A 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

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 effectively apply if-conversion, one must address two major issues: what should be if-converted and when the if-conversion should be applied. A compiler's use of predication as a representation is most effective when large amounts of code are if-converted and if-conversion is performed early in the compilation procedure. On the other hand, the final code generated for a processor with predicated execution requires a delicate balance between control flow and predication to achieve efficient execution. The appropriate balance is tightly coupled with scheduling decisions and detailed processor characteristics. This paper presents an effective compilation framework 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.

1 Introduction

The performance of modern processors is becoming highly dependent on the ability to execute multiple instructions per cycle. In order to realize their performance potential, these processors demand that increasing levels of instruction-level parallelism (ILP) 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 along with speculative execution is generally employed to overcome these limitations [1][2]. However, branch misprediction takes away a significant portion of the potential performance gain. Third, traditional techniques only facilitate exploiting ILP along a single trajectory of control. The ability to concurrently execute instructions from multiple trajectories offers the potential to increase ILP by large amounts. 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 branches effectively in both the compilation and execution of codes. It provides benefits in a compiler 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 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 defining instructions. The use of predicates to guard instruction execution can reduce or even completely eliminate the need for branch control dependences. When all instructions that are control dependent on a branch are predicated using the same condition as the branch, that branch can legally be removed. The process of replacing branches with appropriate predicate computations and guards is known as *if-conversion* [3][4].

The predicated representation provides an efficient and useful model for compiler optimization and scheduling. Through the removal of branches, code can be transformed to contain few, if any, control dependences. Complex control flow transformations can instead be performed in the predication domain as traditional straight-line code optimizations. In the same way, the predicated representation allows scheduling among branches to be performed in a domain without control dependences. The removal of these control dependences increases scheduling scope and affords new freedom to the scheduler [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 removal 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 for a compiler to 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 height reduction along a single program path [12].

Supporting predicated execution introduces two compilation issues: 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, what branches should be removed via if-conversion. Traditionally, full if-conversion has led to positive results for compiling numerical applications [13]. However, for non-numeric applications, selective if-conversion is essential to achieve performance gains [14]. If-conversion works by removing branches and combining multiple paths of control into a single path of conditional instructions. However, when two paths are overlapped, the resultant path can exhibit increased constraints over those of the original paths. One important constraint is resources. Paths which are combined together must share processor resources. The compiler has the responsibility of managing the available resources when making if-conversion decisions so that an appropriate stopping point may be identified. Further if-conversion will only result in an increase in execution time for all the paths involved. As will be discussed in the next section, the problem of deciding what to if-convert is complicated by many factors, only one of which is resource consumption.

The second question that must be addressed is when to apply if-conversion in the compilation procedure. At the broadest level, if-conversion may be applied early in the backend compilation procedure or delayed to occur in conjunction with scheduling. Applying if-conversion early enables the full use of the predicate representation by the compiler to facilitate ILP optimizations and scheduling. In addition, complex control flow transformations may be recast in the data dependence domain to make them practical and profitable. Examples of such transformations include branch reordering, control height reduction [12], and branch combining [15]. 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 the processor characteristics, applying if-conversion at schedule time is the most natural choice. Also, applying if-conversion during scheduling alleviates the need to make the entire compiler backend cognizant of a predicated representation.

An effective compiler strategy for 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 predicate 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 facilitate balancing 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 details the compilation issues and challenges associated with compiling for predicated execution. Section 3 introduces our proposed compilation framework that facilitates taking full advantage of the predicate representation as well as achieving 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 presented in Section 5. Finally, the paper concludes in Section 6.

2 Compilation Challenges

Effective use of predicated execution provides a difficult challenge for ILP compilers. Predication offers the potential for large performance gains when it is efficiently utilized. However, an imbalance of predication and control flow in the generated code can lead to dramatic performance losses. The baseline compilation support for predicated execution assumed in this paper is the hyperblock framework. Hyperblocks and the issues associated with forming quality hyperblocks are first summarized in this section. The remainder of this section focuses on an approach of forming hyperblocks early in the compilation procedure using heuristics. This technique is useful because it exposes the predicate representation throughout the backend optimization and scheduling process. However, this approach has several inherent weaknesses. Solving these weaknesses is the motivation for the framework presented in this paper.

