an opportune location in the hyperblock.

4.3 Policy

After creating the predicate flow graph and removing partial dead code, the identity and characteristics of all paths in a hyperblock are known. With this information, the compiler can make decisions on which transformations to perform. The decision process for partial reverse if-conversion consists of two parts: deciding which predicates to reverse if-convert and deciding where to reverse if-convert the selected predicates. To determine the optimal reverse if-conversion for a given architecture, the compiler could exhaustively try every possible reverse if-conversion, compute the optimal cycle count for each possibility, and choose the one with the best performance. Unfortunately, there are an enormous number of possible reverse if-conversions for any given hyperblock. Consider a hyperblock with p predicates and n instructions. This hyperblock has 2^p combinations of predicates that could be selected for reverse if-conversion. Each of these reverse if-conversions can then locate its branch in up to n locations in the worst case. Given that each of these possibilities must be scheduled to measure its cycle count, this can be prohibitively expensive. Obviously, a heuristic is needed. While many heuristics may perform effective reverse if-conversions, only one is studied in this paper. This heuristic may not be the best solution in all applications, but for the machine models studied in this work it achieves a desirable balance between final code performance, implementation complexity, and compile time.

The process of choosing a heuristic to perform partial reverse if-conversion is affected greatly by the type of scheduler used. Since partial reverse if-conversion is integrated into the prepass scheduler, the type of information provided by the scheduler and the structure of the code at various points in the scheduling process must be matched with the decision of what and where to reverse if-convert. An operation-based scheduler may yield one type of heuristic while a list scheduler may yield another. The policy presented here was designed to work within the context of an existing list scheduler described in [24] and [25]. This list scheduler performs a cycle-by-cycle greedy schedule using an instruction priority function. The partial reverse if-conversion algorithm as integrated in this scheduler is shown in Figure 13.

The first decision addressed by the proposed heuristic is where to place a predicate selected for reverse if-conversion. If one location can be shown to be generally more effective than the rest, then the number of locations to be considered for each

```
Initialize ready_priority_queue;
ric_queue = NULL;
cycle = 0;
num_unsched = Number of operations;
sched_no_ric = Compute_dynamic_cycles_for_hyperblock;
// Each trip through this loop is a new cycle
WHILE num_unsched != 0 DO
    // Handle reverse if-converting branches first
    FOREACH ric_op IN ric_queue DO
        IF Schedule_Op(ric_op, cycle) THEN
            Compute location for each unscheduled op;
            sched_ric_taken = Compute_dynamic_cycles_in_ric_taken_path;
            sched_ric_hb = Compute_dynamic_cycles_in_ric_hyperblock;
            mipred_ric = Estimate_ric_mispreds * miss_penalty;
            ric_cycles = sched_ric_hb + sched_ric_taken;
            ric_cycles = ric_cycles + mispred_ric;
            IF (sched_no_ric > ric_cycles) THEN
                sched_no_ric = sched_ric_hb;
                Place all ops in their no_ric schedule location;
            ELSE
                Unschedule_OP(ric_op);
            Remove ric_op from ric_queue;
    // Then handle regular operations
    FOREACH regular_op IN ready_priority_queue DO
        IF Schedule_Op(regular_op, cycle) THEN
            Remove regular_op from ready_priority_queue;
            num_unsched = num_unsched - 1;
            IF Is_Predicate_Define(regular_op) THEN
                Add reverse if-converting branch to ric_queue;
    cycle = cycle + 1;
```

Figure 13: An algorithm incorporating partial reverse if-conversion into a list scheduler

reverse if-conversion can be reduced from n to 1, an obvious improvement. Such a location exists under the assumption that the reverse if-converting branch consumes no resources and the code is scheduled by a perfect scheduler. It can be shown that there is no better placement than the first cycle in which the value of the predicate to be reverse if-converted is available after its predicate define instruction.[1] Since the insertion of the branch has the same misprediction or taken penalty regardless of its location, these effects do not favor one location over another. However, the location of the reverse if-converting branch does determine how early the paths sharing the same resources are separated and given the full machine bandwidth. The perfect scheduler will always do as well or better when the full bandwidth of the machine is divided among fewer instructions. Given this, the earlier the paths can be separated, the fewer the number of instructions competing for the same machine resources. Therefore, a best schedule will occur when the reverse if-converting branch is placed as early as possible.

Despite this fact, placing the the reverse if-converting branch as early as possible is a heuristic. This is because the two assumptions made, a perfect scheduler and no cost for the reverse if-converting branch, are not valid in general. It seems reasonable, however, that this heuristic would do very well despite these imperfections. Another consideration is code size, since instructions existing on multiple paths must be duplicated when these paths are separated. The code size can be reduced if the reverse if-converting branch is delayed. Depending on the characteristics of the code, this delay may have no cost or a small cost which may be less than the gain obtained by the reduction in code size. Despite these considerations, the placement of the partial reverse if-converting branch as early as possible is a reasonable choice.

The second decision addressed by the heuristic is what to reverse if-convert. Without a heuristic, the number of reverse if-conversions which would need to be considered with the heuristic described above is 2^p. The only way to optimally determine which combination of reverse if-conversions yields the best results is to try them all. A reverse if-conversion of one predicate can affect the effectiveness of other reverse if-conversions. This interaction among predicates is caused by changes in code characteristics after a reverse if-conversion has removed instructions from the hyperblock.

