On the Energy Effectiveness of If-conversion in Superscalar Microprocessors

Eric Zimmerman and Craig Zilles
Department of Computer Science
University of Illinois at Urbana-Champaign
Email: {ezimmerm, zilles}@cs.uiuc.edu

***Abstract*— Branch mispredictions represent a significant obstacle for both performance and power efficiency in modern out-of-order superscalar processors. If-conversion has been proposed as one approach to mitigate hard-to-predict branches that re-converge shortly after the branch. While previous work has extensively characterized the performance potential of predicating hammocks (generally on wide-issue (*e.g.*, 8-wide) machines), little has been reported about the energy implications of the optimization. In this paper, we consider the energy implications of if-conversion and explore whether it is still profitable in more energy efficient, narrower (*e.g.*, 4-wide) machine. Furthermore, we consider non-hammock if-conversion opportunities. The major contributions of this work are: 1) we report more than half of branch mispredictions belong to potentially if-convertible regions when loop peeling and conjunctive branches are considered, 2) we propose heuristics for guiding if-conversion of these non-hammock regions, 3) we make the observation (and exploit it in our heuristics) that a hard-to-predict branch's prediction accuracy is often close to its bias, and 4) we find that if-conversion often does lead to energy savings due to a net reduction in the number of instructions fetched, but that the savings is generally less than the performance improvements achieved.**

I. Introduction

The performance impact of if-conversion — the process of converting re-convergent control flow into data flow — has been extensively studied, especially in the context of in-order issue processors. In in-order machines, the primary motivation for if-conversion is to expand the compiler's optimization scope to enable tighter schedules [1], [2]. If-conversion can also be useful in the context of out-of-order (OOO) processors, by eliminating hard-to-predict branches [3]–[6], and we expect it to be increasingly important in the context of recently proposed large window out-of-order processors (*e.g.*, Continual Flow Pipelines [7] and Kilo-instruction Processors [8]) that are tolerant of resource conflicts and memory latency, but not branch mispredictions.

Motivated by these large window machines and cognizant of the power/energy constraints of modern and future processors, we sought to answer the following previously unanswered questions:

1) **What fraction of mispredicted branches can potentially be removed by if-conversion?** Previous if-conversion research in OOO machines has largely focused on predicating hammocks (*i.e.*, re-convergent control flow generally caused by `if-then` and `if-then-else` clauses). Klauser, et al. found that these simple hammocks represented only 11% of the branch mispredictions in SpecInt 95 programs. Clearly, we must generalize beyond simple hammocks to have a significant impact on the number of branch mispredictions.
2) **Can aggressive predication be profitable in the context of narrower, more energy efficient machines?** Much of the research on predication was done in the context of wide-issue (*e.g.*, 8-wide) machines, before the advent of power/energy being a significant constraint. These wider machines often have idle execution resources that eliminate much of the impact of the dynamic instruction count overhead of if-conversion. Does resource contention preclude achieving speedups in narrower (*e.g.*, 4-wide) machines?
3) **What is the energy impact of if-conversion?** While the energy implications of branch mispredictions are clearly not good — Aragon, et al. reported that their model spent an average of 47% of cycles fetching wrong path instructions and that those instructions were responsible for 28% of the total energy dissipation [9] — the impact of if-conversion is unclear. While if-conversion can reduce the number of branch mispredictions, it requires executing instructions along both control flow paths. To what degree does this elongation of path length mitigate the energy savings of reducing the number of wrong path instructions?

To answer these questions, we extended the LLVM compiler [10] and its Alpha back-end to support feedback-directed if-conversion and used a SimpleScalar-based simulator [11] and Wattch [12] to explore the performance and power/energy implications of if-conversion (our experimental methods are described in Section IV). In the experiments that we will describe, we if-converted one program region at a time, often in many different ways,

so as to explore trade-offs of performance and energy in if-conversion. We found this to provide more insight than aggregate statistics over the whole program. In particular, we make the following observations:

