PROFILE-DIRECTED IF-CONVERSION IN SUPERSCALAR MICROPROCESSORS

BY

ERIC J. ZIMMERMAN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

Acknowledgements

I would like to extend a sincere "thank you" to my advisor, Prof. Craig Zilles, for his many hours of patient instruction, numerous helpful suggestions, and overall coaching in the discipline of computer science research. His sense of humor and commitment to family life is an encouragement to me as I begin my own career. Also, I appreciate his willingness to support me as a research assistant during this project.

I would like to thank the many students I've worked with over the past two years. I especially appreciate the friendship of my research group co-workers, Naveen, Pierre, Nicholas, Lee, David, and Charles, who always made time for this M.S. student from the other cube. I am very thankful for the friendship of officemates Alex and Vibhore. Even though Aunt Sonya's is no more, I will join you guys for lunch anytime. I especially owe my gratitude to Andrew Lenharth and Chris Lattner for their hard work on LLVM to enable this project. Also, thanks to Steven and the YG girls for those late-night snacks, and to Chris for telling me plainly to "just get it done."

Finally, I thank my family and grandparents for their encouragement and prayers as I faced the challenge of graduate school. Thank you for reminding me in word and in action of the truth embodied in Hebrews 12:1-2.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

Introduction

Conventional superscalar microprocessors have improved tremendously in performance, but still meet several limitations in achieving maximum performance. First, a load that misses in the data cache will wait many cycles for the value to arrive. Along with its dependent instructions, it will tie up critical resources including physical registers and scheduler entries. Even though there is often independent work that can be overlapped with the memory latency, many times the processor must stall due to a blocked reorder buffer (ROB). Karkhanis and Smith report about 90% of cache misses to memory result in a blocked ROB [1]. Second, accurate branch prediction remains essential to keeping superscalar pipelines running efficiently. Modern branch predictors are very accurate for most branches, but long pipelines result in many instructions having to be squashed in the unfortunate case of a misspeculation. Aragon et al. report that “an average 8-wide processor spends 47% of its total cycles fetching wrong-path instructions” [2].

Proposed future architectures like *Continual Flow Pipelines* (CFP) [3] effectively provide a large instruction window that is able to tolerate long-latency cache misses. CFP introduces a non-blocking register file and scheduler that off-loads miss-dependent instructions to a *Slice Processing Unit* while the memory request is satisfied. As a result of this design, a single cache miss will not likely cause a structural hazard, but instead, issuing can proceed with instructions that are independent of the cache miss. This allows a CFP machine to service multiple cache misses in parallel, achieving the desirable property of *memory-level parallelism* (MLP) [4]. When processors with a high degree of memory-level parallelism eliminate pipeline stalls due to cache misses, branch misspeculations will become a more acute limiter of system performance.

Branch misspeculations are also responsible for significant energy and power costs in modern microprocessors. Before a misprediction is resolved, many wrong-path instructions are fetched, issued, and executed. These wasted instructions offer some benefit for prefetching future data values and instructions, but the instructions themselves are eventually flushed from the pipeline. Energy consumption of a processor is in part proportional to the number of instructions fetched, so wrong-path instructions represent a significant waste of power and energy. Indeed, Aragon et al. state "branch mispredictions are responsible for around 28% of the power dissipated by a typical processor due to the useless activities performed by instructions that are squashed" [2].

We believe that CFP-style processors' heightened misspeculation penalty in terms of power and performance can be reduced through predicated execution.

Predicated execution is a mechanism by which microprocessors fetch and execute instructions but conditionally commit the results based on a predicate value that evaluates to true or false. Predicated execution is often used to eliminate conditional branches in a process called *if-conversion.* Predicated execution is commonly studied in the context of EPIC architectures where it is largely utilized to increase scheduling scope for statically scheduled machines like Intel's Itanium 2 [5]. Dynamic predication, proposed by Klauser, et al. is an interesting exception for superscalar architectures because it requires no support in the instruction set architecture (ISA) for predicated instructions [6]. Instead, the predicated execution is performed dynamically by the register renamer and pipeline back-end. This method is limited to the if-conversion of *simple branch hammocks*, which are used to implement if-then-else statements.

In this thesis, our goal is to identify the control structures associated with frequently mispredicting branches in a modern out-of-order microprocessor. In Chapter 2, we classify these frequently mispredicting branches into several categories, and suggest possible ways to improve their predictability or remove the need for speculation entirely through predication. In Chapter 3, we suggest a series of profile-guided heuristics for selecting favorable branches to if-convert in a modern out-of-order processor. In Chapter 4, we implement several profile-directed compiler transformations to illustrate the predication mechanisms and our selection heuristics for deciding when to employ them. Using microprocessor simulations, we measure the effectiveness of the transformations in reducing the misspeculation penalty in terms of energy and performance. In Chapter 5, we summarize our results and discuss how predicated execution fits with future trends in microprocessor design.

CHAPTER 2

A Taxonomy of Mispredicting Branches

The majority of branch mispredictions are caused by a small fraction of branches, the hard-to-predict branches [7]. We compiled a list of branches with the greatest occurrence of mispredictions for several benchmarks from the Spec2000 integer benchmark suite. We then observed program control flow graphs and defined a set of common branch categories. We classified the mispredicting branches according to these categories to determine how the mispredictions are distributed. Figure 2.1 shows the distribution of branch mispredictions by category. In this chapter, we describe the categories in more detail. We also discuss ways to reduce the cost of mispredicting these branches with predicated execution.

In this thesis, the branch structures we target for if-conversion are: simple hammocks, nested hammocks, simple loop backedges (and their guarding branches), and conjunctive branches. While Figure 2.1 shows a mixed representation for these branches among the Spec2000 integer benchmarks, these branch structures contribute to over fifty percent of the overall mispredictions, on average. Also, most of the benchmarks in Figure 2.1 have fewer than 10% of their branch instructions contributing to over 99% of the branch mispredictions. We will focus on these hard-to-predict branches in order to measure the effects of predicated execution on performance and energy.

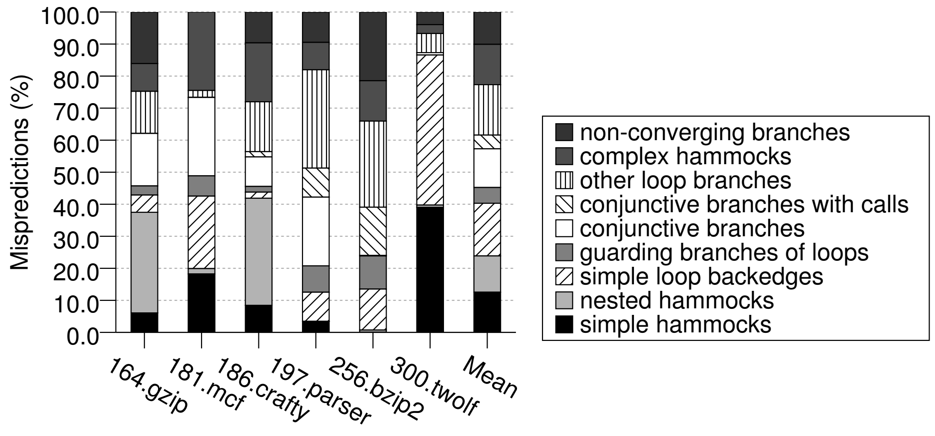

Figure 2.1 Branch mispredictions by category
We considered six Spec2000 integer benchmarks, as described in Section 4.1.

2.1 Hammocks

Simple hammocks represent if-then and if-then-else statements and are easy to if-convert with instruction set architecture (ISA) support or by using dynamic predication. *Half hammocks* implement if-then statements and *full hammocks* implement if-then-else statements. Simple hammocks may not contain function calls or system instructions, which generally prohibit predicated execution. Figure 2.2 shows an example of if-conversion for three hammocks.

Nested hammocks are composed of simple hammocks or other nested hammocks. Predicated execution is more costly in the case of nested hammocks because there are more than two paths to follow. It is not necessary to if-convert every branch in a nested hammock. Inner branches may be highly biased and better left for speculation. However, if outer hammock branches are if-converted, the bias of the inner branch may change unfavorably, as shown in Section 4.3.2.

Complex hammocks are hammocks that include additional control structure before reconverging at the join point. Nested hammocks are a special case of complex hammocks, but we distinguish between these cases because complex hammocks in general may contain an internal loop or function call that makes if-conversion difficult to perform.

Non-converging branches include hammocks that contain a return instruction at the end of one of the control paths. Without a confluence of control paths, it is impossible to if-convert the branch.

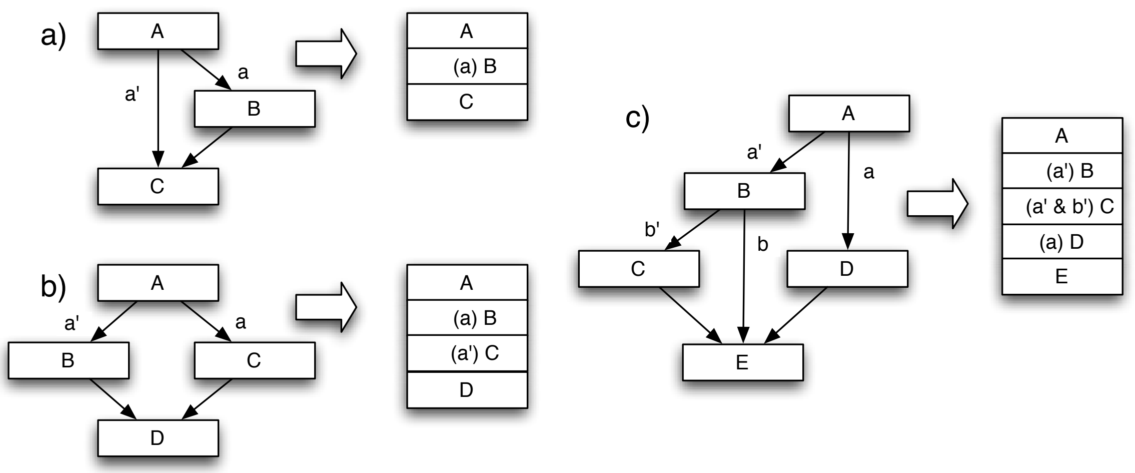

Figure 2.2 If-conversion example for a half hammock (a), full hammock (b), and nested hammock (c)

2.2 Loops

Backedge branches of loops that iterate many times are generally easy to predict, and loops that iterate a constant number of times can be completely unrolled or run with zero-overhead loop instructions like those found in many embedded architectures. However, if a loop executes for a small number of iterations and the trip count isn't predictable, the backedge branch will be frequently mispredicted. The misprediction penalty of these loops can often be lowered by combining *loop peeling* with predication. Loop peeling is a technique similar to loop unrolling by which the body of the loop is duplicated and specialized to execute the first or last iterations of the loop. Predicating the peeled iterations eliminates the need to speculate on these hard-to-predict backedge branches but comes with an overhead cost of having to fetch and execute unused iterations of the loop body. Figure 2.3 illustrates a peeled loop where the second and third iterations are executed unconditionally with the liveout values predicated on the union of the earlier exit conditions.

As in the case of hammocks, loops must be free of function calls and system instructions for predication to be a practical option. Loop peeling works best when the loop body is small with little or no internal control flow, because there will be a lower penalty for an unused iteration. The original version of the loop is kept as a successor to the final peeled copy for the case when trip count exceeds the number of peeled iterations. The branch category labeled *simple loop backedges* in Figure 2.1 represents loops with a single backedge branch and no internal function calls. Our loop peeling study targets this class of backedge branches.

Most loops have a *guarding branch* that skips over the loop body when the loop should not be entered. A guarding branch can be difficult to predict in situations where a loop varies between entering and skipping over a loop. If the guarding branch is difficult to predict, it can be eliminated by peeling the first loop iteration and if-converting the guarding branch. On the other hand, if the guarding branch is highly biased, it may be better left for speculation.