In the context of a list scheduler, a logical heuristic is to consider each potential reverse if-conversion in a cycle-by-cycle fashion, in the order in which the predicate defines are scheduled. This heuristic is used in the algorithm shown in Figure 13. This has the desirable effect of making the reverse if-conversion process fit seamlessly into a list scheduler. It is

[1] There exist machines where the placement of a branch a number of cycles after the computation of its condition removes all of its mispredictions [17]. In these machines, there are two locations which should be considered, immediately after the predicate define instruction and in the cycle in which the branch mispredictions are eliminated.

also desirable because each reverse if-conversion is considered in the context of the decisions made earlier in the scheduling process.

In order to make a decision on each reverse if-conversion, a method of evaluating its profitabiity must be employed. For each prospective reverse if-conversion, three schedules must be considered: the code schedule without the reverse if-conversion, the code schedule of the hyperblock with the reverse if-converting branch inserted and paths excluded, and the code schedule of the paths excluded by reverse if-conversion. Together they yield a total of $3p$ schedules for a given hyperblock. Each of these three schedules needs to be compared to determine if a reverse if-conversion is profitable. This comparison can be written as: $sched\_cycles_{no\_ric} > sched\_cycles_{ric\_hb} + sched\_cycles_{ric\_taken} + (mispred_{ric} * miss\_penalty)$ where $sched\_cycles_{no\_ric}$ is the number of dynamic cycles in the schedule without reverse if-conversion applied, $sched\_cycles_{ric\_hb}$ is the number of dynamic cycles in the schedule of the transformed hyperblock, $sched\_cycles_{ric\_taken}$ is the number of dynamic cycles in the target of the reverse if-conversion, and $mispred_{ric}$ is the number of mispredictions introduced by the reverse if-conversion branch. The $mispred_{ric}$ can be obtained through profiling or static estimates. $miss\_penalty$ is the branch misprediction penalty. This comparison is computed by lines 10 through 14 in Figure 13.

While the cost savings due to the heuristic is quite significant, $3p$ schedules can still be quite costly for the more complicated machine models. To reduce this cost, it is possible to reuse information gathered during one schedule in a later schedule.

The first source of reuse is derived from the cycle-by-cycle operation of the list scheduler itself. At the point each reverse if-conversion is considered, all previous instructions have been scheduled in their final locations by lines 8 or 22 in Figure 13. Performing the scheduling on the reverse if-conversion and the original scenario only needs to start at this point. The number of schedules is still $3p$, but the number of instructions in each schedule has been greatly reduced by the removal of instructions already scheduled.

The second source of reuse takes advantage of the fact that, for the case in which the reverse if-conversion is not done, the schedule has already been computed. At the time the previous predicate was considered for reverse if-conversion, the schedule was computed for each outcome. Since the resulting code schedule in cycles is already known, no computation is necessary for the current predicate's $sched\_cycles_{no\_ric}$. This reuse reduces the total number of schedules computed to $2p + 1$, with each schedule considering only the unscheduled instructions at each point due to the list scheduling effect.

This reuse is implemented in Figure 13 by lines 5 and 16.

Another way to reduce the total number of instructions scheduled is to take advantage of the fact that the code purged from the block is only different in the "then" and "else" blocks but not in the control equivalent split or join blocks. Once the scheduler has completely scheduled the "then" and "else" parts, no further scheduling is necessary since the remaining schedules are likely to be very similar. The only differences may be dangling latencies or other small differences in the available resources at the boundary. To be more accurate, the schedules can continue until they become identical, which is likely to occur at some point, though is not guaranteed to occur in all cases. An additional use for the detection of this point is code size reduction. This point is a logical location to branch from the ejected block back into the original hyperblock.

With all of the above schedule reuse and reduction techniques, it can be shown that the number of times an instruction is scheduled is usually $1 + 2d$, where d is that instruction's depth in its hammock. In the predication domain, this depth is the number of predicates defined in the chain used to compute that instruction's guarding predicate.

If the cost of scheduling is still high, estimates may be used instead. There are many types of scheduling estimates which have been proposed and can be found in the literature [25]. While many may do well for machines with regular structures, others do not. It is possible to create a hybrid scheduler/estimator which may balance good estimates with compile time cost. As mentioned previously, the schedule height of the two paths in the hammock must be obtained. Instead of purely scheduling both paths, which may be costly, or just estimating both paths, which may be inaccurate, a part-schedule and part-estimate approach may obtain more accurate results with lower cost. In the context of a list scheduler, one solution is the following: The scheduler could schedule an initial set of operations and estimate the schedule on those remaining. Accurate results will be obtained for the scheduled portion, in addition, the estimate may be able to benefit from information obtained from the schedule, as the characteristics of the scheduled code may be likely to match the characteristics of the code to be estimated. In the experiments presented in the next section, actual schedules are used in the decision to reverse if-convert because the additional compile time was acceptable.

4.4 Code Example

The examples presented up to this point in this section have been artificially crafted to illustrate the application of partial reverse if-conversion. It is, however, useful to examine the operation of the framework in a more realistic setting. The inner