1) We found that, on average, over 50% of branches belong to regions that are if-convertible (in contrast to the above noted 11%). The primary distinction between these two numbers is that we consider if-converting nested hammocks, simple loop back edges, and *conjunctive* branches (*i.e.*, those resulting from non-trivial predicates to `if` statements). (Section II)
2) In developing heuristics for predicting when to if-convert loops and conjunctive branches we found it necessary to estimate misprediction rates for branches altered or introduced by the program transformation. We found that, for hard-to-predict branches, the branch's bias serves as a good proxy for its prediction rate. Also, we found that for accuracy it is necessary to actually generate the if-converted code so as to take into account any optimizations enabled by the branch removal. (Section III)
3) We verified that a very wide machine is not necessary to exploit if-conversion. We found that even a 4-wide machine can tolerate the extra operations that are generated by if-conversion (at least when targeting only the worst branches). (Section V)
4) Finally, we observed that energy savings are generally correlated to performance improvements, but typically the fraction of energy saved is less than the fraction of cycles saved. This is easily explained. Whereas both performance and energy benefit from a reduction in branch mispredictions, energy increases roughly linearly with increased path length, whereas performance is only slightly impacted because many of the additional instructions can be executed in parallel with the existing critical path. (Section V)

Due to space limitations, we must assume that the reader is already familiar with the basic concept of if-conversion, how it can be applied to simple hammocks, and mechanisms for implementing predicated execution. This material is already well covered by the existing literature [13].

II. A Taxonomy of Mispredicting Branches

The majority of branch mispredictions are caused by a small fraction of branches: the hard-to-predict branches [3]. To identify which of these mispredictions could be eliminated by if-conversion, we had LLVM categorize the control-flow structure associated with each branch and annotate each branch with this information. We then ran the benchmarks through a functional simulator to count the number of branch mispredictions for each branch. The resulting categorization of mispredictions for the six Spec2000 integer benchmarks that LLVM's Alpha backend currently supports is shown in Figure 1.

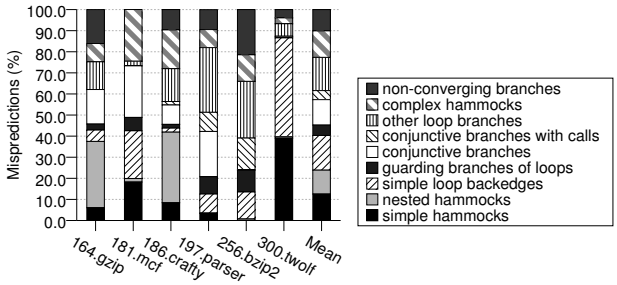

Fig. 1. Branch mispredictions by category.

Across the benchmarks, roughly 50% of the mispredictions are covered by five categories: simple hammocks, nested hammocks, simple loop back edges and their guarding branches, and conjunctive branches. These categories are if-convertible, as we show in this section. `Bzip2` is the notable outlier, with only 24% of mispredictions eliminatable via if-conversion. The remaining branches are generally not suitable for if-conversion due to the inclusion of loops, calls to non-trivial functions, or system calls on one or both of the paths to be if-converted.

A. Hammocks

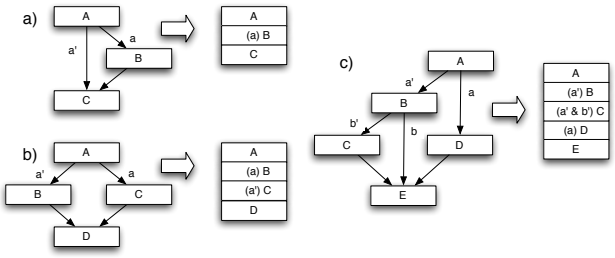

Fig. 2. If-conversion example for a half hammock (a), full hammock (b), and nested hammock (c).

Figure 2 shows examples of the three classes of hammocks that we consider for if-conversion in this study and demonstrates how these regions can be if-converted, which serves to introduce our notation for the more complicated examples. *Half hammocks* (a) implement if-then statements and *full hammocks* (b) implement if-then-else statements. *Nested hammocks* (c) are composed of simple hammocks or other nested hammocks. Importantly, it is not necessary to if-convert every branch in a nested hammock; inner branches may be highly biased and better left for speculation.

B. Loops

While back-edge branches of loops that iterate many times are generally easy to predict, loops with small, unpredictable trip counts can be responsible for a large fraction of mispredictions. The misprediction penalty of these loops can often be lowered by combining *loop peeling* with predication. Predicating the peeled iterations eliminates the need to speculate on these hard-to-predict back-edge branches but comes with an overhead cost of having to fetch and execute unused iterations of the loop body in some instances.