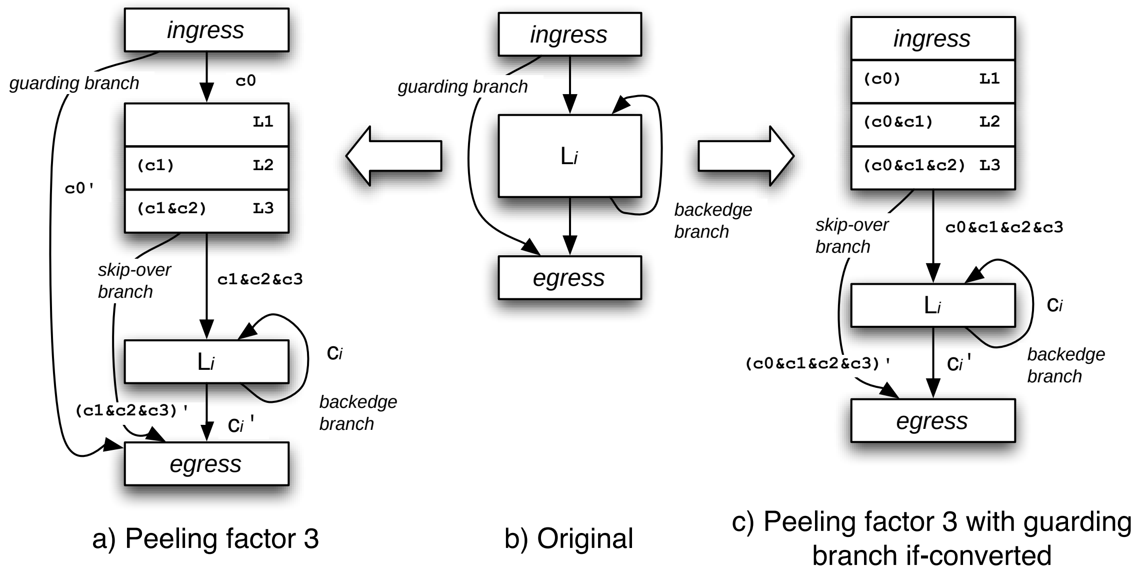

Figure 2.3 Simple loop with guarding branch (b)
Three iterations are peeled in addition to if-converting the guarding branch (c) or without if-converting the guarding branch (a).

Loops are often nested with arbitrary breaks and continues. In general, it is impractical to peel and predicate iterations from an outer loop. Branches that implement breaks and continues can sometimes be merged with the backedge to improve their predictability as described in Section 2.3.

The category *other loop branches* in Figure 2.1 represents branches from outer loops and from inner loops containing calls.

2.3 Conjunctive Branches

Conjunctive branch is a term given to a group of branches commonly used to implement logical AND or logical OR conditions. Sometimes the conditions in a conjunctive branch pair are individually hard to predict but easier to predict as a unit. Therefore, it may be beneficial to coalesce these conditions using comparison and logical operators. That is, the first branch is changed to point unconditionally to the second block, and the second block is changed to branch on the aggregate condition, as shown in Figure 2.4a. Care must be taken when there are side-effects performed with the comparisons. The liveouts from the side-effect operations in the second block should be predicated to ensure correct execution. Merging the two branches can create a hard-to-predict branch in place of two easier to predict branches, so this transformation should not be performed in every situation.

We define the common target block as the *fall-out block*. The fall-out block represents the true destination for a logical OR condition or the false destination for a logical AND condition. We define the second branch's other successor as the *advance block*.

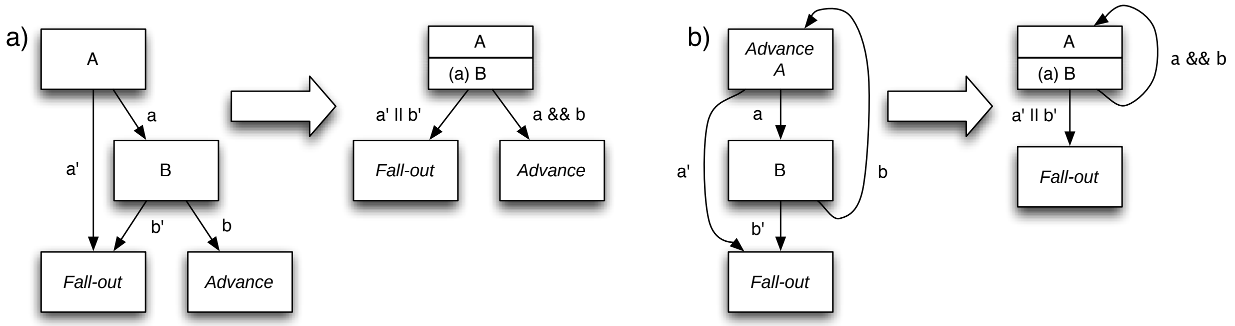

Figure 2.4 Conjunctive branch pair and breaking loop variant

A common variation of a conjunctive branch pair is a loop with an early exit condition. If the early exit branch is difficult to predict, it may be advantageous to merge this condition with the loop backedge as shown in Figure 2.4b. As with traditional conjunctive branch merging, it is necessary to predicate all liveouts defined after the early exit branch on the inverse of the early exit condition.

Conjunctive branch pairs that contain a function call as part of the second basic block occur frequently enough to be identified as a separate category in Figure 2.1. Branch-merging may not be applied if there are memory or register side effects performed as a result of the function call.

CHAPTER 3

If-Conversion Heuristics

We attempt to create a set of heuristics for identifying profitable hammock predication, loop peeling, and conjunctive branch merging opportunities. We assume the availability of profile data to guide our selections. Previous work contains an example of a profile-based if-conversion heuristic, where the authors consider the trade-off between misspeculated cycles saved and lengthened schedule height for a predicated region [8]. This heuristic is primarily focused on the schedule height of an in-order processor like the Itanium. Other work has presented a profile-directed if-conversion heuristic for out-of-order processors [9]. The authors developed a profile-based heuristic to be combined with hammock-size and predictability methods for selecting profitable if-conversion opportunities. They found the profile-directed heuristic outperformed the other methods in terms of reducing mispredictions and overall cycles.

In this chapter, we extend this prior work to develop a more general if-conversion heuristic for out-of-order processors, using cycle estimates of the fetching and execution latencies imposed by predication. We do not limit our consideration to hammock structures, but extend our reasoning to consider loop peeling and conjunctive branch coalescing. We also consider the change in net predictability brought about by the branch transformations, where applicable. In contrast to the static instruction scheduling required by an in-order processor, instructions in an out-of-order processor are scheduled dynamically, making the relationship between performance and the compiler schedule less clear. We model two primary factors of if-converting code: 1) the added cost *to fetch* wrong-path instructions that will be nullified, and 2) the added cost *to execute* wrong-path temporaries that will be replaced with a correct-path liveout. Both factors potentially lengthen the critical path to negate the benefit of cycles recovered from branch mispredictions.

In Section 3.1, we describe predication mechanisms commonly implemented in out-of-order processors and how this influences the cost of predicated execution. In Sections 3.2, 3.3, and 3.4, we apply critical-path reasoning to develop heuristics for predicating hammocks, peeling loop iterations, and unifying conjunctive branches.

3.1 Predication Mechanisms in Out-of-order Processors

Predicated instruction set architectures support either full or partial predication [10]. *Full predication* indicates most instructions contain a source operand specifying a predicate value. If the predicate is evaluated as true, the instruction will commit its result; however, a false predicate causes the the instruction to be nullified without changing the program state. The IA64 architecture, for example, supports full predication with sixty-four single-bit predicate registers that are used to guard the execution of most instructions. Full predication is normally implemented on in-order processors due to the complexity of tracking conditional definitions in the register renamer of an out-of-order machine [8].

Partial predication delivers support for predication in the form of conditional move (cmove) or select instructions. Depending on its predicate specifier, a select instruction chooses one of two source operands to write to its destination. A conditional move instruction has a single source operand and a predicate operand. If the predicate is true, the cmove will copy the source register value to the destination register. If the predicate is false, the cmove instruction behaves as a no-op. The Alpha ISA is an example of a partially-predicated ISA that supports conditional moves.

In this study we use the Alpha ISA as our compiler target and simulation ISA. Using conditional moves, we are able to conditionally execute most machine operations. When generating code, we perform a compiler analysis to determine which instructions generate liveout values from a block. Each liveout value in a predicated code region is first written to a temporary register. If the predicate is true, a cmove instruction copies the temporary value to the correct liveout register. Non-liveout values do not need to be predicated. When deciding if predication is profitable in a partially-predicated environment, we must account for the overhead of conditional move instructions inserted to predicate liveout values.

Store instructions are also considered to be liveout values and can also be predicated using conditional moves. The cmove guards the store's effective address. If the store is executed with a false predicate, the cmove will replace the effective address with a safe memory location where the value will be stored without affecting program state. In our simulation environment, we chose this location to be the null address. In general, it is important for the processor to support non-excepting instructions. Predicated code potentially accesses illegal memory addresses, divides by zero, and overflows value bounds. Therefore, these events should not terminate execution when caused by an instruction with a false predicate.

3.2 Hammock Selection

When deciding whether a hammock should be predicated, we consider whether the benefit of reducing mispredictions outweighs the cost of added instructions and critical path elongation. The average cycle cost of *speculating* on a hammock branch is simply the probability of mispredicting the branch times the misprediction latency of the machine. Obviously, there is no benefit to predicating a branch that is seldom misspeculated.

On the other hand, there are several factors to consider in estimating the average cycle cost of *predicating* a simple hammock:

1. When instructions following the hammock are independent of hammock liveouts, the primary cost of predication is the greater latency *to fetch* these independent instructions. We estimate this latency by averaging the number of overhead instructions on the wrong path over all hammock executions and dividing by the machine's instruction fetching width.

$$overhead_{fetch} = (average\ wrong\_path\ length + average\ inserted\ cmoves)/IF\_width \tag{3.1}$$

2. When hammock contexts are imbalanced and instructions following the hammock depend on one or more hammock liveouts, the cost *to execute* the wrong path is dominant. If these

liveouts are defined on the shorter hammock path, predication lengthens the time until the value becomes available by requiring execution of wrong-path instructions and a conditional move. Figure 3.1 illustrates this lengthened dependency height. If we assume the hammock liveout definition is on the critical path, we can approximate the execution overhead imposed by predicated execution by:

$$overhead_{execute} = \max(average\ wrong\_path\ height - average\ correct\_path\ height, 0) + 1 \quad (3.2)$$

Wrong_path height represents the height of the longest dependency chain for a liveout definition on the wrong path. Similarly, *correct_path height* represents the height of the longest dependency chain for a liveout definition on the correct path. These values must be weighted by execution frequency and subtracted to find the overhead cycle cost to execute the wrong-path instructions. One instruction is added to the liveout height difference to represent the conditional move instruction used to specify the correct liveout value.

```
if (node) {
  parent = node->parent;
  uncle = parent->brother;
  age = uncle->age;
}
else {
  age = self->age;
  age = age + 1;
}
```

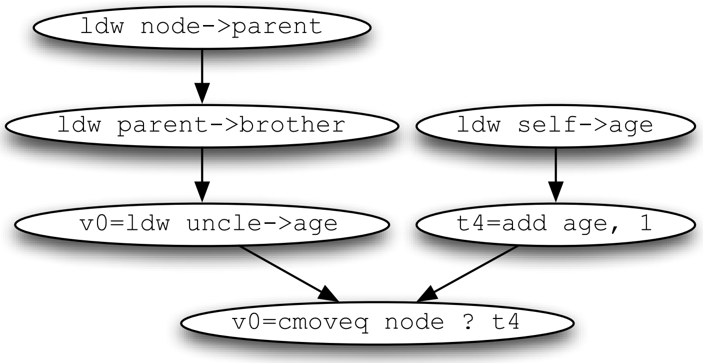

Figure 3.1 Predication lengthens liveout dependency chain height
If the *then* clause is the correct path, predication lengthens the dependency height by one. If the *else* clause is the correct path, predication lengthens the dependency height by two.

We conservatively estimate the predication cost of a hammock as the maximum of the costs defined in equations 3.1 and 3.2. When the speculation cost exceeds the predication cost, the hammock should be marked for predication. Predication can offer additional benefits not modeled in our selection heuristic, including exposing common subexpressions for reduction. Also, if a hammock is small, predicating it may allow the processor to fetch the entire hammock body contiguously for a performance gain. We found in our experiments that $overhead_{fetch}$ tends to dominate $overhead_{execute}$, except in certain instances of small half hammocks.