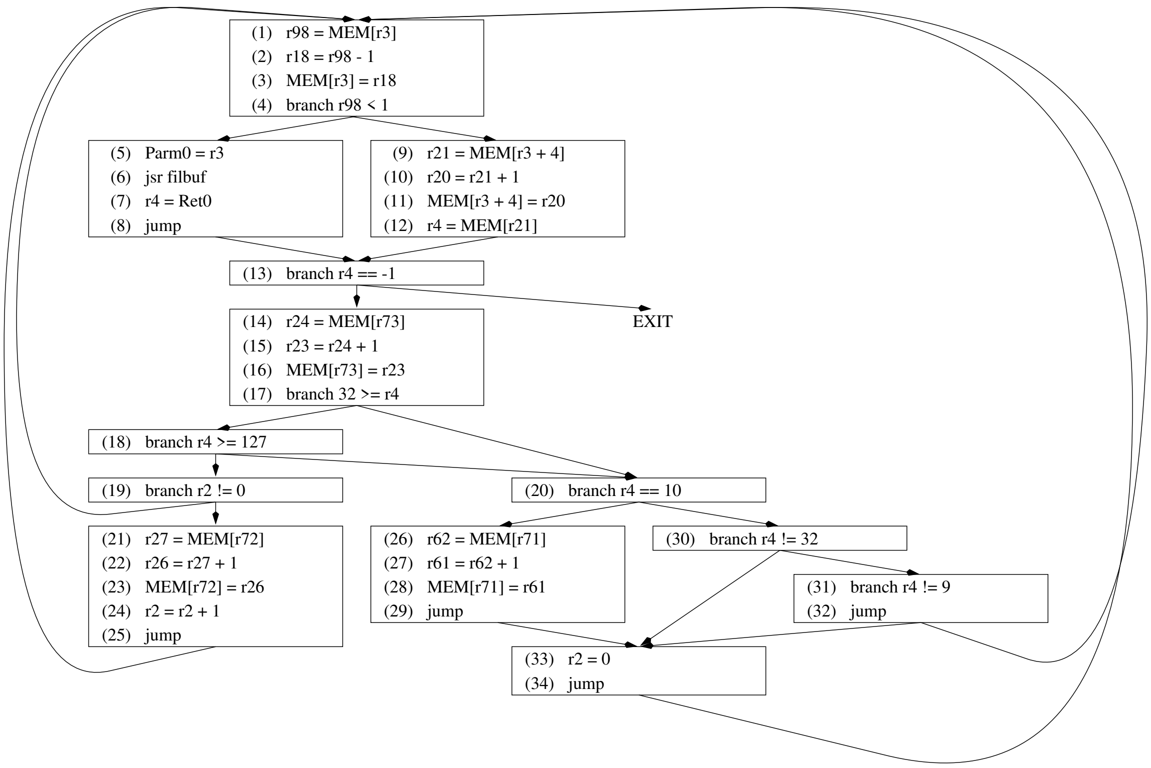


Figure 14: The control flow graph from the *UNIX* utility *wc*.

loop from our smallest benchmark, the *UNIX* utility *wc*, is chosen for this purpose. Figure 14 presents the original control flow graph for this code segment. Instructions are numbered 1 through 34 for reference. For this code segment, there are 22 paths of execution. The hyperblock formation heuristics must consider all of the possible paths to identify those profitable for inclusion in a hyperblock. The interaction between the paths is also important because they share many instructions.

The hyperblock formation heuristics select all instructions except 5-8 to combine into a single hyperblock. The resultant schedule of the hyperblock for an example three-issue processor is shown in Figure 15a. The processor utilized for this example is assumed to consist of one arithmetic unit, one memory unit, and one branch unit. All instructions have a latency of one cycle except loads, which have a latency of two cycles. Using branch profile information, the estimated execution cycle count for the hyperblock is 2.0M cycles. Unfortunately, the original code in which each basic block is scheduled separately requires only 1.7M cycles to execute. The hyperblock formation heuristics were too aggressive, leading to an oversubscription of the processor resources.

The partial reverse if-conversion framework is used to overcome this performance loss. The main problem with the

A: 1	(1) r98 = MEM[r3]		
2	(9) r21 = MEM[r3 + 4]		
3	(2) r18 = r98 - 1		
4	(3) MEM[r3] = r18	(4) branch r98 < 1	(10) r20 = r21 + 1
5	(11) MEM[r3 + 4] = r20		
6	(12) r4 = MEM[r21]		
7	(14) r24 = MEM[r73]		
8	(13) branch r4 == -1, EXIT	(17) p5(ot), p2(uf) = 32 >= r4	(21) r27 = MEM[r72]
9	(18) p5(ot), p3(uf) = r4 >= 127 <p2>	(26) r62 = MEM[r71]	
10	(19) p4 = 0 == r2 <p3>		
11	(20’) p7(ot), p6(uf) = r4 == 10 <p5>		
12	(30) p7(ot), p8(uf) = r4 == 32 <p6>		
13	(15) r23 = r24 + 1		
14	(20) p9 = r4 == 10 <p5>	(16) MEM[r73] = r23	
15	(22) r26 = r27 + 1		
16	(24) r2 = r2 + 1 <p4>	(23) MEM[r72] = r26 <p4>	
17	(27) r61 = r62 + 1		
18	(31) p7(ot) = r4 == 9 <p8>	(28) MEM[r71] = r61 <p9>	
19	(33) r2 = 0 <p7>	jump A	

(a)

A: 1	(1) r98 = MEM[r3]		
2	(9) r21 = MEM[r3 + 4]		
3	(2) r18 = r98 - 1		
4	(12) r4 = MEM[r21]	(10) r20 = r21 + 1	
5	(3) MEM[r3] = r18	(4) branch r98 < 1	
6	(11) MEM[r3 + 4] = r20	(17) p5(ot), p2(uf) = 32 >= r4	(13) branch r4 == -1, EXIT
7	(14) r24 = MEM[r73]	(18) p5(ot), p3(uf) = r4 >= 127 <p2>	
8	(20) p9 = r4 == 10 <p5>	(26) r62 = MEM[r71]	(18’) jump B <p3>
9	(20’) p7(ot), p6(uf) = r4 == 10 <p5>		
10	(30) p7(ot), p8(uf) = r4 == 32 <p6>		
11	(31) p7(ot) = r4 == 9 <p8>		
12	(15) r23 = r24 + 1		
13	(27) r61 = r62 + 1 <p9>	(16) MEM[r73] = r23	
14	(33) r2 = 0 <p7>	(28) MEM[r71] = r61 <p9>	jump A