2.1 Background

The *hyperblock* is a structure created to facilitate optimization and scheduling for predicated architectures [14]. A hyperblock is a set of predicated basic blocks in which control may only enter from the top, but may exit from one or more locations. Hyperblocks are formed by applying tail duplication and if-conversion over a set of carefully selected paths. Inclusion of a path into a hyperblock is done by considering its profitability. The profitability is determined by four pieces of information: resource utilization, dependence height, hazard presence, and execution frequency. One can gain insights into effective hyperblock formation heuristics by understanding how each characteristic can lead to performance loss.

The most common cause of poor hyperblock formation is excessive resource consumption. The resources required by overlapping the execution of multiple paths are the union of the resources required for each individual path. Consider if-converting a simple if-then-else statement. The resultant resource consumption for the hyperblock will be the combination of the resources required to separately execute the “then” and “else” paths. If each path alone consumes almost all of the processor resources, the resultant hyperblock would require substantially more resources than the processor has available. As a result, hyperblock formation results in a significant slowdown for both paths. Of course, these calculations do not account for the benefits gained by if-conversion. The important point is that resource oversubscription has the potential to negate all benefits of hyperblock formation or even degrade performance.

Poor hyperblocks may also be formed by not carefully considering the dependence height of if-conversion. A hyperblock which contains multiple paths will not complete until all of its constituent paths have completed. Therefore, the overall height of the hyperblock is the maximum of all the original paths' de-

pendence heights. Consider the if-conversion of a simple if-then-else statement with the “then” path having a height of two and the “else” path having a height of four. The height of the resultant hyperblock is the maximum of both paths, four. As a result, the “then” path is potentially slowed down by two times. The compiler must weigh this negative against the potential positive effects of if-conversion to determine whether this hyperblock is profitable to form.

Another way poor hyperblocks may be formed is through the inclusion of a path with a hazard. A hazard is any instruction or set of instructions which hinders efficient optimization or scheduling of control paths other than its own. Two of the most common hazards are subroutine calls with unknown side effects and store instructions which have little or no alias information. Hazards degrade performance because they force the compiler to make conservative decisions in order to ensure correctness. For this reason, inclusion of a control path with a hazard into a hyperblock generally reduces the compiler's effectiveness for the entire hyperblock, not just for the path containing the hazard.

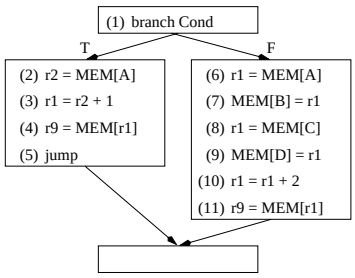
Execution frequency is used as a measure of a path's importance and also provides insight into branch behavior. This information is used to weigh the trade-offs made in combining execution paths. For example, it may be wise to penalize an infrequently executing path by combining it with a longer frequently executing path and removing the branch joining the two.

2.2 Pitfalls of Hyperblock Selection

The original approach used in the IMPACT compiler to support predicated execution is to form hyperblocks using heuristics based on the four metrics described in the previous section. Hyperblocks are formed early in the backend compilation procedure to expose the predicate representation throughout all the backend compilation phases. Heuristic hyperblock formation has been shown to perform well for relatively regular machine models. In these machines, balancing resource consumption, balancing dependence height, and eliminating hazards are done effectively by the carefully crafted heuristics. However, experience shows that several serious problems exist that are difficult to solve with this approach. Three such problems presented here are optimizations that change code characteristics, unpredictable resource interference, and partial path inclusion.

Optimization. The first problem occurs when code may be transformed after hyperblock formation. In general, forming hyperblocks early facilitates optimization techniques that take advantage of the predicate representation. However, the hyperblock formation decisions can change dramatically with compiler transformations. This can convert a seemingly bad formation decision into a good one. Likewise, it can convert a seemingly good formation decision into a bad one.

Figure 1a shows a simple hammock to be considered for if-conversion.[1] The taken path has a dependence height of three cycles and consumes three instruction slots after if-conversion has removed instruction 5. The fall-through path consists of a dependence height of six cycles and a resource consumption of six instruction slots. A simple estimate indicates that combining these paths would result in a penalty for the taken path of three cycles due to the fall-through path's large dependence height. Figure 1b shows this code segment after hyperblock formation and further optimizations. The first optimization performed was renaming to eliminate the false dependences $7 \rightarrow 8$ and $8 \rightarrow 10$. This reduced the dependence height of the hyperblock to three cycles.