Nested hammock execution is influenced by the same characteristics as simple hammock execution. That is, nested hammock liveouts may not be used immediately, causing the number of instructions fetched on the wrong paths to delay the fetching of more critical instructions following the hammock. In this case, the average wrong-path length must be weighted according to the execution frequencies of all paths through the nested hammock. Alternately, nested hammock liveouts may be used soon after they are defined, causing any delay in production of the liveout to stall the execution of the liveout's consumer. The dependence height of the correct-path liveout must be compared to the liveout's dependence height on all other paths, weighted by their execution frequencies.

Nested hammocks have the added characteristic of containing internal branches that may or may not be predicated. Speculating on easy-to-predict inner branches may reduce the instruction overhead of if-conversion. On the other hand, the presence of inner branches may restrict the freedom of the static code scheduler to efficiently place and combine instructions. We investigate this trade-off in Section 4.3.2.

3.3 Loop Selection

As with hammock predication, determining the optimal number of loop peels involves balancing a trade-off between reducing the number of branch mispredictions and lengthening the critical path by requiring the processor to fetch and execute unneeded wrong-path instructions. We approximate the benefit of peeling n iterations of a loop by:

$$benefit(n) = misprediction\_latency \times \left(\sum_{i=0}^{n} misps_i - \min\left(\sum_{i=0}^{n} tripcounts_i, \sum_{i=n+1}^{\infty} tripcounts_i\right)\right) \quad (3.3)$$

$misps_i$ is the number of mispredictions occurring on trip i of the loop, and $tripcounts_i$ is the number of times the loop exited on the i^{th} iteration. This expression measures the number of backedge mispredictions that are saved through peeling. From this savings, we must deduct the number of mispredictions added to the skip-over branch that skips over the loop body when no iterations are required beyond the peeled iterations. We estimate the number of mispredictions for this branch by assuming it is correlated to the branch's bias. Loop peeling may not be profitable in cases where it introduces a hard-to-predict skip-over branch.

The overhead of peeling n iterations from a loop is represented by:

$$\begin{aligned} overhead_{fetch}(n) &= \frac{peel\_size}{IF\_width} \times \sum_{i=0}^{n-1}(n-i) \times tripcounts_i && (3.4)\\ overhead_{execute}(n) &= (max\_liveout\_height + 1) \times \sum_{i=0}^{n-1}(n-i) \times tripcounts_i && (3.5)\\ overhead(n) &= \max(overhead_{fetch}(n), overhead_{execute}(n)) && (3.6) \end{aligned}$$

$peel\_size$ is the number of instructions that comprise a peeled iteration. In many cases it is overly conservative to estimate this value as the size of the loop body due to the effects of constant propagation and dead code elimination on the peeled code. IF_width represents the number of instructions fetched per cycle. $max\_liveout\_height$ is the length of the longest dependency chain for a liveout definition in the loop body, with one extra link for the guarding conditional move instruction that is added to each peeled iteration. $tripcounts_i$ is the number of times the loop exited on the i^{th} iteration. Note that exits after few iterations (i.e., low trip counts) become more costly in terms of fetch and execution latency as the peeling factor n is increased. $tripcounts_0$ is defined as the number of times control skips over the loop via the guarding branch, which contributes to the overhead costs only if the guarding branch is if-converted. If the guarding branch is not if-converted, the sums in equations 3.3, 3.4, and 3.5 should begin at one instead of zero.

We can determine the optimal number of loop iterations to peel by finding i that satisfies $\max_i(benefit(i) - overhead(i))$. This selection heuristic does not take into account the positive side-effects of wrong-path prefetching[11]. For example, a mispredicted hammock branch within a loop will often prefetch data values for future loop iterations of the current loop or of a subsequent loop. In determining the optimal number of iterations to peel, our heuristic does not consider the code-expansion pressure placed on the instruction cache, although we expect this to be low since generally only small loops are viable for peeling.

3.4 Conjunctive Branch Selection

When deciding whether to coalesce the branches of a conjunctive branch pair, it is vital to know whether the predictability of the combined branch improves on the predictability of the pair. An improvement in predictability may result from an improvement in branch bias, an improvement in branch correlation, or both. We continue to consider two sources of overhead latency: the added instruction fetching burden and the lengthened dependency chain required for predicated execution.

In the conjunctive branch in Figure 3.2, block A is executed unconditionally and does not contribute to an overhead cost. Block B presents an instruction fetch overhead only when branch A would have proceeded to the fall-out block. The probability of this event is $p_{\mathit{fall\_out}}(a)$. $\mathit{cmoves}(B)$ is the number of conditional move instructions required to predicate block B's liveout values. The conditional moves present an overhead fetching cost regardless of branch A's outcome. There may be an overhead execution cost for the combined branch if branch A would have proceeded to the fall-out block. B's side-effect instructions may have a greater dependency height than A's side effect instructions. As in the previous heuristics, we must add one for the cmove that conditionally defines the result. We estimate the overhead of merging branches A and B as the maximum of these overhead costs.

$$\mathit{merge\_overhead}_{\mathit{fetch}} = \frac{\mathit{size}(B)p_{\mathit{fall\_out}}(a) + \mathit{cmoves}(B)}{\mathit{IF\_width}} \tag{3.7}$$

$$\mathit{merge\_overhead}_{\mathit{execute}} = \max(\mathit{max\_height}(B) - \mathit{max\_height}(A), \mathit{0})p_{\mathit{fall\_out}}(a) + 1 \tag{3.8}$$

$$\mathit{merge\_overhead} = \max(\mathit{merge\_overhead}_{\mathit{fetch}}, \mathit{merge\_overhead}_{\mathit{execute}}) \tag{3.9}$$

Intuitively, the conjunctive pair should be combined for a performance gain when the number of cycles saved by fewer misspeculations outweighs the overhead of fetching and executing unnecessary instructions. This is expressed more formally in equation 3.10. $p(a)$ is the probability that control reaches block B. $p_{misp}(x)$ represents the probability that branch x is mispredicted.

$$\begin{aligned} & \textit{misprediction\_latency} \times (p_{misp}(a) + p(a)p_{misp}(b)) > \\ & \textit{misprediction\_latency} \times p_{misp}(ab) + \textit{merge\_overhead} \end{aligned} \tag{3.10}$$

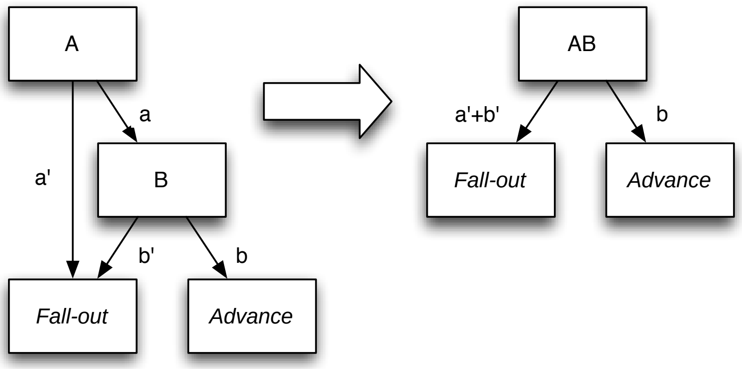

Figure 3.2 Improving branch bias by combining branches

Knowing when and how merging conjunctive branches will improve the misprediction rate, p_{misp}, is less clear. Difficult-to-predict branches often have misprediction rates that correspond to their branch biases[1]. Clearly, this rate will not exceed 50% in the long run, or the predictor could choose the opposite direction for greater accuracy.

If we consider a conjunctive pair of hard-to-predict branches A and B in Figure 3.2, when will merging the branches improve the net bias? Branch A is taken a' times out of $a + a'$ and branch B is taken b' times out of $b + b'$. Assuming branch A and B are predicted according to branch bias, the total number of mispredictions will be $\min(a, a') + \min(b, b')$. Assuming the combined branch is predicted according to its bias, it will be mispredicted $\min(b, a' + b')$ times. Therefore, branch combining in this scenario offers a misprediction savings of:

[1]A concrete example of conjunctive branch pairs with this property appears in Section 4.5. In all intervals except interval 2, each branch in this structure is predicted with an accuracy that is within 5% of the branch bias. For these branches, improving the net bias translates to a savings in the number of mispredictions. The exception in interval 2 occurs when branch X is mispredicted only 2.5% of the time while it is taken 18.9% of the time.

$$\textit{Misprediction savings} = \min(a, a') + \min(b, b') - \min(b, a' + b') \tag{3.11}$$

This value is positive as long as $a' + b' > (a + a')/2$. That is, a conjunctive branch pair must have a net bias toward the fall-out block for branch combining to expose this bias for a gain in predictability. It is important to consider the added instruction overhead and liveout delay caused by the branch-merging transformation, which may offset the branch misprediction savings. This can be determined from equation 3.10, where $p_{misp}(ab)$ becomes $\min(b, a' + b')/(a + a')$.

It is also important to consider the opportunity to improve a branch's predictability through correlation for conjunctive branch pairs. Figure 3.3 shows an example from benchmark `254.gap` where branch 3 is the second branch in a conjunctive pair. The condition of branch 3 matches the condition of earlier branch 1. The heavy lines trace the dominant path through this region, which indicates that control frequently passes over branch 3. Even though branch 2 is biased toward the fall-out block, it is still responsible for many mispredictions. Combining the conjunctive pair in this case will expose the hidden branch correlation between branches 1 and 3 for a net gain in predictability.

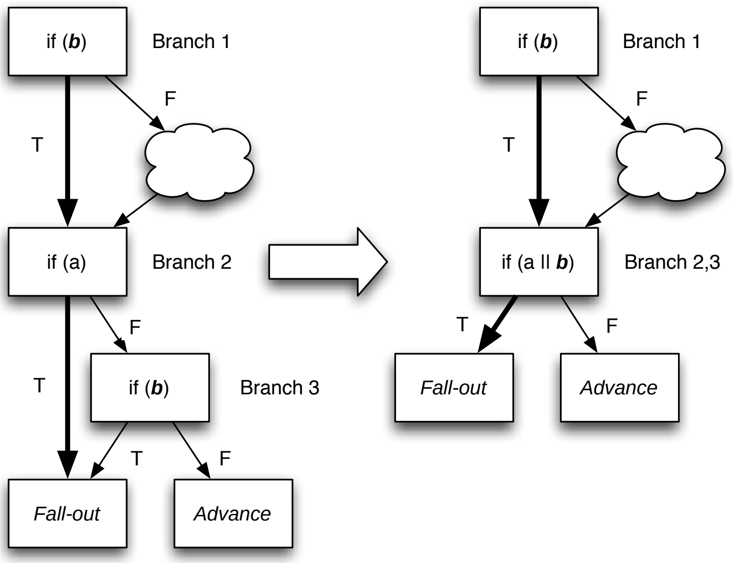

Figure 3.3 Improving branch correlation by combining branches

CHAPTER 4

Evaluation

To investigate the impact of predicated execution on out-of-order processors, we created a "standard" feedback-directed optimization (FDO) system, which means we collect profile data to direct the code transformations of a traditional compiler, rather than an "online" FDO system, which integrates profiling and re-compilation functions and performs them at runtime. We describe our system in Section 4.1. In Sections 4.3, 4.4, and 4.5, we examine the effectiveness of our profile-directed transformations on reducing the misprediction penalties associated with hammocks, loops, and conjunctive branches in certain Spec2000 integer benchmarks.