B: 1	(19) p7(ot) = r4 == 9		
2	(19’) jump C <p4>	(15) r23 = r24 + 1	
3	(16) MEM[r73] = r23	jump A	

C: 1	(21) r27 = MEM[r72]	(15) r23 = r24 + 1	
2	(24) r2 = r2 + 1	(16) MEM[r73] = r23	
3	(22) r26 = r27 + 1		
4	(23) MEM[r72] = r26	jump A	

(b)

Figure 15: Schedule for *wc* after hyperblock formation and optimization (a), and schedule after partial reverse if-conversion of that hyperblock (b).

original hyperblock is that the arithmetic unit is over-saturated with instructions. Therefore, the scheduler needs to choose one or more paths to eject to reduce the pressure on the arithmetic unit. The resultant schedule with partial reverse if-conversion is shown in Figure 15b. Two reverse if-conversions are performed. First, predicate p3 is reverse if-converted by inserting a new jump instruction ($18'$). The jump conditionally branches to a new hyperblock, labeled B, when p3 is TRUE. The new jump is inserted into the schedule of the original hyperblock at the earliest cycle when p3 is available, cycle 8. All unscheduled instructions are put into one of three categories: required when p3 is TRUE, required when p3 is FALSE, or required for either value of p3. Instructions 19, 21, 22, 23 and 24 are only needed when p3 is TRUE, thus they are ejected from the original hyperblock. Instructions 15 and 16 are needed for either value of p3, thus they must be replicated in both the original and the new hyperblock. The remainder of the instructions are required when p3 is FALSE, so they remain in the original hyperblock. The remainder of the original hyperblock is scheduled without further reverse if-conversions.

A second reverse if-conversion is applied when hyperblock B is scheduled. Predicate p4 is reverse if-converted by introducing a new jump instruction ($19'$). The jump conditionally branches to the new hyperblock, labeled C, when p4 is TRUE. As a result of the transformation, instructions 21, 22, 23 and 24 are ejected from hyperblock B. Instructions 15 and 16 are again replicated. The scheduling process is complete when hyperblock C is scheduled.

The overall result is that execution of the code with partial reverse if-conversion is reduced to 1.3M cycles. This compares favorably with 2.0M cycles for the original hyperblock code and 1.7M cycles for the original basic block code. The primary reason for the improvement in this example was the ability to eject instructions to reduce the contention for the arithmetic unit.

5 Experimental Results

The partial reverse if-conversion framework described in this paper has been implemented in the second generation instruction scheduler of the IMPACT compiler. This section presents an experimental evaluation of this framework.

5.1 Methodology

The IMPACT compiler utilizes a machine description file to generate code for a parameterized superscalar processor. To measure the effectiveness of the partial reverse if-conversion technique, a machine model similar to many current processors was selected. The machine modeled is a four-issue, in-order superscalar processor that contains two integer ALU's, two memory ports, one floating point ALU, and one branch unit. The instruction latencies assumed match those of the HP PA-7100 microprocessor [26]. The instruction set contains non-trapping versions of all potentially excepting instructions, with the exception of branch and store instructions, to support aggressive speculative execution. The instruction set also contains support for predication similar to that provided in the PlayDoh architecture [8].

The execution time for each benchmark is derived from the static code schedule weighted by dynamic execution frequencies obtained from profiling. Static branch prediction based on profiling is also utilized. Benchmark performance ignores dynamic stall cycles associated with the memory system including instruction and data cache misses. Previous experience with this method of run time estimation has demonstrated that it accurately estimates simulations of an equivalent machine with perfect caches.

The benchmarks used in this experiment consist of 14 non-numeric programs: the six SPEC CINT92 benchmarks, *008.espresso*, *022.li*, *023.eqntott*, *026.compress*, *072.sc*, and *085.cc1*; two SPEC CINT95 benchmarks, *132.ijpeg* and *134.perl*; and six *UNIX* utilities *cccp*, *cmp*, *eqn*, *grep*, *wc*, and *yacc*.

5.2 Results

For the experiments, the performance of the traditional hyperblock and the new partial reverse if-conversion frameworks are compared. The hyperblocks formed in these experiments represent those currently formed by the IMPACT compiler's hyperblock formation heuristic for the target machine [14]. For the traditional hyperblock framework, these hyperblocks are directly scheduled for the target machine. These same hyperblocks were also used as input to the partial reverse if-conversion framework. In this case, the hyperblocks are scheduled and reverse if-converted as appropriate for the target machine. The performance of both frameworks is presented relative to the results obtained with superblock compilation for the same target machine. Superblock compilation performance is selected as the base because it represents the best possible performance currently obtainable by the IMPACT compiler without predication [27].

Overall performance. Figures 16 and 17 compare the overall benchmark performance of the hyperblock and partial reverse if-conversion frameworks. Performance is reported as the speedup in execution cycles versus superblock compilation with the height of the bars in both graphs are computed as, $superblock\_cycles/technique\_cycles$.