(a)

1	(1) p1, $\overline{p2}$ = 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)

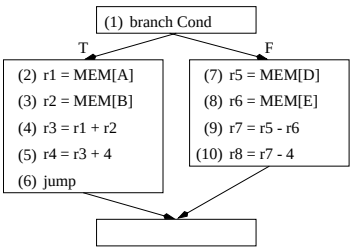
Figure 1: Hyperblock formation of seemingly incompatible paths with positive results due to code transformations. The T and F annotations in (a) indicate the taken and fall-through path for the conditional branch. r2 is not referenced outside the T block.

If a heuristic could foresee that dependence height would no longer be an issue, it may still choose not to form this hyperblock due to resource considerations. An estimate of ten instructions after if-conversion could be made by inspecting Figure 1a. Unfortunately, ten instructions needs at least four cycles to complete on a three issue machine, which would still penalize the taken path by one cycle, indicating that the combination of these paths may not be beneficial. After an instruction merging optimization in which instructions 2 and 6 were combined and 4 and 11 were combined, the instruction count becomes eight. The final schedule consists of only three cycles.

Figure 1 shows that even in simple cases a heuristic which forms hyperblocks before some optimizations must anticipate the effectiveness of those optimizations in order to form profitable hyperblocks. In this example, some optimizations could potentially be done before hyperblock formation, such as renaming. However, others, like instruction merging, could not have been. In addition, some optimizations may have been applied differently when performed before if-conversion, because the different code characteristics will result in different trade-offs.

Resource Interference. A second problem with heuristic hyperblock formation is that false conclusions regarding the resource compatibility of the candidate paths may often be reached. As a result, paths which seem to be compatible for if-conversion turn out to be incompatible. The problem arises because resource usage estimation techniques, such as the simple ones used in this section or even other more complex techniques, generally assume

[1] For all code examples presented in this section, a simple machine model is used. The schedules are for a three issue processor with unit latencies. Any resource limitations for the processor that are assumed are specified with each example. These assumptions do not reflect the machine model or latencies used in the experimental evaluation (Section 5).



(a)

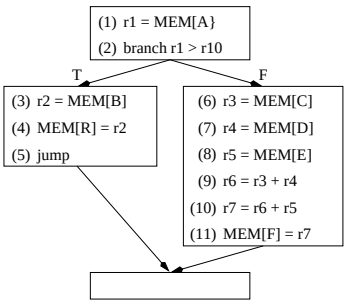
1	(1) p1, $\overline{p2}$ = Cond	(2) r1 = MEM[A]	
2	(3) r2 = MEM[B] <p1>		
3	(4) r3 = r1 + r2 <p1>	(7) r5 = MEM[D] <p2>	
4	(5) r4 = r3 + 4 <p1>	(8) r6 = MEM[E] <p2>	
5	(9) r7 = r5 - r6 <p2>		
6	(10) r8 = r7 - 4 <p2>		

(b)

Figure 2: Hyperblock formation of seemingly compatible paths that results in performance loss due to resource incompatibility.

that resource usage is evenly distributed across the block. In practice, however, few paths exhibit uniform resource utilization. Interactions between dependence height and resource consumption cause violations of the uniform utilization assumption. In general, most paths can be subdivided into sections that are either relatively parallel or relatively sequential. The parallel sections demand a large number of resources, while the sequential sections require few resources. When two paths are combined, resource interference may occur when the parallel sections of the paths overlap. For those sections, the demand for resources is likely to be larger than the available resources, resulting in a performance loss.