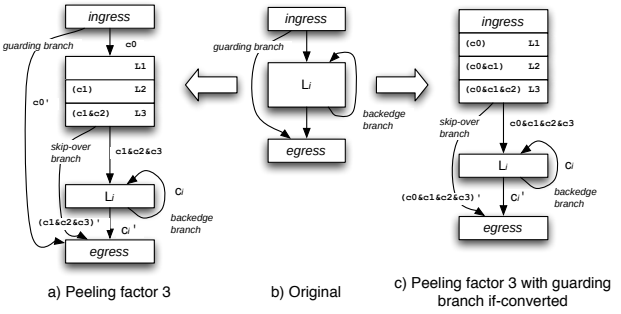

Fig. 3. Simple loop with guarding branch (b). Three iterations are peeled in addition to if-converting the guarding branch (c) or without if-converting the guarding branch (a).

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 3 illustrates a loop where three iterations are peeled and predicated with and without if-converting the guarding branch on the right and left, respectively.

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. Branches that implement breaks and continues can sometimes be merged with the back edge to improve their predictability as described in the next subsection. In general, it is impractical to peel and predicate iterations from an outer loop.

C. Conjunctive Branches

A *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 4a. Any liveouts from the second block should be predicated to ensure correct execution. This transformation needs to be carefully applied as merging the two branches can create a hard-to-predict branch in place of two easier to predict branches.

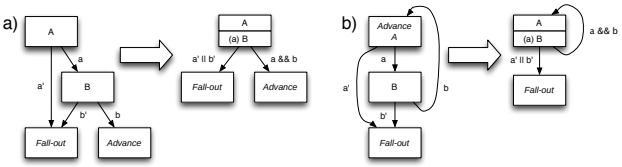

Fig. 4. Conjunctive branch pair and breaking loop variant.

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 back edge as shown in Figure 4b.

III. If-Conversion Heuristics

Not all if-convertible control-flow structures are beneficial to if-convert, so heuristics are required to guide the transformations. In particular, much of the benefit of if-conversion in OOO processors is due to the elimination of branch mispredictions, and hard-to-predict branches are difficult to identify without a profile. Hence, we consider profile-guided heuristics in this section.

Like the previous work in profile-based if-conversion heuristics [4], [6], we consider the trade-off between misspeculation cycles saved and increased schedule height due to if-conversion. To consider the costs of if-conversion, we first discuss the predication support in our simulated machine in Section III-A. Then, because previous work has focused on if-converting hammocks, we focus here on our heuristics for if-converting loop and conjunctive branches, in Sections III-B, and III-C, respectively. As previously noted, these heuristics need to consider changes in branch predictability and not just branch removal.

A. Predication Mechanisms in Out-of-order Processors

Our experiments are done in the context of the Alpha instruction set, which provides conditional move (or `cmove`) instructions (a form of partial predication [13]). A `cmove` has a single source operand and a predicate operand. If the predicate is true, the `cmove` will copy the

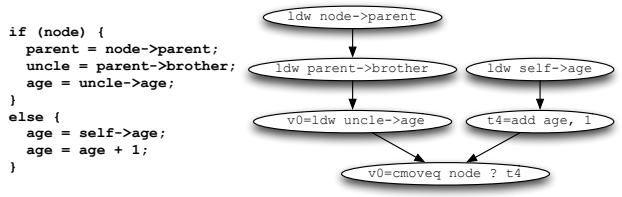

Fig. 5. Predication lengthens liveout dependency chain height. If the `then` clause is the correct path, predication lengthens the dependency height by one. If the `else` clause is the correct path, predication lengthens the dependency height by two.

source register value to the destination register. Otherwise, the `cmove` instruction behaves as a nop.

The two costs of if-conversion are best demonstrated by an example. Consider the code fragment in Figure 5; when it is if-converted, the dataflow in both clauses is executed and written to temporary variables, and a conditional move is inserted to select between the two. The first cost is the need to fetch extra instructions (*i.e.*, when the `then` path is taken, we still have to fetch the instructions for the `else` path). If we are in a region of execution that is fetch limited, then fetching additional instructions will extend the critical path length by the number of extra instructions divided by the fetch width. The second cost is an elongation of the execution critical path; traditional predication requires that both dataflow chains be evaluated before the `cmove` can execute. In all cases, at least one instruction (the `cmove`) is added to the dependence height, but more can be added when the two paths' dataflow heights are unequal (*e.g.*, when the `else` path in Figure 5 would be taken, the critical path is elongated by a minimum of two operations). To be conservative, we assume the impact of if-conversion is the worse of these two costs. The benefit of eliminating mispredictions must outweigh this cost for if-conversion to be profitable.