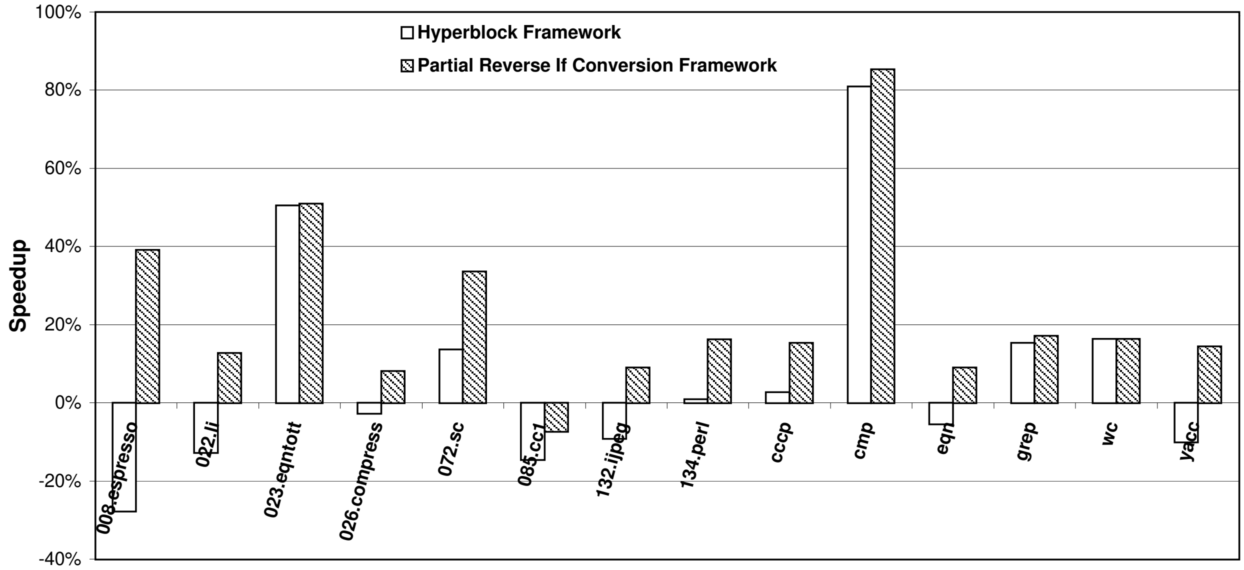


Figure 16: Performance increase over superblock exhibited by the hyperblock and partial reverse if-conversion frameworks with no misprediction penalty.

Figure 16 shows the performance of the hyperblock and partial reverse if-conversion frameworks assuming no branch misprediction penalty. Since branch mispredictions are not factored in, benchmarks exhibiting performance improvement in this graph show that predication has performed well as a compilation model. In particular, the compiler has successfully overlapped the execution of multiple paths of control to increase ILP. Hyperblock compilation achieves some speedup for half of the benchmarks, most notably for *023.eqntott*, *cmp*, *072.sc*, *grep*, and *wc*. For these programs, the hyperblock techniques successfully overcome the problem superblock techniques were having in fully utilizing processor resources. On the other hand, hyperblock compilation results in a performance loss for the other half of the benchmarks. This dichotomy is a common problem experienced with hyperblocks: hyperblocks can do well, but performance often falls victim to poor hyperblock selection.

In all cases, partial reverse if-conversion improved upon or matched the performance of the hyperblock code. For six of the benchmarks, partial reverse if-conversion was able to change a loss in performance by hyperblock compilation into a gain. This is most evident for *008.espresso* where a 28% loss was converted into a 39% gain. For *072.sc*, *134.perl*, and *cccp*, partial reverse if-conversion was able to significantly magnify relatively small gains achieved by hyperblock compilation. These results indicate that the partial reverse if-converter was successful at undoing many of the poor hyperblock formation decisions while capitalizing on the effective ones. For the four benchmarks where hyperblock techniques were

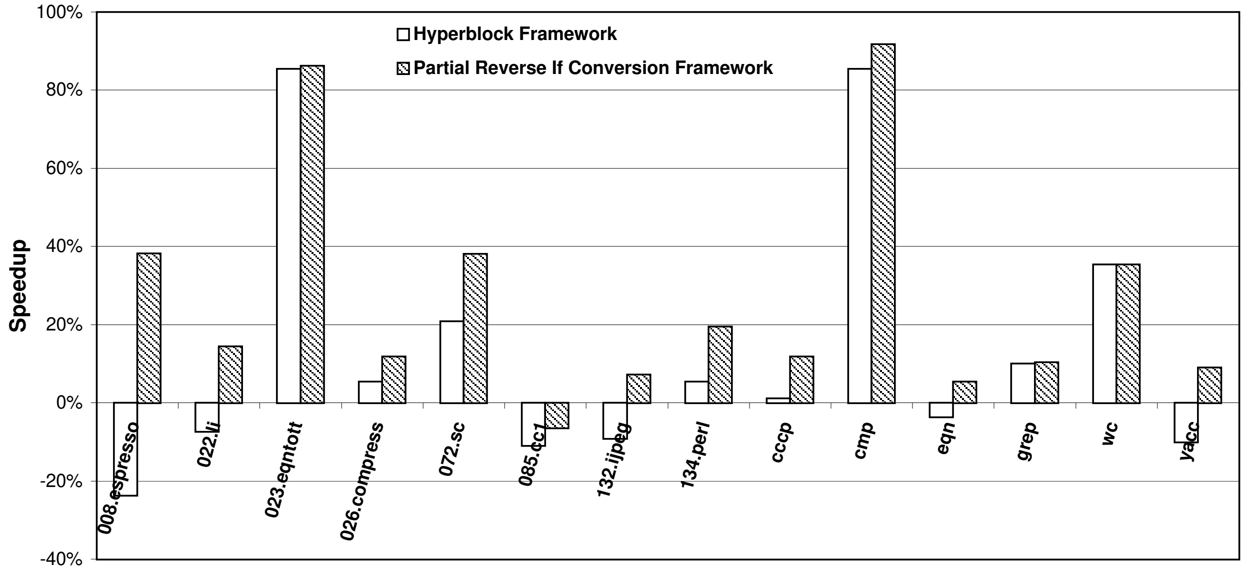


Figure 17: Performance increase over superblock exhibited by the hyperblock and partial reverse if-conversion frameworks with a four cycle misprediction penalty.

highly effective, *023.eqntott*, *cmp*, *grep*, and *wc*, partial reverse if-conversion does not have a large opportunity to increase performance since the hyperblock formation heuristics worked well in deciding what to if-convert.