To illustrate this problem, consider the example in Figure 2. The processor assumed for this example is again three issue, but at most one memory instruction may be issued each cycle. The original code segment, Figure 2a, consists of two paths with dependence heights of three cycles. The resource consumption of each path is also identical, four instructions. These paths are concluded to be good candidates for if-conversion. Figure 2b shows the hyperblock and its resulting schedule. Since there are no obvious resource shortages, one would expect the resultant schedule for the hyperblock to be identical in length to the schedules of each individual path, or four cycles. However, the hyperblock schedule length turns out to be six cycles. This increase is due to resource interference between the paths. Each path is parallel at the start and sequential at the end. In addition, the parallel sections of both paths have a high demand for the memory resource. With only one memory resource available, the paths are sequentialized in parallel sections. Note that if the requirements for the memory resource were uniformly distributed across both paths, this problem would not exist as the individual schedule lengths are four cycles and there are a total of four memory instructions. However, due to the characteristics of these paths, resource interference results in a performance loss for both paths selected for the hyperblock.



(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 3: An efficient hyperblock formed through the inclusion of a partial path.

Partial Paths. The final problem with current heuristic hyperblock formation is that paths may not be subdivided when they are considered for inclusion in a hyperblock. In many cases, including part of a path may be more beneficial than including or excluding that entire path. Such an if-conversion 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 by overlapping a part of the candidate path with the other paths selected for inclusion in the hyperblock.

To illustrate the effectiveness of partial if-conversion, consider the example in Figure 3. The three issue processor assumed for this example does not have any resource limitations other than the issue width. Figure 3a shows two paths which are not compatible due to mismatched dependence height. However, by including all of the taken path and four instructions from the fall-through path, an efficient hyperblock is created. This hyperblock is shown in Figure 3b. 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 as the original. However, the schedule did benefit from the partial overlap of both paths. The destination of branch instruction $2''$ contains the code to complete the fall-through path is shown in Figure 3c.

In theory, hyperblock formation heuristics may be extended to support partial paths. Since each path could be divided at any instruction in the path, the heuristics would have to consider many more possible selection alternatives. However, the feasibility of extending the selection heuristics to operate at the finer granularity of instructions, rather than whole paths, is questionable due the complex nature of the problem.

3 Proposed Compilation Framework

Compilation for predicated execution can be challenging as described in Section 2. To create efficient code, a delicate balance between control flow and predication must be created. The desired 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 structure for making intelligent tradeoffs between control flow and predication so the desired balance can be achieved.

Given the difficulties presented in Section 2.2 with forming hyperblocks early in the backend compilation process, a seemingly natural strategy is to perform if-conversion in conjunction with instruction scheduling. This can be achieved by integrating if-conversion within the scheduling process itself. A 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 make intelligent if-conversion decisions. In addition, all compiler optimizations are usually complete when scheduling is reached, thus the problem of the code changing after if-conversion does not exist.

However, a very serious problem associated with performing if-conversion during scheduling time is the restriction on the compiler's use of the predicate representation to perform control flow transformations and predicate specific optimizations. With the schedule-time framework, the introduction of the predicate representation is delayed until schedule time. As a result, all transformations targeted to the predicate representation must either be foregone or delayed. If these transformations are delayed, much more complexity is added to a scheduler which must already consider many issues including control speculation, data speculation, and register pressure to achieve desirable code performance. Additionally, delaying only some optimizations until schedule time creates a phase ordering which can cause severe difficulties for the compiler. Generally, most transforms have profound effects on one another and must be repeatedly applied in turn to achieve desirable results. For example, a transformation, such as control height reduction [12], may subsequently expose a critical data dependence edge that should be broken by expression reformulation. However, until the control dependence height is reduced, there is no profitability to breaking the data dependence edge, so the compiler will not apply the transform. This is especially true since expression reformulation has a cost in terms of added instructions. The net result of the schedule-time framework is a restriction in the use of the predicate representation which limits the effectiveness of back-end optimizations.

Given that if-conversion at schedule time limits the use of the predicate representation for optimization and given that if-conversion at an early stage is limited in its ability to estimate the final code characteristics, it is logical to look to an alternative compilation framework. This paper proposes 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 backend compilation procedure. Then at schedule time, the compiler adjusts the final amount of predication to efficiently match the target architecture. The compilation framework, shown in Figure 4, 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, adjustment of hyperblocks during scheduling, is proposed in this work and has been termed *partial reverse if-conversion*.

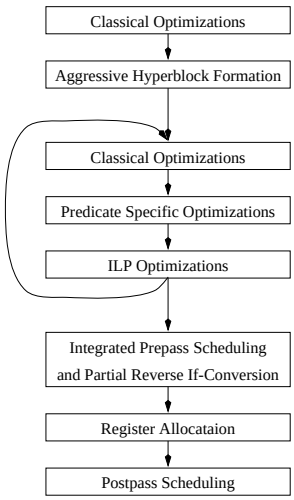


Figure 4: Phase ordering diagram for the compilation framework.