4.1 Experimental Setup

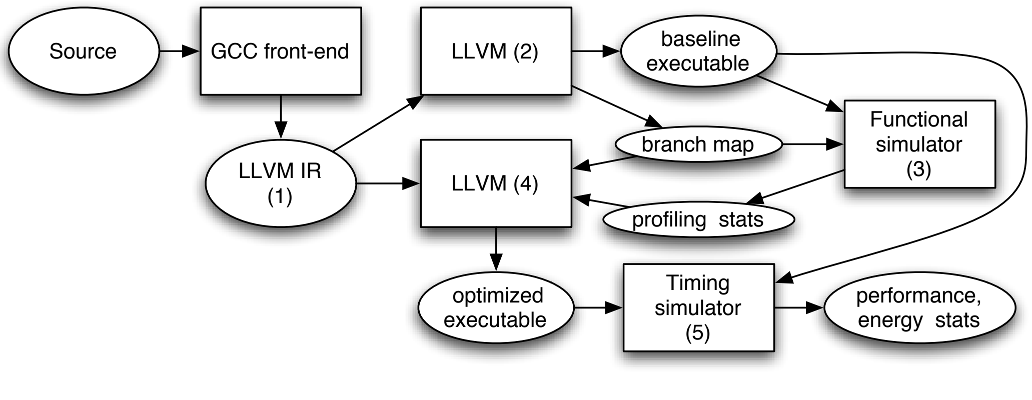

Figure 4.1 Flow chart of our feedback directed optimization (FDO) setup

Figure 4.1 illustrates the composition of our feedback-directed optimization system. This work was done in the context of the LLVM compiler and a SimpleScalar-based simulator. Our experiments involved two phases: 1) a profiling phase to collect data for applying the selection heuristics from Chapter 3, and 2) an experimental phase where we transformed code regions in response to the heuristics and evaluated the performance and energy gains. Because we are using profile-based if-conversion, it was important to have a mechanism for relating profiled branches in the generated code to the branch instructions in the compiler's internal representation (IR). Below, we discuss the components of this infrastructure and then walk through our experimental method.

We chose the *Low-Level Virtual Machine* (LLVM) compilation framework to implement our analyses and code transformations [12]. LLVM is a virtual instruction set and compiler back-end with many sophisticated features that were used in this study:

- Convenient static single-assignment code representation, with a rich set of pre-existing optimization passes
- GCC-based front-end support for compiling C and C++ code
- Back-end support for the Alpha ISA, enabling the use of SimpleScalar-based simulation tools
- Support for predicated execution with a built-in select statement in the internal representation (IR).
- Existance of a *PC marker* identifier to associate a branch in the generated code with a branch in the IR structure
- Helpful documentation and assistance for adding new features

In Figure 4.1, steps 1 and 2, we compiled the Spec2000 integer benchmarks to LLVM bytecode and created analysis passes to identify the previously described branch structures. We annotated every basic block in the program with a unique *PC marker* intrinsic instruction. This marker passes through to the generated code as a symbol in the object file and serves to associate the LLVM basic block structure with the instruction address of its terminating branch. For each benchmark considered, we created a branch map containing the *PC marker* identifiers from the IR and their associated branch instruction addresses in the generated code. This branch map was used to track the higher-level branch structures for profiling and transformation.

To collect branch statistics, we modified a functional simulator from from the SimpleScalar simulation infrastructure for Alpha ISA [13]. The functional simulator is given the *PC marker* branch map as an input file. In step 3, the functional simulator observes these branches and tracks their prediction behaviors during the benchmark simulation. For a hammock structure, the simulator records branch mispredictions and bias values in addition to the number of conditional move and wrong-path instructions that would be executed if the hammock were if-converted. In the case of loops, the functional simulator records trip counts and misprediction data. Conjunctive branches are observed to determine if combining the branches would improve the net predictability. The profile-based if-conversion heuristics use this profile data to determine whether a branch should be if-converted. In step 4, we generated an optimized executable in response to the heuristic outcome.

Because our goal in this study was to explore the impact of branch transformations on performance and energy, we chose to focus our transformations on a single branch for each experiment. In contrast to measuring aggregate results for many transformations in a benchmark, we hoped this would give greater insight about the veracity of our if-conversion hypotheses. For each branch we were studying, we generated and tested other code versions in addition to the heuristic choice to learn if the heuristic was choosing the best branch transformation to apply. The branches we select are from the following six benchmarks, which represent the subset of Spec2000 integer benchmarks that runs in our simulation infrastructure: `164.gzip`, `181.mcf`, `186.crafty`, `197.parser`, `256.bzip2`, and `300.twolf`.

In step 5, a SimpleScalar-based out-of-order microprocessor timing simulator is used to measure the performance effects of the branch transformations. The timing simulator is highly configurable, and we selected parameters that are representative of a modern-day superscalar processor. Our primary configuration features a 4-wide instruction fetching width and a 128-entry *register update unit* (RUU) that is comparable to the instruction window sizes of current superscalar processors. We also considered a 4096-entry RUU to represent the instruction window size of a future CFP-like machine. In our simulations, however, we did not observe a high enough incidence of L2 data cache misses to cause a significant performance difference between the two configurations. L2 cache misses to main memory are the primary bottleneck that large-window machines like CFP attempt to mitigate, and a lack of L2 cache misses lowers the MLP-effectiveness of a large-window machine running these benchmarks. In light of this observation, we decided to omit the 4096-entry RUU configuration from the evaluation. To measure the impact of the transformations on CPU energy usage, we used the Wattch [14] component for SimpleScalar. Energy numbers for Wattch are reported from the *cc_3* non-ideal clock gating measurement.

The branch predictor is a combined two-level and bimodal predictor. The two-level predictor has a 4096-entry table size and 8-bit history register. The bimodal table size is 2048 entries and the meta-table size is 1024 entries. The branch target buffer and return address stack are simulated as perfect. The minimum branch misprediction penalty is 12 cycles. We configured a 32KB, 2-way associative L1 data cache and a 64KB, 4-way associative L1 instruction cache. The L2 cache is 1MB in size, 4-way set associative, and guards a 200-cycle latency to main memory. Our configuration uses a perfect memory dependence predictor.

4.2 Path Length Determination

Different versions of code will have varying path lengths, and it is important for comparison to ensure that each version is performing an equivalent amount of work. One way to ensure this is to run the benchmarks from start to completion, but this approach was prohibitive under our simulation environment. Instead, we maintained a data structure in the functional simulator that tracked the access count of each branch it was observing. We modified the simulator to output the number of times each branch had been visited since the simulation began in periods of 100 million instructions. Because our peeling transformations will alter the number of iterations in a loop, we tracked the number of times control arrives at each loop rather than how many times its backedge branch was visited. An example of this counting method is shown in Figure 4.2.

We used the access count information to designate start and end points in the baseline and if-converted code. From our profile data and access counts, we selected three trace intervals to simulate. We selected these three intervals to represent a range of misspeculation levels for the program phases in which the branch is active. The start point in each interval is identified by the

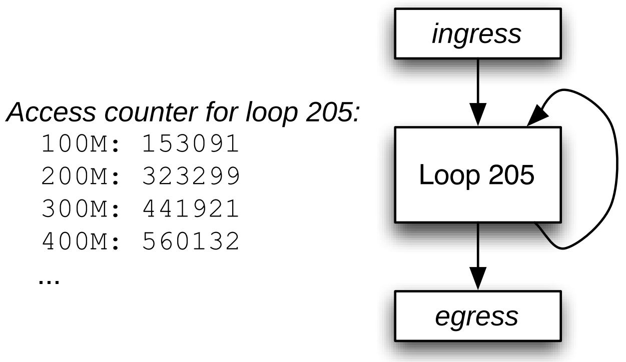

Figure 4.2 Example of counting method for ensuring equal path lengths.
The access count for loops is measured at the ingress point. This value will not change when the loop is peeled.

branch structure's access count at the start of the interval; the end point is marked by the branch structure's access count at the end of the interval. The path length of the predicated code will vary depending on the number of wrong-path instructions that result from predication. Because the baseline and optimized code versions start and stop at corresponding branch instructions, we can measure performance and energy gains by directly comparing their cycle times and energy measurements.

4.3 Hammock Evaluation

In this section, we evaluate the effectiveness of our hammock selection heuristic. It is worth noting that the LLVM *tail duplication* pass caused many hammocks to be transformed into non-converging branch structures. Tail duplication attempted to replace the unconditional branches to the join point with copies of the join code itself. To facilitate our study, we modified the tail duplication pass to exclude hammock structures from consideration. This had a negligible impact on performance and opened up additional predication opportunities. We did not alter the primary function of tail duplication in LLVM, which is to rotate loop branches into do-while form.

Each experiment in our evaluation is represented by four data points. The first three columns compare the number of cycles, path length, and energy consumption to those measurements in the baseline configuration. The fourth column compares the ratio of instructions fetched to instructions committed between the two configurations. This ratio correlates to misprediction rate.

4.3.1 Simple hammocks

In Figure 2.1, we identified that simple hammocks contribute to a high occurrence of mispredictions in benchmarks `181.mcf` (18%) and `300.twolf` (39%). We if-converted seven hammocks that account for the majority of these mispredictions and found that predication offers a performance gain in almost every case, as shown in Figures 4.3 and 4.4. According to the if-conversion heuristic presented in Section 3.2, six of the seven branches should be if-converted for a gain in performance. The heuristic correctly indicates hammock 262 in `181.mcf` should not be predicated due to its low misprediction rate and high wrong-path instruction fetching overhead, which average 7.8% mispredictions and 12 overhead instructions fetched, respectively. We observed hammock 262 in `181.mcf` to show inconsistent gains with a large slowdown in the third trace interval. This interval represents a median level of prediction accuracy (98%) for the hammock branch, while intervals one and two represent periods of uncharacteristically poor predictability (<90%).

In Figures 4.3 and 4.4, we list dynamic instruction coverage for each hammock to provide context for its execution frequency. It is interesting to observe hammocks showing consistent energy savings after if-conversion are the ones that have the highest speculation costs. Hammocks 262 and 276 have the lowest speculations costs, with misprediction rates of 7.8% and 15.5%. These misprediction rates are not high enough for predication to overcome the added energy cost of executing both paths through these hammocks. Even though the heuristic indicates that if-converting hammock 276 is profitable, it correctly estimates a smaller gain for if-conversion of this hammock than it does for any of the other profitable hammocks in this experiment.

4.3.2 Nested hammocks

For nested hammocks, we studied two structures responsible for a large number of mispredictions in benchmarks `164.gzip` and `186.crafty`.

The nested hammock shown in Figure 4.5 is found in one of the most active code regions in the `164.gzip` benchmark. The outermost branch 45 is heavily biased to skip over the inner hammocks.

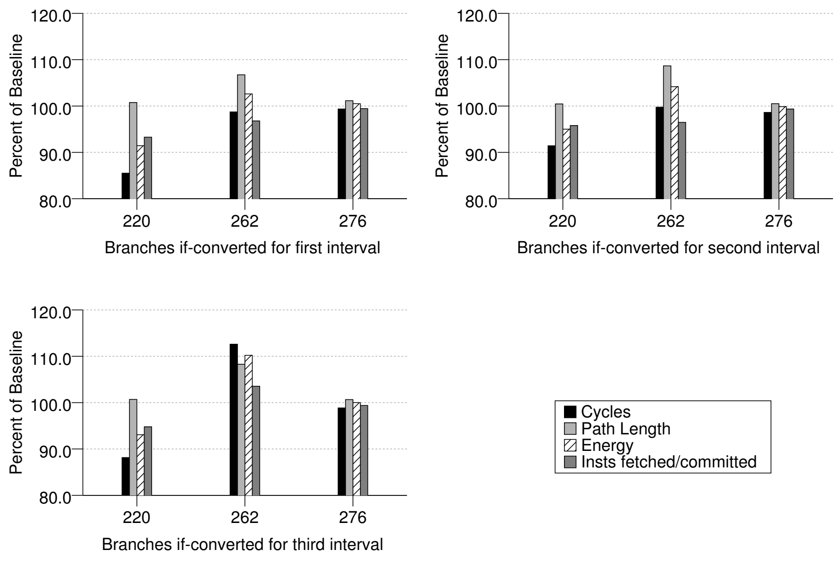

Figure 4.3 Results for if-converting simple hammocks in `181.mcf`
The trace intervals for hammock 220 do not correspond to those of hammocks 262 and 276. Hammock coverage is measured as a percent of the overall dynamic instructions in the interval. In the intervals we measured, the hammock coverage for hammock 220 averaged 14.2%. Hammock 262 averaged 13.5% coverage and hammock 276 averaged 2.73% coverage.