It is useful to examine the performance of the worst performing benchmark, *085.cc1*, more closely. For this benchmark, both frameworks result in a performance loss with respect to superblock compilation. Partial reverse if-conversion was not completely successful in undoing the bad hyperblock formation decisions. This failure is due to the policy that requires the list scheduler to decide the location of the reverse if-converting branch by its placement of the predicate define instruction. Unfortunately, the list scheduler may delay this instruction as it may not be on the critical path and is often deemed to have a low scheduling priority. Delaying the reverse if-conversion point can have a negative effect on code performance. To some extent this problem occurs in all benchmarks, but is most evident in *085.cc1*.

Figure 17 shows the performance of the benchmarks in the same manner as Figure 16 except with a branch misprediction penalty of four cycles. In general, the relative performance of hyperblock code is increased the most when mispredicts are considered because it has the fewest mispredictions. The relative performance of the partial reverse if-conversion code is also increased because it has fewer mispredictions than the superblock code. But, partial reverse if-conversion inserts new branches to accomplish its transformation, so this code contains more mispredictions than the hyperblock code. For several of the benchmarks, the number of mispredictions was actually larger for hyperblock and partial reverse if-conversion than

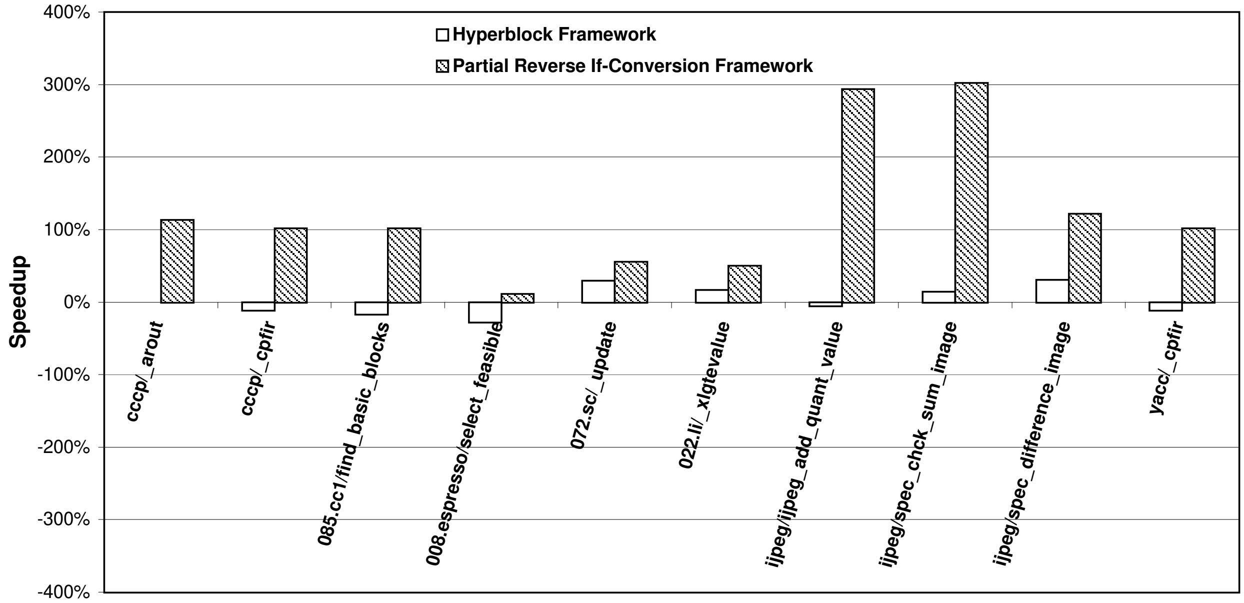


Figure 18: Selected function level performance increase over superblock exhibited by the hyperblock and partial reverse if-conversion frameworks with no misprediction penalty.

that of superblock. When applying control flow transformations in the predicated representation, the compiler will actually create branches with much higher mispredict rates than those removed. Additionally, the branches created by partial reverse if-conversion may be more unbiased than the the combination of branches in the original superblock they represent.

Function-level performance. It is illustrative to examine performance at a finer granularity for a better understanding of the results. Figure 18 compares the performance of selected functions from the benchmarks in the same manner as Figure 16. The figure assumes no branch misprediction penalty for the results. The functions were selected based on two criteria: contributing a high fraction of the overall benchmark execution time and benefiting a large amount from partial reverse if-conversion. The figure shows the potential of partial reverse if-conversion to increase performance dramatically. In particular, two functions in *132.ijpeg* achieve greater than 250% gain with partial reverse if-conversion. Hyperblock compilation is relatively ineffective for these two functions. The ability to adjust the control structure of the code during scheduling, however, allows the execution time of these same hyperblocks to be reduced dramatically. Even the worst overall benchmark, *085.cc1*, contains an important function that achieves 100% gain with partial reverse if-conversion.