The first phase of the compilation framework is to aggressively perform hyperblock formation. The hyperblock former does not need to exactly compute 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 their benefits. 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 criteria that is still enforced in the first phase of hyperblock formation is avoiding paths with hazards. 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 is to adjust 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 [16]. Reverse if-conversion was originally proposed as the inverse process to if-conversion. Branching code that contains no predicates is generated from a block of predicated code. This allows code to be compiled using a predicate representation, but 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 are profitable to overlap 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 will insert 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 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 repeatedly applied to the same hyperblock until the resulting code is desirable.

The strategy used for this compilation framework can be viewed analogously 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 predictability 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. 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. Each of these steps is discussed in turn.

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.

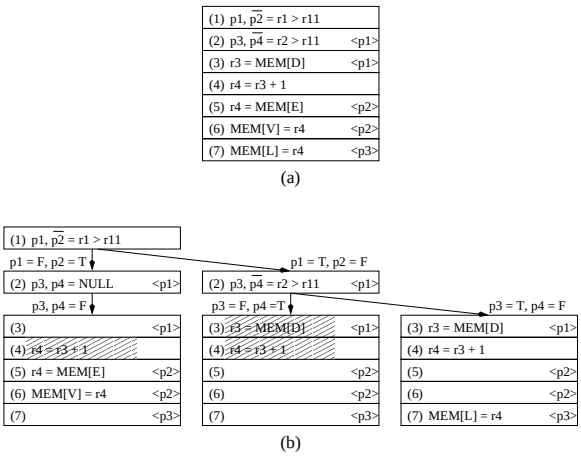


Figure 5: Predicate flow graph with partial dead code elimination given that r3 and r4 are not live out of this region.

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 5 shows a predicated code segment and its predicate flow graph. The predicate flow graph shown in Figure 5b is created in the following manner. The first instruction in Figure 5a 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 defining 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 [17][18]. 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

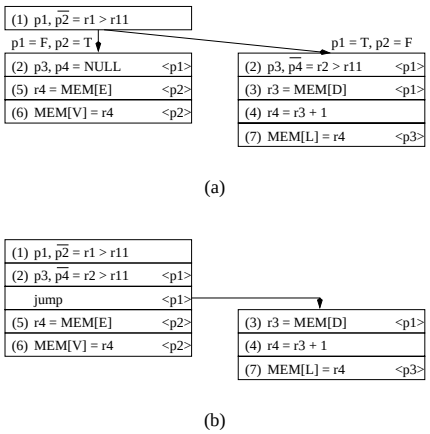


Figure 6: Predicate flow graph (a) and a partial reverse if-conversion of predicate p1 located after instructions 1 and 2 (b).

potential path explosion problems in the PFG. This is because the number of paths in a PFG is exponentially proportional to the number of independent predicates whose live ranges overlap. Fortunately, this does not happen in practice until 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 predicate 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 5, 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 only indicates that these instructions are dead under certain predicate conditions, this process is termed *predicate partial dead code removal* and is related to other types of partial dead code removal [19]. The result of partial dead code removal indicates that instructions 3 and 4 would generate correct code and would not execute 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 6 is used to aid this discussion.

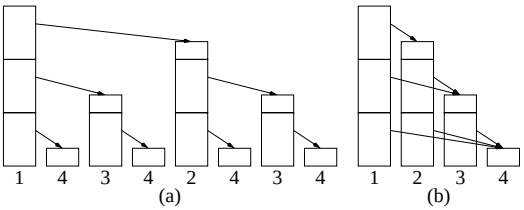


Figure 7: Simple code size reduction on multiple partial reverse if-conversions applied to an unrolled loop. Each square represents an unroll of the original loop.

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 6b 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 only with respect to the predicate which is being reverse if-converted. This keeps the number of paths under control since a the single predicate PFG can contain no more than two paths. Figure 6a 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 all re-entries into the hyperblock effectively divide the original hyperblock into two smaller hyperblocks.