When store instructions are present in if-converted regions, they are predicated by using a `cmove` to guard 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. Also, we assume the availability of non-excepting versions of excepting instructions for use in if-converted regions.

B. 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 (*i.e.*, $ben(n)$) by:

$$ben(n) = misp\_lat \times (\sum_{i=0}^{n} misps_i - \min(\sum_{i=0}^{n} trip_i, \sum_{i=n+1}^{\infty} trip_i)) \quad (1)$$

$misp\_lat$ is the average penalty of mispredicting a branch. $misps_i$ is the number of mispredictions occurring on trip i of the loop, and $trip_i$ is the number of times the loop exited on the i^{th} iteration. This expression measures the number of back-edge mispredictions that are saved through peeling. From this savings, we must deduct the number of mispredictions added to the *skip-over branch* (see Figure 3) 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:

$$oh_{fetch}(n) = \frac{peel\_size}{IF\_width} \times \sum_{i=0}^{n-1} (n-i) \times trip_i \quad (2)$$

$$oh_{execute}(n) = (max\_ht + 1) \times \sum_{i=0}^{n-1} (n-i) \times trip_i \quad (3)$$

$$overhead(n) = \max(oh_{fetch}(n), oh_{execute}(n)) \quad (4)$$

$peel\_size$ 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, because constant propagation and dead code elimination can reduce the size of the peeled code. IF_width represents the number of instructions fetched per cycle. max_ht is the length of the longest dependency chain for a liveout definition in the loop body; the plus one is for the guarding conditional move instruction that is added to each peeled iteration. $trip_0$ 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 1, 2, and 3 should begin at one instead of zero. 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.

We can determine the optimal number of loop iterations to peel by finding the i that satisfies the expression $\max(ben(i) - overhead(i))$. In determining the optimal

number of iterations to peel, our heuristic does not currently consider code-expansion, although we find this to be low since generally only small loops are viable for peeling.

C. Conjunctive Branch Selection

Like other if-conversion opportunities, coalescing a conjunctive branch pair will only improve performance when the overhead of executing additional instructions can be overcome by an increase in branch predictability, either from an improvement in bias or correlation. This is expressed formally in equation 5, where $p(a)$ is the probability that control reaches block B and $p_{misp}(x)$ is the probability that branch x is mispredicted.

$$misp\_lat \times (p_{misp}(a) + p(a)p_{misp}(b)) > misp\_lat \times p_{misp}(ab) + overhead \quad (5)$$

$overhead$ can be computed in much the same way as was done for hammocks and loop unrolling. Although block A in Figure 4 is executed unconditionally, block B presents an instruction fetch overhead when branch A would have proceeded to the fall-out block. This overhead (detailed in Equation 6) is a function of the size of block B, the probability of falling-out without executing block B ($p_{fo}(a)$), and the number of conditional move instructions required to predicate block B's liveout values ($cmoves(B)$). If values produced in block B are live on the block's exit, then there may also be an execution overhead to consider. At minimum, we pay a one cycle cost for the added `cmove`, but the cost is even greater if B's side-effecting instructions have a greater dependency height than A's. As above, we estimate the overhead of merging branches A and B as the maximum of these overhead costs.

$$oh_{fetch} = \frac{size(B)p_{fo}(a) + cmoves(B)}{IF\_width} \quad (6)$$

$$max\_ht_{B-A} = max\_ht(B) - max\_ht(A) \quad (7)$$

$$oh_{execute} = \max(max\_ht_{B-A}, 0)p_{fo}(a) + 1 \quad (8)$$

$$overhead = \max(oh_{fetch}, oh_{execute}) \quad (9)$$

Knowing when and how merging conjunctive branches will improve the misprediction rate, p_{misp}, is less clear (at least without generating the if-converted code and re-profiling). We find that difficult-to-predict branches often have prediction rates that correspond closely to their branch biases (*i.e.*, a difficult-to-predict branch with a 70% bias is likely to have a prediction rate around 70% and, hence, a misprediction rate of 30%). Figure 6 plots the difference between branch bias and prediction rate for problem branches — those with more than 500 mispredictions and less than 80% prediction accuracy — across all of the benchmarks. The cumulative distribution indicates that 70% of dynamic branches belong to static branches whose bias is within 6% of the branch's prediction accuracy.