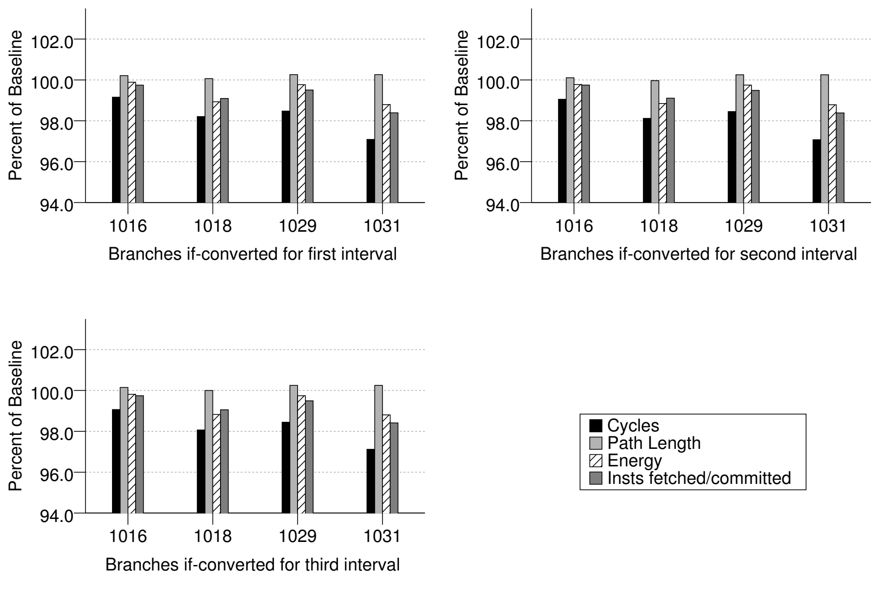

Figure 4.4 Results for if-converting simple hammocks in `300.twolf`
Hammock coverage is measured as a percent of the overall dynamic instructions in the interval. In the intervals we measured, the hammock coverage for hammock 1016 averaged 1.13%. Hammock 1018 averaged 1.62% coverage. Hammocks 1029 and 1031 averaged 2.10% coverage and 2.80% coverage, respectively.

The extra work required *to fetch* the inner region's instructions dominates the savings in wrong-path cycles. However, the middle branch 46 has an even bias, and is a good candidate for if-conversion. Figure 4.6 shows modest performance gains when the middle branch is if-converted by itself or with the inner branch 47. Our selection correctly selects branches 46 and 47 for if-conversion and passes on branch 45.

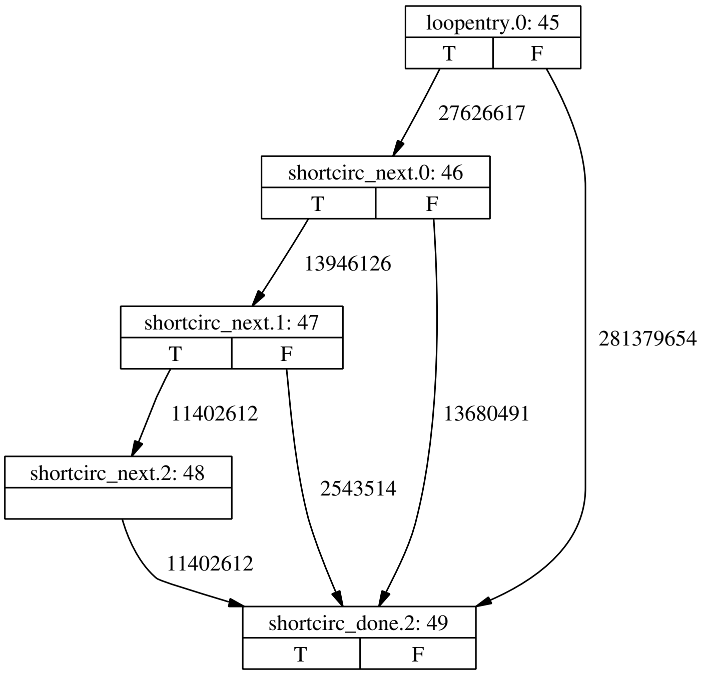

Figure 4.5 Control flow graph for nested hammock from function *longest_match* in `164.gzip`

Function *FirstOne* in `186.crafty` returns the position of the first non-zero bit in a 64-bit value by splitting the value into four 16-bit words and using the first non-zero word to index into a lookup table containing the correct value to return. The control structure for this function is a nested hammock with four branches. The first three branches in Figure 4.7 are responsible for nearly 20% of the overall branch mispredictions in `186.crafty`. The final branch is heavily biased and does not represent a misprediction burden. Mainly because of the high instruction fetching overheads calculated by the heuristic, none of these hammock branches were selected for if-conversion. However, this did not account for the greater freedom allowed for the static code scheduler to place and combine instructions in if-converted code. Our simulations confirmed that

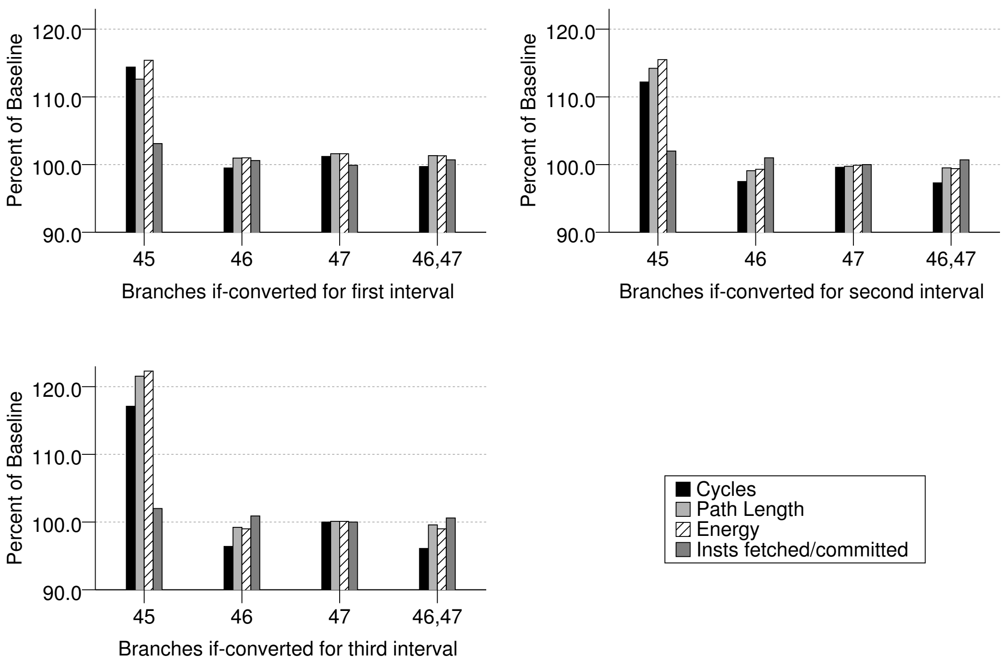

Figure 4.6 Results for if-converting nested hammocks in function *longest_match*
In trace interval 1, the instructions in the hammock were responsible for 13% of the dynamic instructions. In intervals 2 and 3, the instructions accounted for 18% and 26%, respectively.

if-conversion is beneficial for this nested hammock, contrary to the heuristic result.

We evaluated all combinations for if-converting the set of branches and found the best performing combination to be the case of if-converting all branches including the final branch D. If-converting each of the three difficult branches A, B, and C is also a good option, but presents an interesting trade-off: the predictability of branch D becomes worse when it is unguarded by branches A, B, and C. As shown in Figure 4.8, branch D is visited much more often in this case. Because branch D's condition is independent of the conditions for A, B, and C, the additional executions result in a weaker bias for branch D than when it was guarded by branches A, B, and C. It is important to keep in mind that predicating outer branches in a nested hammock may influence the predictability of inner branches. This property is known as *misprediction migration* [15].

To maintain D's favorable bias, we merged branch D's condition with the logical AND of the branch conditions for A, B, and C. As shown in Figure 4.9, merging the branch condition improved the predictability and performance over the unmerged case, but also raised the instruction count with the added AND instructions. The final half hammock contained only one instruction besides

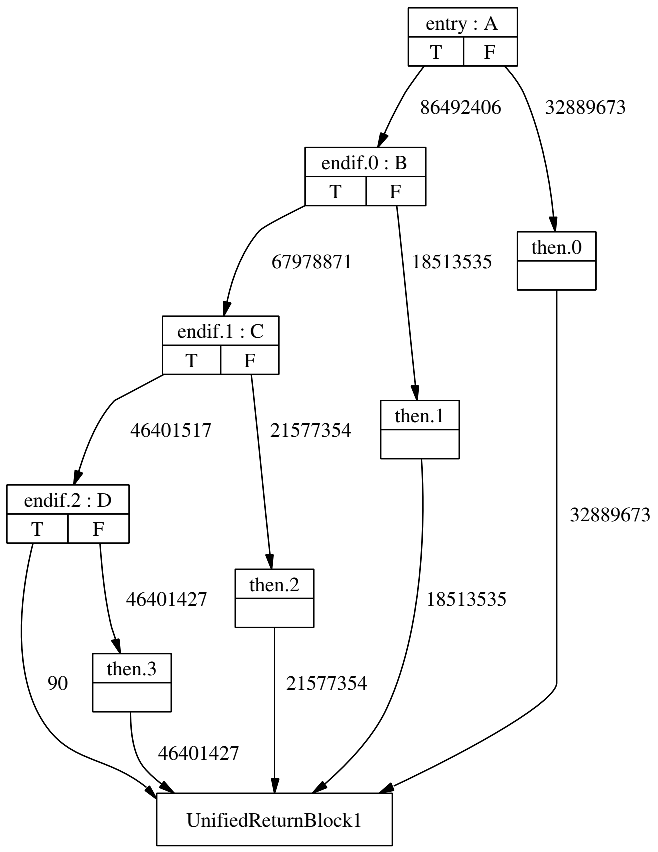

Figure 4.7 Control flow graph for function *FirstOne* in `186.crafty`

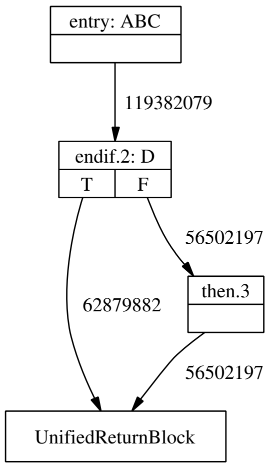

Figure 4.8 Example of changing inner branch bias due to if-conversion in function *FirstOne*

Notice the change in bias of branch D after if-converting branches A, B, and C which results in *misprediction migration.*

the branch. Our results indicate that if-converting the final branch offered greater static code scheduling freedom without the overhead to fetch and execute a large number of extra instructions.

Another interesting finding shown by Figure 4.9 is that better performance does not always correlate with energy savings. The widening discrepancy between cycles and energy in the last four columns shows that the energy cost of the increased path length outweighs the energy savings resulting from fewer branch mispredictions.