The advantage of branching back into the original hyperblock is a large 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 large 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 7a shows the original result of repeated reverse if-conversions on an unrolled loop. Figure 7b 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. To remove all unnecessary code growth, a method which jumps back into the hyperblock at an opportune location needs to be created.

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 chosen 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 if-convert. An operation-based scheduler may yield one type of heuristic while a list scheduler may yield another. The policy determining how to reverse if-convert presented here was designed to work within the context of an existing list scheduler. The algorithm with this policy integrated into the list scheduler is shown in Figure 8.

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

```
1  Initialize ready_priority_queue;
2  ric_queue = NULL;
3  cycle = 0;
4  num_unsched = Number of operations;
5  sched_no_ric = Compute_dynamic_cycles_for_hyperblock;
   // Each trip through this loop is a new cycle
6  WHILE num_unsched != 0 DO
     // Handle reverse if-converting branches first
7    FOREACH ric_op IN ric_queue DO
8      IF Schedule_Op(ric_op, cycle) THEN
9        Compute location for each unscheduled op;
10       sched_ric_taken = Compute_dynamic_cycles_in_ric_taken_path;
11       sched_ric_hb = Compute_dynamic_cycles_in_ric_hyperblock;
12       mipred_ric = Estimate_ric_mispreds * miss_penalty;
13       ric_cycles = sched_ric_hb + sched_ric_taken;
14       ric_cycles = ric_cycles + mispred_ric;
15       IF (sched_no_ric > ric_cycles) THEN
16         sched_no_ric = sched_ric_hb;
17         Place all ops in their no_ric schedule location;
18       ELSE
19         Unschedule_OP(ric_op);
20       Remove ric_op from ric_queue;
     // Then handle regular operations
21   FOREACH regular_op IN ready_priority_queue DO
22     IF Schedule_Op(regular_op, cycle) THEN
23       Remove regular_op from ready_priority_queue;
24       num_unsched = num_unsched - 1;
25       IF Is_Predicate_Define(regular_op) THEN
26         Add reverse if-converting branch to ric_queue;
27   cycle = cycle + 1;
```

Figure 8: An algorithm incorporating partial reverse if-conversion into a list scheduler

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 defining instruction.[2] 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 seperated. The code size can be reduced if the reverse if-converting branch is delayed. Depending on the characteristics

[2] There exist machines where the placement of a branch a number of cycles after the computation of its condition removes all of its mispredictions [20]. In these machines, there are two locations which should be considered, immediately after the predicate defining instruction and in the cycle in which the branch mispredictions are eliminated.

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 characteristecs 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 top-down fashion, in the order in which the predicate defines are scheduled. This heuristic is used in the algorithm shown in Figure 8. This has the desirable effect of making the reverse if-conversion process fit seemlessly into a list scheduler. It is 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 to evaluate it 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 comparision is computed by lines 9 through 15 in Figure 8.

While the cost savings due to the heuristic is quite significant, $3p$ schedules for more complicated machine models can still be quite costly. 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 top-down property of the list scheduler itself. At the point each reverse if-conversion is considered, all previous instructions have been scheduled in their final location by lines 8 or 22 in Figure 8. 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 source of reuse takes the total schedules computed down to $2p + 1$ with each schedule only considering the unscheduled instructions at each point due to the list scheduling effect. This reuse is implemented in Figure 8 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. 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 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 by 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.

5 Experimental Results

This section presents an experimental evaluation of the partial reverse if-conversion framework.

5.1 Methodology

The partial reverse if-conversion techniques described in this paper have been implemented in the second generation instruction scheduler of the IMPACT compiler. The 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 chosen. The machine modeled is a 4-issue superscalar processor with in-order execution 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. The instruction set contains a set of 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. 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

Figures 9 and 10 compare the performance of the traditional hyperblock compilation framework and the new compilation framework with partial reverse if-conversion. The hyperblocks formed in these graphs represent those currently formed by the IMPACT compiler's hyperblock formation heuristic for the target machine. These same hyperblocks were also used as input to the partial reverse if-converter. The results obtained are therefore conservative since more aggressive hyperblocks would create the potential for better results. The bars represent the speedup achieved by these methods relative to superblock compilation. This is computed as follows: $superblock_cycles/technique_cycles$. Superblock compilation performance is chosen as the base because it represents the best possible performance currently obtainable by the IMPACT compiler without predication [21].