One of the benchmarks that performed best overall was *072.sc*. For this program, hyperblock compilation increased performance by a fair margin, but partial reverse if-conversion increased this gain substantially. Most of *072.sc*'s perfor-

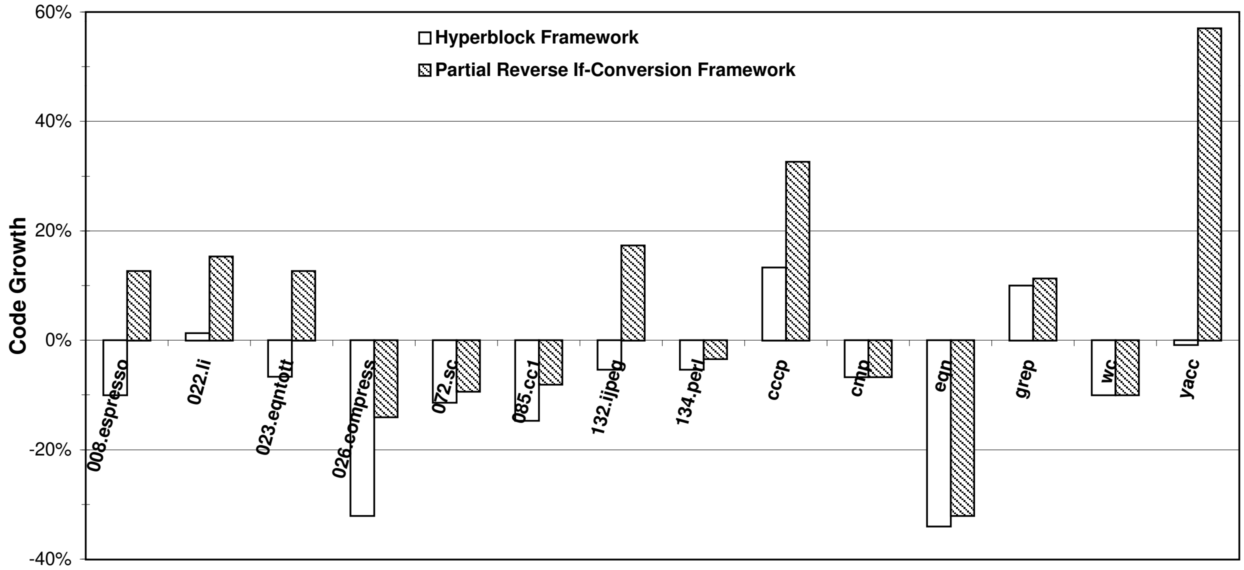


Figure 19: Relative static code size exhibited by the hyperblock and partial reverse if-conversion frameworks compared with superblock.

mance gain was achieved by transforming a single function, *_update*. This function with superblock compilation executes in 25.6 million cycles, with a rather sparse schedule due to a large number of data and control dependences. Hyperblock compilation increases the available ILP by eliminating a large fraction of the branches and overlapping the execution of multiple paths of control. This brings the execution time down to 19.7 million cycles. While the hyperblock code is much better than the superblock code, it has excess resource consumption on some paths which penalizes other paths. The partial reverse if-converter was able to adjust the amount of if-conversion to match the available resources, efficiently utilizing the processor. As a result, the execution time for the *_update* function is reduced to 16.8 million cycles with partial reverse if-conversion, a 52% performance improvement over the superblock code.

Code size. The static size of code generated using the hyperblock and partial reverse if-conversion frameworks is compared to the size of superblock code in Figure 19. From the figure, the use of predicated execution by the compiler has varying effects on the code size. The reason for this behavior is a tradeoff between increased code size caused by if-conversion with the decreased code size due to less tail duplication. With superblocks, tail duplication is performed extensively to customize individual execution paths. With predication, multiple paths are overlapped via if-conversion, so less tail duplication is required. The figure also shows that the code produced with the partial reverse if-conversion framework is consistently larger than the hyperblock framework. On average, the partial reverse if-conversion code is 14%

larger than the hyperblock code, with the largest growth occurring for *yacc*.

Common to all the benchmarks which exhibit a large code growth was a failure of the simple code size reduction mechanism presented Section 4. Inspection of the resulting code indicates that many instructions are shared in the lower portion of the tail duplications created by the partial reverse if-converter. For this reason, one can expect these benchmarks to respond well to a more sophisticated code size reduction scheme.

Application statistics. Finally, the frequency of partial reverse if-conversions that were performed to generate the performance data is presented in Table 1. The "Reverse If-Conversions" column specifies the actual number of reverse if-conversions that occurred across the entire benchmark. The "Opportunities" column specifies the number of reverse if-conversions that could potentially have occurred. The number of opportunities is equivalent to the number of unique predicate definitions in the application, since each predicate define can be reverse if-converted exactly once. All data in Table 1 are static counts. The table shows that the number of reverse if-conversions that occur is a small fraction of the opportunities. This behavior is desirable as the reverse if-converter should try to minimize the number of branches it inserts to achieve the desired removal of instructions from a hyperblock. In addition, the reverse if-converter should only be invoked when a performance problem exists. In cases where the performance of the original hyperblock cannot be improved, no reverse if-conversions need to be performed. The table also shows the expected correlation between large numbers of reverse if-conversions and larger code size increases of partial reverse if-conversion over hyperblock (Figure 19).

6 Conclusion

In this paper, we have presented an effective framework for compiling applications for architectures which support predicated execution. The framework consists of two major parts. First, aggressive if-conversion is applied early in the compilation process. This enables the compiler to fully utilize the predicate representation in applying aggressive ILP optimizations and control flow transformations. The second component of the framework is applying partial reverse if-conversion at schedule time. This delays the final if-conversion decisions until the point during compilation when the relevant information about the code content and the processor resource utilization is known.