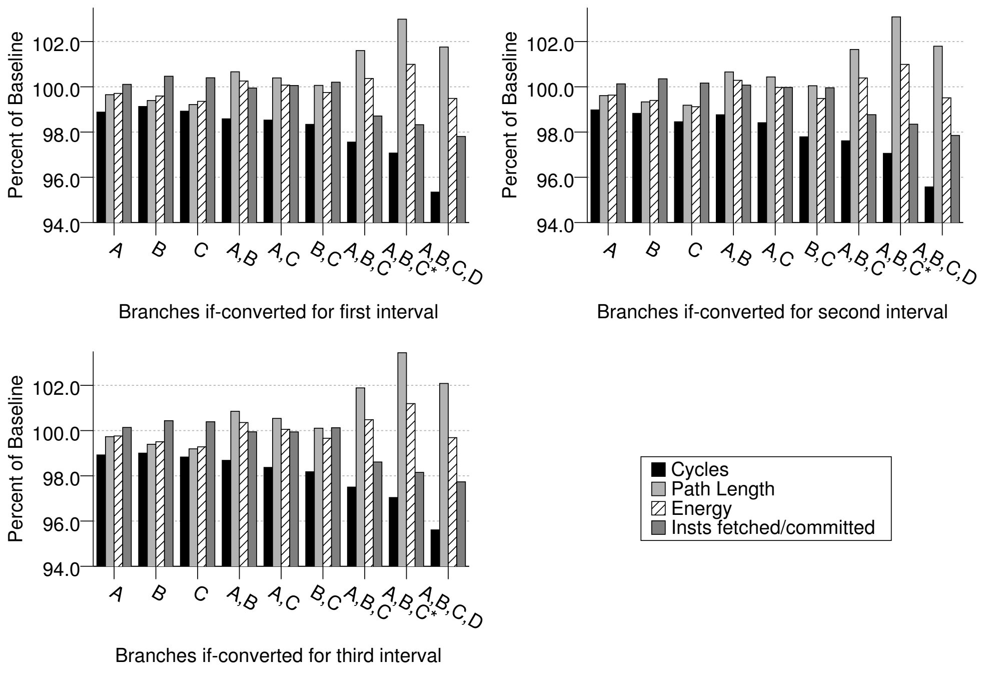

Figure 4.9 Results for if-converting nested hammocks in function *FirstOne*
The column labeled *A,B,C** represents the case where branch D's condition was combined with the conditions of branches A, B, and C for a gain in performance over the uncombined case *A,B,C*. The instructions in this nested hammock are responsible for around 6.0% of the dynamic instructions in the trace intervals we simulated.

4.4 Loop Peeling Evaluation

As shown in Figure 2.1, we found that many branch mispredictions occur among simple loop backedges in benchmarks `181.mcf` (22%) and `300.twolf` (46%). We selected five backedge branches that are responsible for a majority of these mispredictions and tested a range of loop peeling factors.

Loop 265 in Figure 4.10 is a backedge that loops around full hammock 262 from Section 4.3.

The backedge branch is responsible for nearly 15% of `181.mcf`'s mispredictions. In order to isolate the effects of loop peeling, we do not predicate the internal hammock in this experiment.

For the first interval, the best performance gain is obtained for one peel. Our heuristic incorrectly estimated no benefit for loop peeling in this interval, but it did not account for the 2% shorter path length resulting from the removal of dead instructions in the peel. The fourth bar indicates loop peeling does not provide an overall misprediction savings in this case. The heuristic correctly estimates a optimal peeling factor of two for the second and third intervals. Through simulations, we verified this selection provides a performance and energy return in both cases. The trip count distribution for the third interval shows peeling two iterations removes the need to enter the loop body for most executions (over 85%), enabling a 5% performance gain and 3% energy savings.

Loop 275 is made up of a single basic block that is part of a partitioning operation in `181.mcf`'s *quicksort* implementation. It is responsible for 6% of the overall branch mispredictions in this benchmark. The guarding branch is frequently mispredicted and by if-converting this branch, we are able to obtain the speedup shown in column 0 of Figure 4.11. Our loop peeling heuristic correctly indicated the guarding branch should be predicated, but no additional peeling should be performed. However, our tests show peeling one or two iterations in addition to if-converting the guarding branch offers a slight additional savings in cycles and energy. Both loop peeling examples from `181.mcf` show that energy usage is closely correlated to the program's path length.

Loops 1036, 1038, and 1040 in `300.twolf` are consecutive single-block loops found in function *new_dbox_a* in `300.twolf`. They combine to cause over 13% of the mispredictions in this benchmark. As shown in the right side of Figures 4.12, 4.13, and 4.14, the three loops in this function have the characteristic that the backedge branch is mispredicted more than once per loop exit, on average.

Loop 1036 is an interesting example that shows how loop peeling benefits from constant propagation. Figure 4.12 shows there is a savings in path length even for peeling factors that exceed the loop's average trip count. The induction variable in this loop is a counter that is initialized to zero. With constant propagation, the peeled iterations are able to perform a series of constant-indexed loads rather than adding to a base address. This eliminates three add instructions per iteration for a loop body that began with only seven instructions. In general, constant propagation and dead code elimination help specialize the peeled iterations for greater efficiency. We found our model was often too conservative in estimating the best peeling factor. This is especially apparent in this example. The heuristic chose to peel eight iterations, but there are greater performance gains at fifteen peels.

Loop 1038 is very similar to loop 1036, except it traverses an array in reverse order, preventing the simpification of as many instructions through constant propagation. Our heuristic suggested a peeling factor of eight for this loop, which is near-optimal for performance, as we show in Figure 4.13. Energy consumption is more directly affected by the increasing path length than the number of mispredictions saved, and we measure the best energy savings when the loop is peeled six times. It is not clear why a peeling factor of three affects performance and energy differently than the other peeling factors, but it appears to be a result of the particular combined 2-level and bimodal predictor configuration used in the simulator. We were able to eliminate this discontinuity in the branch misprediction rate by changing our predictor configuration to use a similar bimodal/local branch predictor.

In Figure 4.14, loop 1040 has a more even trip count distribution than the other loops, which helps limit its path length explosion until peels seven and eight. In this example, energy consumption leads path length growth as the peeling factor is increased. The heuristic selected a peeling factor of two for this loop, but the best performance improvement was measured at five peels. Like the other loops in `300.twolf`, the peel size in loop 1040 is significantly reduced by constant propagation and dead code elimination.

As indicated in the examples, our loop peeling heuristic lacks the ability to quantify the positive effects of constant propagation and dead code elimination. Even without accounting for this factor, our heuristic identified peeling factors that resulted in a positive performance gain in almost every case. We also observed a trend that energy consumption correlates more closely with path length than with the number of mispredictions saved. Loop peeling is an effective optimization for lowering energy consumption insofar as it enables the path length to be reduced through constant propagation and dead code elimination.

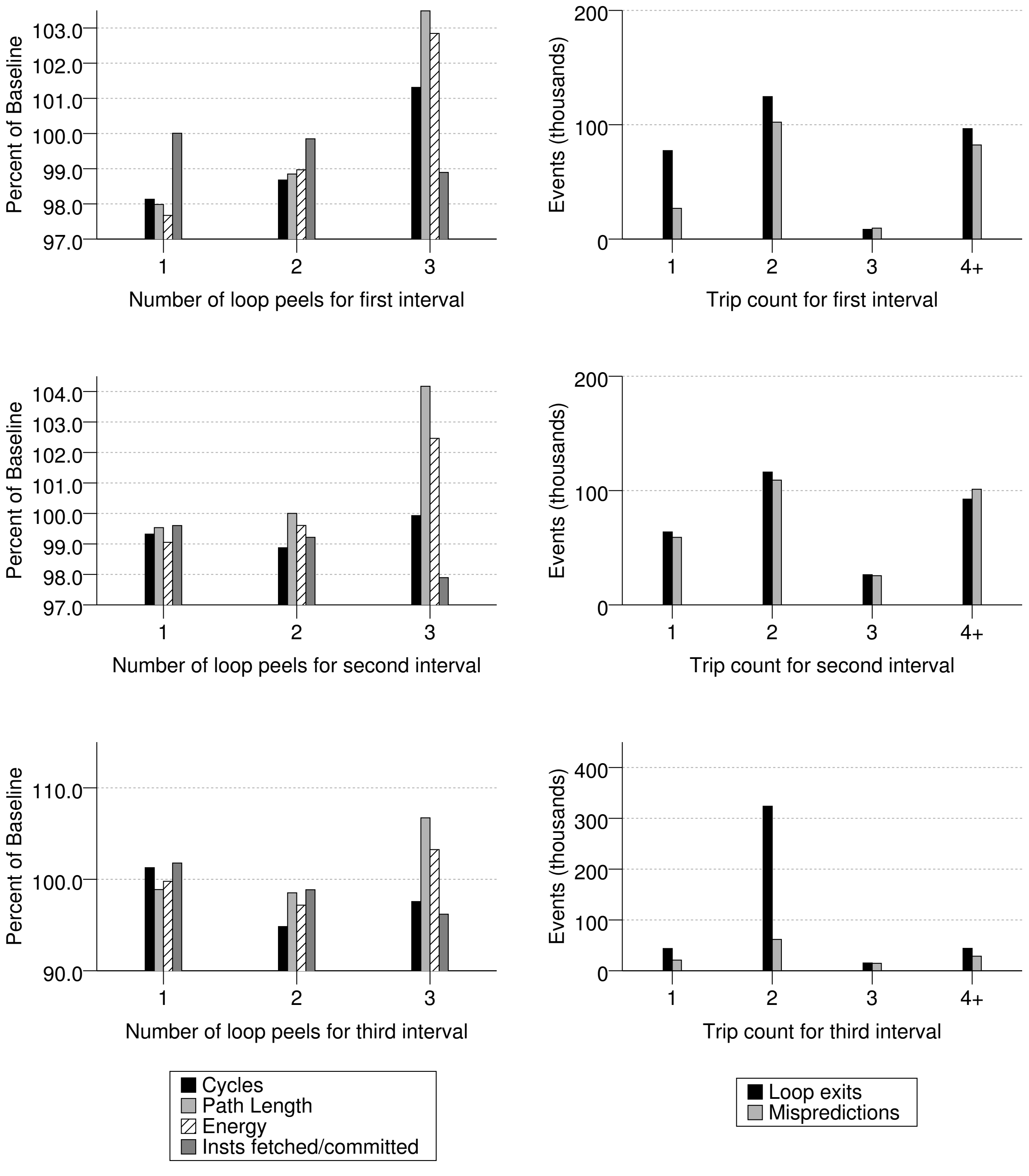

Figure 4.10 Loop peeling factors for loop 265 in `181.mcf`

The instructions in this loop are responsible for around 25% of the dynamic instructions in the trace intervals we simulated. Columns 1, 2, and 3 represent loop peeling factors for loop 265 with the inner hammock 262 speculated.

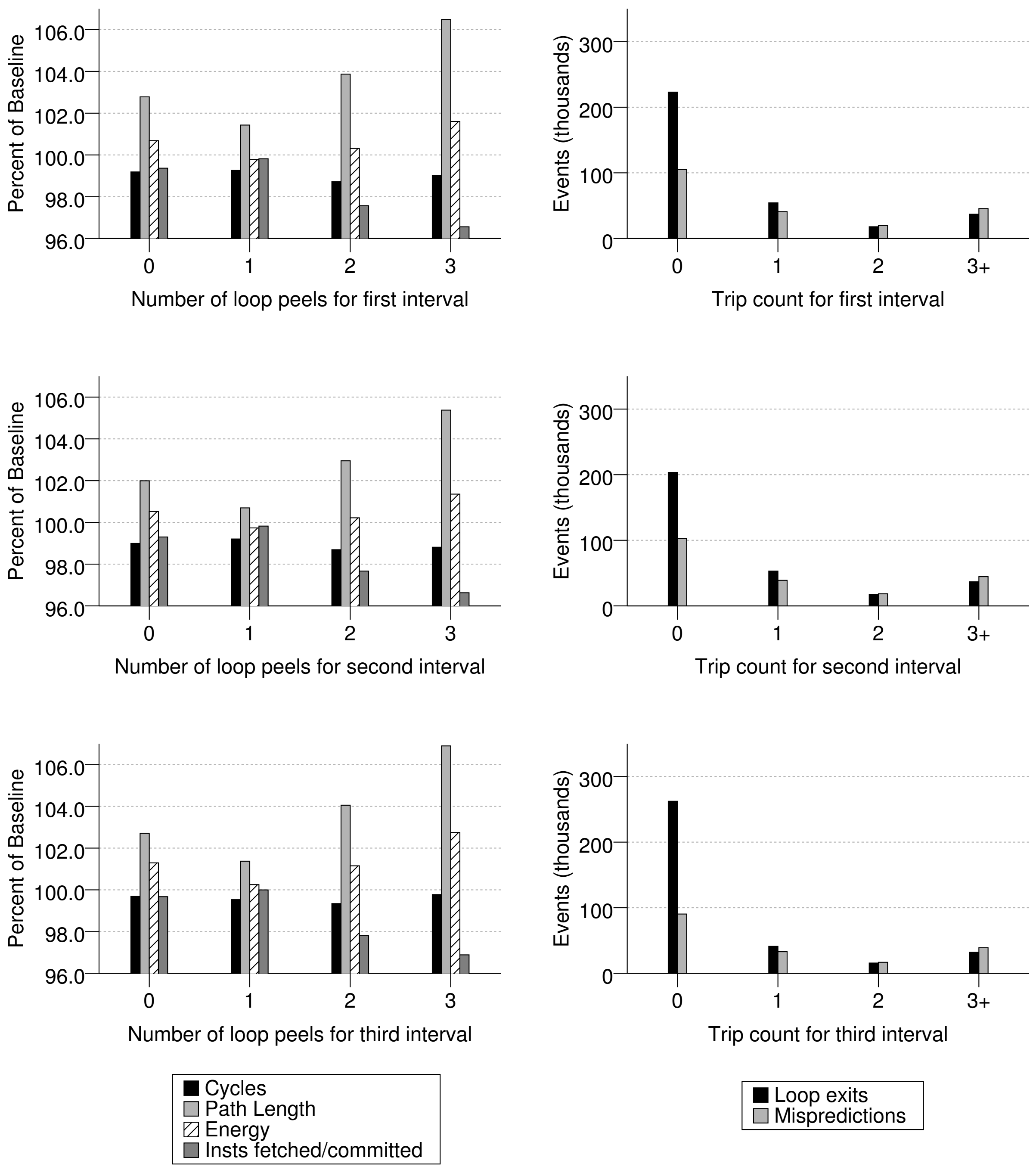

Figure 4.11 Loop peeling factors for loop 275 in `181.mcf`
The instructions in this loop are responsible for around 4.3% of the dynamic instructions in the trace intervals we simulated.