Figure 9 shows the performance of the hyperblock and partial reverse if-conversion compilation frameworks assuming perfect branch prediction. 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 half of the benchmarks. This dichotomy is a common problem experienced with hyperblocks and indicates that hyperblocks can do well, but often performance is victim to poor hyperblock selection.

In all cases, partial reverse if-conversion improved upon or

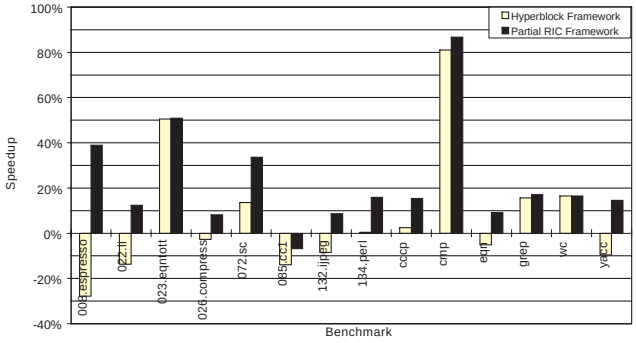


Figure 9: Performance increase over superblock exhibited by the hyperblock and partial reverse if-conversion frameworks with no misprediction penalty.

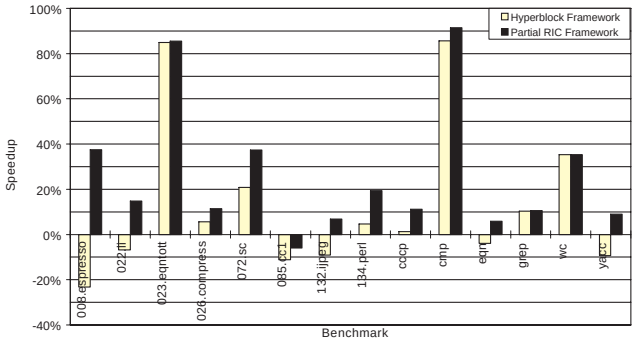


Figure 10: Performance increase over superblock exhibited by the hyperblock and partial reverse if-conversion frameworks with a four cycle misprediction penalty.

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 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 two of the benchmarks more closely. The worst performing benchmark is *085.cc1*, for which 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 defining instruction. Un-

fortunately, 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*.

One of the best performing benchmarks was *072.sc*. For this program, hyperblock compilation increased performance by a fair margin, but the partial reverse if-conversion increased this gain substantially. Most of *072.sc*'s performance gain was achieved by transforming a single function *_update*. This function with superblock compilation executes in 25.6 million cycles. However, the schedule is rather sparse 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 to efficiently utilize 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.

Figure 10 shows the performance of the benchmarks in the same manner as Figure 9 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 that of superblock. When applying control flow transformations in the predicated representation, such as branch combining, 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.

The static code size exhibited by using the hyperblock and partial reverse if-conversion compilation frameworks with respect to the superblock techniques is presented in Figure 11. 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. Whereas 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 hyperblock. 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 earlier. 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.

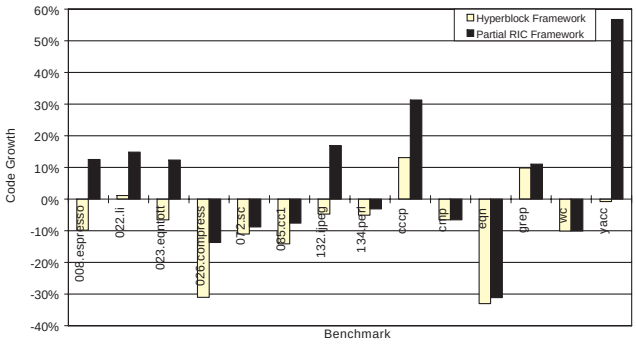


Figure 11: Relative static code size exhibited by the hyperblock and partial reverse if-conversion frameworks compared with superblock.

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.

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 relatively 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 11).

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 take full advantage of the predicate representation to apply 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 are 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 to undo 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 Gyllenhaal, Teresa Johnson, Brian Deitrich, Daniel Connors, John Sias, Kevin Crozier 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] 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.

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

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

[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, and R. A. Bringmann, "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] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.