A first generation partial reverse if-converter was implemented and the effectiveness of the framework was measured for this paper. The framework was able to capitalize on the benefits of predication without being subject to the sometimes negative side effects of over-aggressive hyperblock formation. Furthermore, additional opportunities for performance improvement were exploited by the framework, such as partial path if-conversion. These points were demonstrated by the hyperblock performance losses which were converted into performance gains, and by moderate gains which were further magnified. We expect continuing development of the partial reverse if-converter and the surrounding scheduling infrastructure to further enhance performance. In addition, the framework provides an important mechanism for eliminating the negative effects of overly aggressive transformations at schedule time. With such a backup mechanism, unique opportunities are introduced for the aggressive use and transformation of the predicate representation early in the compilation process.

Acknowledgments

The authors would like to thank John Sias, Kevin Crozier, John Gyllenhaal, Teresa Johnson, Brian Deitrich, Daniel Connors, and all the members of the IMPACT compiler team for their support, comments, and suggestions. This research has been supported by the National Science Foundation (NSF) under grant CCR-9629948, Intel Corporation, Advanced Micro Devices, Hewlett-Packard, SUN Microsystems, and NCR. Additional support was provided by an Intel Foundation Fellowship.

References

[1] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th International Symposium on Computer Architecture*, pp. 135–148, May 1981.

[2] T. Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pp. 51–61, November 1991.

[3] N. J. Warter, *Modulo Scheduling with Isomorphic Control Transformations*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.

[4] 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.

[5] J. C. Park and M. S. Schlansker, "On predicated execution," Tech. Rep. HPL-91-58, Hewlett Packard Laboratories, Palo Alto, CA, May 1991.

[6] P. Y. Hsu and E. S. Davidson, "Highly concurrent scalar processing," in *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 386–395, June 1986.

[7] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *IEEE Computer*, vol. 22, pp. 12–35, January 1989.

[8] V. Kathail, M. S. Schlansker, and B. R. Rau, "HPL PlayDoh architecture specification: Version 1.0," Tech. Rep. HPL-93-80, Hewlett-Packard Laboratories, Palo Alto, CA, February 1994.

[9] D. N. Pnevmatikatos and G. S. Sohi, "Guarded execution and branch prediction in dynamic ILP processors," in *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 120–129, April 1994.

[10] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. W. Hwu, "Characterizing the impact of predicated execution on branch prediction," in *Proceedings of the 27th International Symposium on Microarchitecture*, pp. 217–227, December 1994.

[11] G. S. Tyson, "The effects of predicated execution on branch prediction," in *Proceedings of the 27th International Symposium on Microarchitecture*, pp. 196–206, December 1994.

[12] M. Schlansker, V. Kathail, and S. Anik, "Height reduction of control recurrences for ILP processors," in *Proceedings of the 27th International Symposium on Microarchitecture*, pp. 40–51, December 1994.

[13] 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.

[14] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, R. A. Bringmann, and W. W. Hwu, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the 25th International Symposium on Microarchitecture*, pp. 45–54, December 1992.

[15] S. A. Mahlke, R. E. Hank, J. McCormick, D. I. August, and W. W. Hwu, "A comparison of full and partial predicated execution support for ILP processors," in *Proceedings of the 22th International Symposium on Computer Architecture*, pp. 138–150, June 1995.

[16] D. I. August, J. W. Sias, J. Puiatti, S. A. Mahlke, D. A. Connors, K. M. Crozier, and W. W. Hwu, "The program decision logic approach to predicated execution," in *Proceedings of the 26th International Symposium on Computer Architecture*, pp. 208–219, May 1999.

[17] D. I. August, D. A. Connors, J. C. Gyllenhaal, and W. W. Hwu, "Architectural support for compiler-synthesized dynamic branch prediction strategies: Rationale and initial results," in *The 3rd International Symposium on High-Performance Computer Architecture*, pp. 84–93, February 1997.

[18] J. C. Gyllenhaal, *An Efficient Framework For Performing Execution-Constraint-Sensitive Transformations That Increase Instruction-Level Parallelism*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1997.

[19] D. I. August, W. W. Hwu, and S. A. Mahlke, "A framework for balancing control flow and predication," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pp. 92–103, December 1997.

[20] N. J. Warter, S. A. Mahlke, W. W. Hwu, and B. R. Rau, "Reverse if-conversion," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pp. 290–299, June 1993.

[21] R. Johnson and M. Schlansker, "Analysis techniques for predicated code," in *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 100–113, December 1996.

[22] D. M. Gillies, D. R. Ju, R. Johnson, and M. Schlansker, "Global predicate analysis and its application to register allocation," in *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 114–125, December 1996.

[23] J. Knoop, O. Ruthing, and B. Steffen, "Partial dead code elimination," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementaton*, pp. 147–158, June 1994.

[24] R. A. Bringmann, *Compiler-Controlled Speculation*. PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1995.

[25] B. L. Deitrich and W. W. Hwu, "Speculative hedge: Regulating compile-time specculation against profile variations," in *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 70–79, December 1996.

[26] Hewlett-Packard Company, Cupertino, CA, *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, 1990.

[27] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229–248, January 1993.

Benchmark	Reverse If-Conversions	Opportunities
008.espresso	204	1552
022.li	50	393
023.eqntott	43	443
026.compress	11	56
072.sc	33	724
085.cc1	479	3827
132.ijpeg	134	1021
134.perl	42	401
cccp	77	1046
cmp	4	49
eqn	33	326
grep	3	103
wc	0	88
yacc	247	1976

Table 1: Application frequency of partial reverse if-conversion.