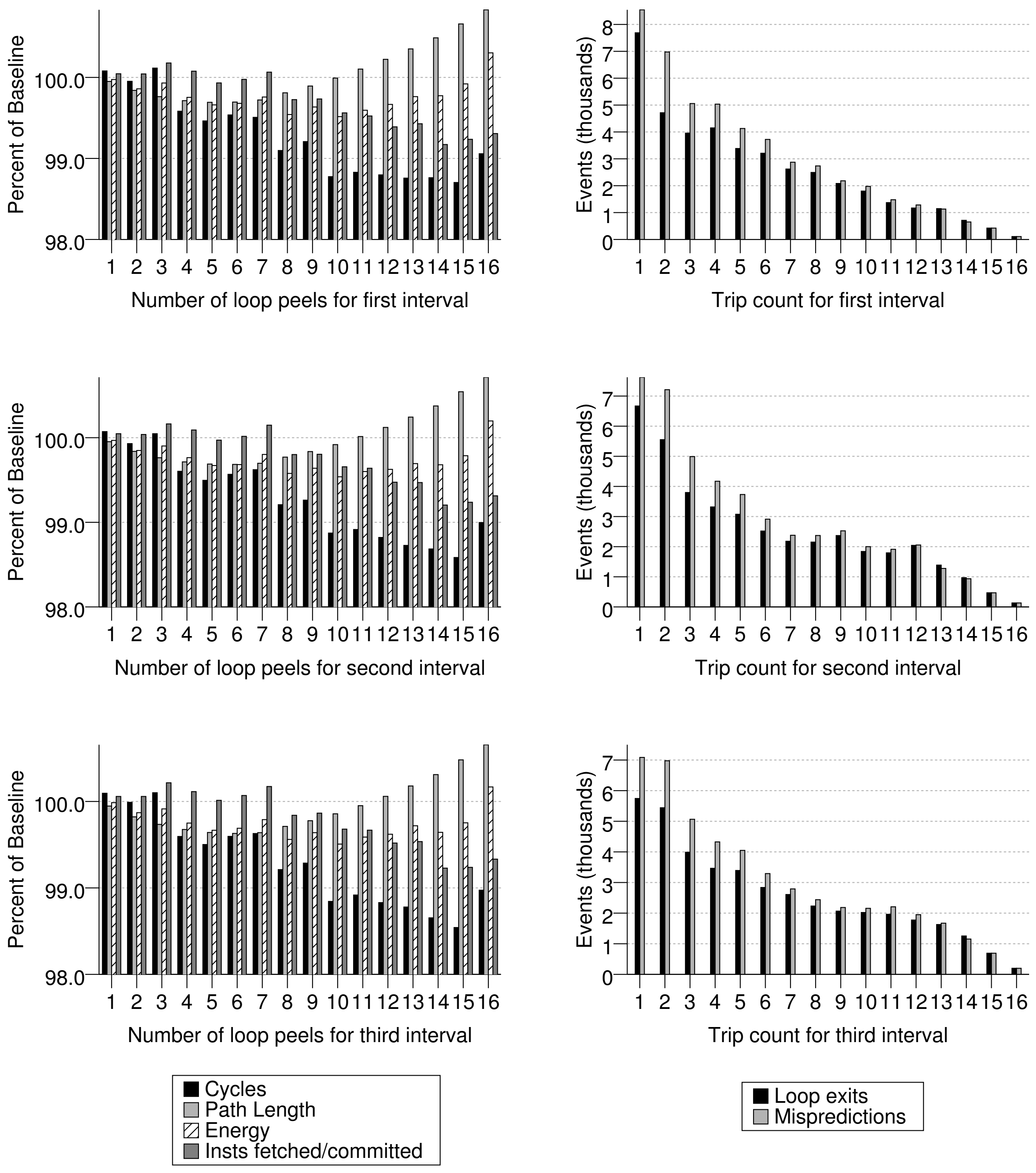

Figure 4.12 Loop peeling factors for loop 1036 in `300.twolf`
The instructions in this loop are responsible for around 1.6% of the dynamic instructions in the trace intervals we simulated. The trip count for this loop never exceeds sixteen.

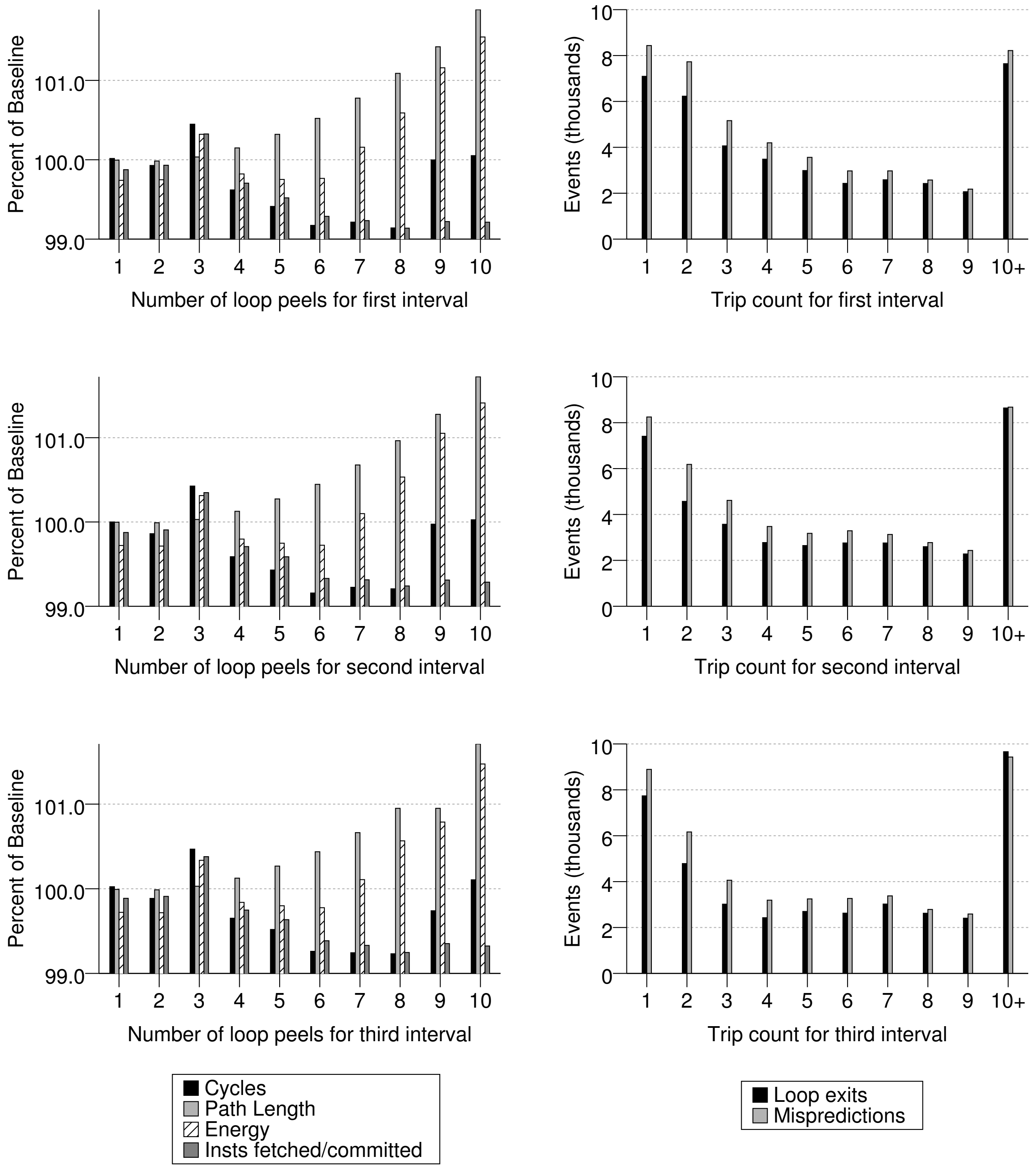

Figure 4.13 Loop peeling factors for loop 1038 in `300.twolf`
The instructions in this loop are responsible for around 2.3% of the dynamic instructions in the trace intervals we simulated. The trip count for this loop never exceeds sixteen.

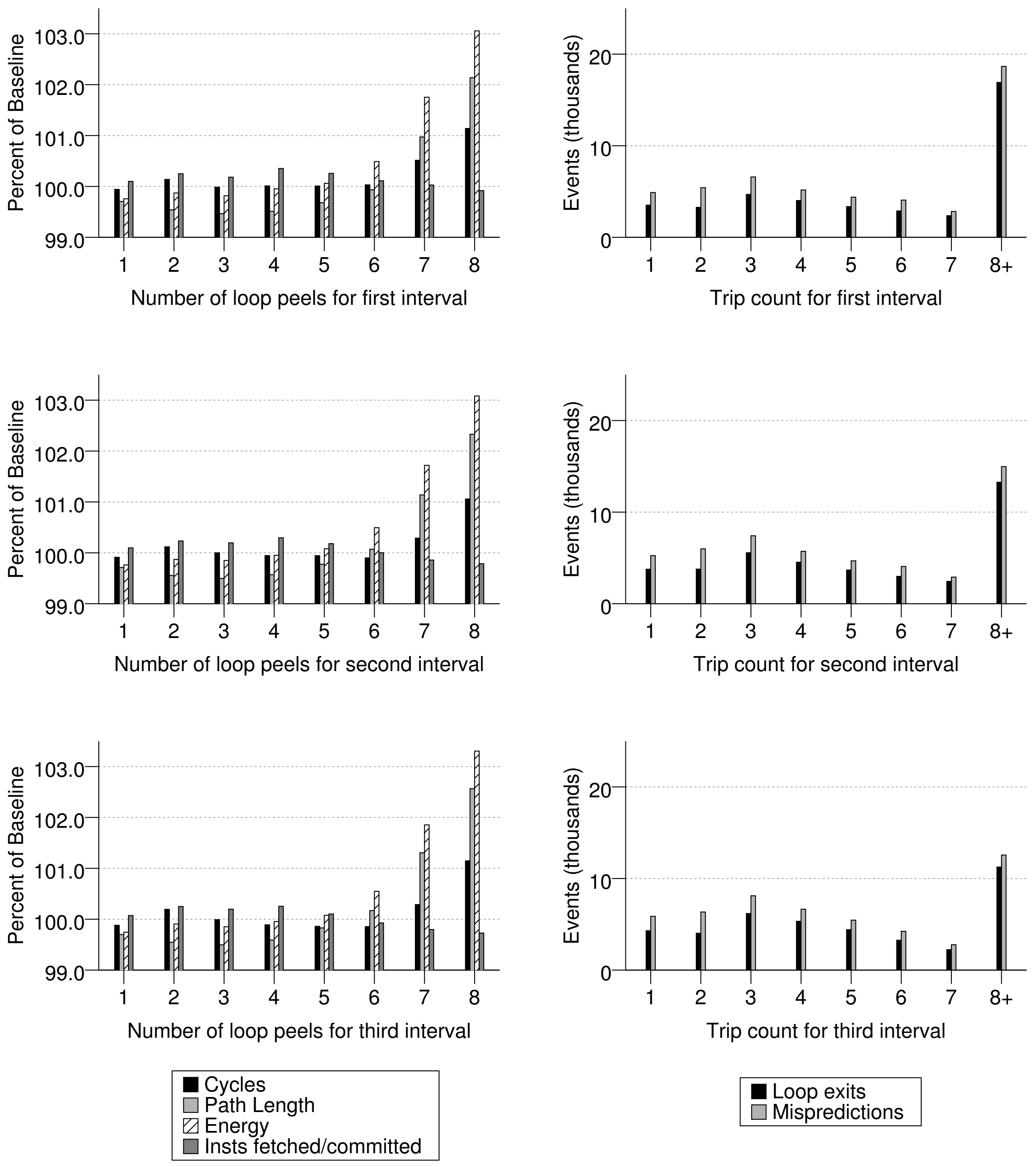

Figure 4.14 Loop peeling factors for loop 1040 in `300.twolf`
The instructions in this loop are responsible for around 4.2% of the dynamic instructions in the trace intervals we simulated. The trip count for this loop never exceeds sixteen.

4.5 Conjunctive Branch Evaluation

Figure 4.15 contains a source listing of `181.mcf`'s function *bea_is_dual_infeasible*. Figure 4.16 shows the control flow graph of this function contains two conjunctive branch pairs. Each branch in these pairs is mispredicted with a rate that approximately equals the branch bias. This function provides a good example of how combining conjunctive branches can improve overall bias and predictability in hard-to-predict branches.

```
int bea_is_dual_infeasible( arc_t *arc, cost_t red_cost )
{
    return(    (red_cost < 0 && arc->ident == AT_LOWER)
            || (red_cost > 0 && arc->ident == AT_UPPER) );
}
```

Figure 4.15 Source listing of function *bea_is_dual_infeasible* in `181.mcf`

Because branches W, X, Y, and Z in Figure 4.16 are mispredicted with roughly the same frequency as the branch biases, we can apply the conjunctive branch bias heuristic from Section 3.11. Merging branches W and X results in a slight improvement in bias, while merging branches Y and Z results in a significant improvement due to the heavy fall-out bias of branch Z. Combining all four branches offers no improvement in bias over the merged pairs, because the YZ and WX fall-out edge weights do not exceed half of the influx value.

According to our selection heuristic, the savings in misspeculated cycles due to branch combining exceeds the instruction fetching and execution overhead for combined branch YZ, but not for the other combinations. Figure 4.17 confirms that combining branches Y and Z offers a 1% to 2% reduction in cycle and energy cost, while the other combinations do not improve in these areas.

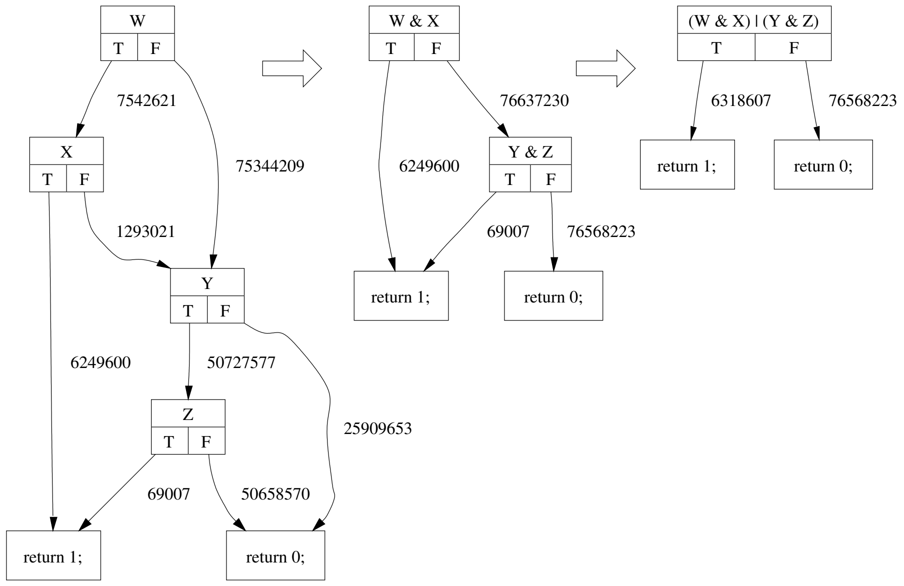

Figure 4.16 Improving branch bias by combining branches
Combining branches Y and Z provides the most significant improvement in net bias.

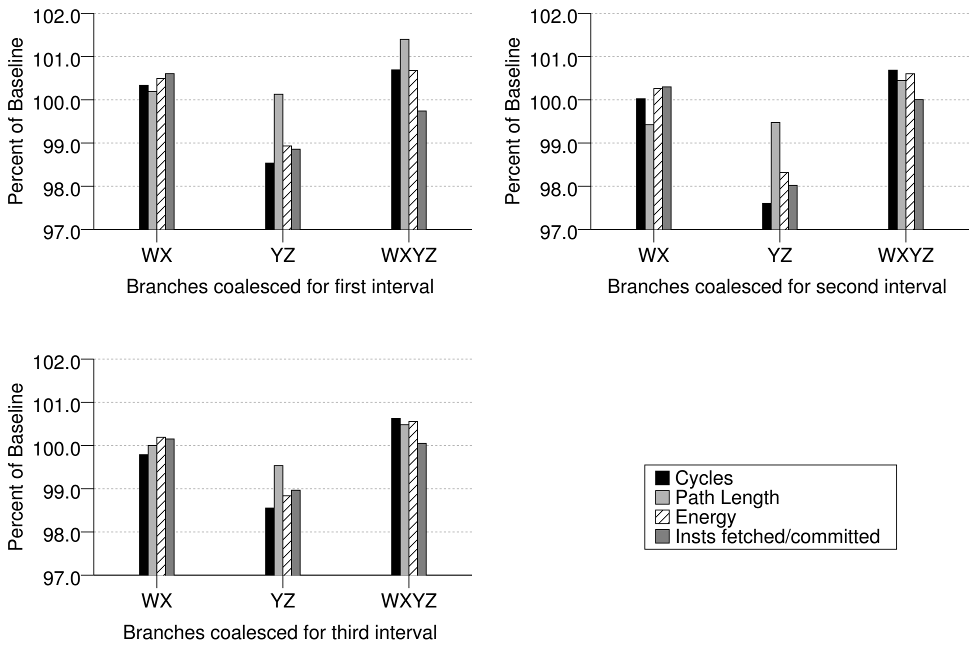

Figure 4.17 Conjunctive branch evaluation
The instructions in this loop are responsible for around 7.6% of the dynamic instructions in the trace intervals we simulated.

CHAPTER 5

Conclusion

In this thesis, we have shown predicated execution can be used to reduce the rising relative cost of branch misspeculations in out-of-order processors. We developed a classification of control flow structures and quantified each structure's contribution to the overall number of branch mispredictions in the Spec2000 integer benchmarks. We investigated how predicated execution can be utilized to reduce the misprediction cost presented by these branches.

We developed heuristics for profile-directed if-conversion in a dynamically scheduled processor with conditional move instructions. These heuristics model two inefficiencies of predicated execution: the wasted cycles spent fetching wrong-path instructions, and the added latency to produce a predicated liveout value. We implemented an offline feedback-directed optimization system using LLVM to analyze and optimize the benchmark code, and we executed the code on an out-of-order microprocessor simulator.

We evaluated the results to characterize the performance and energy advantages of predicated execution. We verified our hammock selection heuristics were effective in identifying profitable hammocks to if-convert. When extending our heuristic to consider nested hammocks, we showed it is important to consider the effects of static code scheduling and misprediction migration in determining which branches to if-convert. For loop peeling, we found constant propagation and dead code elimination often make the loop peels more efficient than the original loop body. We showed that when prediction accuracy correlates to branch bias in a conjunctive pair, it can be profitable to coalesce the branches for a net improvement in bias and predictability.

There are many questions left to be answered related to deploying predication for better performance and lower energy in superscalar microprocessors.

It would be interesting to show predication is an effective tool for a broad range of workloads, including those with a higher frequency of L2 cache misses. These are the programs that will be most impacted by the MLP capacity of future large-window machines. Indeed, it is important future work to extend this study to learn how mispredicting branches in other benchmarks compare to those in the Spec2000 integer benchmarks. Do other benchmarks have a similar distribution of mispredicting branches? Are there new, important branch structures that were overlooked in this study, and if so, can they be predicated? Do other benchmarks display as extreme a distribution property as the Spec2000 integer benchmarks, where often fewer than 10% of the static branch instructions account for over 99% of the overall mispredictions?

Also, how do our if-conversion heuristics fare with other benchmarks? Are loop peeling factors frequently underestimated due to constant propagation? To what extent does predication aid static code scheduling for an out-of-order processor? Is condition merging for nested hammocks an effective way to limit misprediction migration? Does predication erase the positive prefetching effect of misspeculations, and will this still be important as future large-window machines become more tolerant of memory latency [11]?

We have shown predicated execution can deliver modest performance and energy gains on modern dynamically scheduled microprocessors, and we believe it will grow in importance as branch misspeculation becomes a more significant barrier to performance in future machines.

REFERENCES

[1] T. Karkhanis and J. Smith, “A day in the life of a data cache miss,” in *WMPI*, 2002.

[2] J. Aragon, J. Gonzalez, and A. Gonzalez, “Power-aware control speculation through selective throttling,” in *HPCA*, pp. 103–112, 2003.

[3] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, “Continual flow pipelines,” in *ASPLOS*, pp. 107–119, 2004.

[4] Y. Chou, B. Fahs, and S. G. Abraham, “Microarchitecture optimizations for exploiting memory-level parallelism,” in *ISCA*, pp. 76–89, 2004.

[5] J. W. Sias, S. Ueng, G. A. Kent, I. M. Steiner, E. M. Nystrom, and W. W. Hwu, “Field-testing IMPACT EPIC research results in Itanium 2,” in *ISCA*, pp. 26–39, 2004.

[6] A. Klauser, T. M. Austin, D. Grunwald, and B. Calder, “Dynamic hammock predication for non-predicated instruction set architectures,” in *PACT*, pp. 278–285, 1998.

[7] P. Chang, E. Hao, Y. N. Patt, and P. P. Chang, “Using predicated execution to improve the performance of a dynamically scheduled machine with speculative execution,” in *PACT*, 1995.

[8] W. Chuang, B. Calder, and J. Ferrante, “Phi-predication for light-weight if-conversion,” in *CGO*, pp. 179–190, 2003.

[9] S. Mantripragada and A. Nicolau, “Using profiling to reduce branch misprediction costs on a dynamically scheduled processor,” in *ICS*, pp. 206–214, 2000.

[10] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W. W. Hwu, “A comparison of full and partial predicated execution support for ILP processors,” in *ISCA*, pp. 138–149, 1995.

[11] O. Mutlu, H. Kim, D. N. Armstrong, and Y. N. Patt, “Understanding the effects of wrong-path memory references on processor performance,” in *WMPI*, pp. 56–64, 2004.

[12] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *CGO*, 2004.

[13] D. C. Burger and T. M. Austin, “The SimpleScalar tool set, version 2.0,” Tech. Rep. CS-TR-1997-1342, 1997.

[14] D. Brooks, V. Tiwari, and M. Martonosi, “Wattch: a framework for architectural-level power analysis and optimizations,” in *ISCA*, pp. 83–94, 2000.

[15] B. Simon, B. Calder, and J. Ferrante, “Incorporating predicate information into branch predictors,” in *HPCA*, pp. 53–64, 2003.