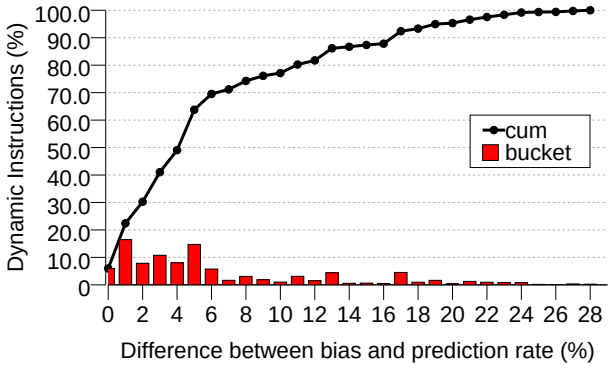

Fig. 6. The bulk of dynamic instances of problem branches are from static branches whose bias is within 6% of prediction rate.

If we use bias as a proxy for prediction rate, we can map out the conditions required for an improvement in misprediction rate. With regard to Figure 4, we would expect branch A to be mispredicted about $\min(a, a')$ times and branch B to be mispredicted about $\min(b, b')$ times. The combined branch goes to the fall-out block $a' + b'$ times and to the advance block b times, so we would expect $\min(a' + b', b)$ mispredictions in the if-converted code, yielding a misprediction savings of:

$$Misp\_savings = \min(a, a') + \min(b, b') - \min(a' + b', b) \quad (10)$$

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.

We have also observed situations in which if-conversion exposes new correlations for branch predictors to exploit. Figure 7 shows an example from the benchmark `gap`, where branch 3 — the second branch in a conjunctive pair — has the same condition as branch 1. In this region the dominant path is shown by the heavy lines, which indicates that control frequently passes around branch 3. Despite branch 2's bias, it is still responsible for many mispredictions, many of which are eliminated by exposing the correlation with branch 3 through if-conversion. We have not yet implemented compiler support to detect these correlations.

Heuristic Effectiveness As we discuss in Section V, we find these heuristics to be good predictors of the best if-conversion, except in a few instances where they prove

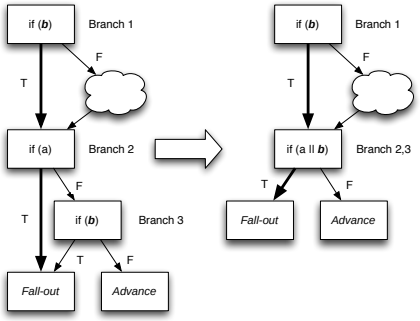

Fig. 7. Improving branch correlation by combining branches.

too conservative. These cases occur when if-conversion enables other optimizations (*e.g.*, constant propagation and dead code elimination). This could be resolved by generating a version of the if-converted code for use by the heuristics (as was previously done for hammocks [6]), but this was not easy to do in our infrastructure.

IV. Experimental Method

We used an offline feedback-directed optimization (FDO) methodology, as shown in Figure 8, using the *Low-Level Virtual Machine* (LLVM) compilation framework [10] and a SimpleScalar-based simulator [11]. Our experiments involved two phases: 1) a profiling phase to collect data for applying the selection heuristics from Section III, and 2) an experimental phase where we transformed code regions in response to the heuristics and evaluated the performance and energy gains.

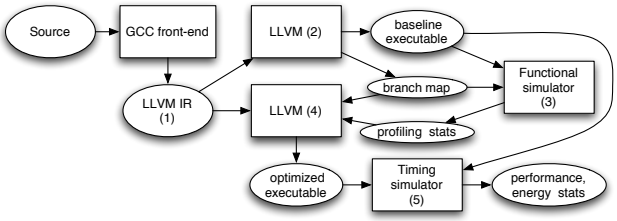

Fig. 8. Flow chart of our feedback directed optimization (FDO) setup

In the first phase, we compiled the Spec2000 integer benchmarks to LLVM byte code (step 1) and created analysis passes to identify the previously described branch structures. The branches in these structures were marked with annotations that are carried through code generation (step 2), allowing the construction of a PC-to-IR (internal representation) map to enable decorating the IR with profile information. A functional simulator (step 3) was used to collect branch biases, misprediction rates, and trip count distributions (for loops) to guide the heuristics.

In the second phase, we performed the profile-guided if-conversion to generate an optimized executable (step 4), which was simulated on a a SimpleScalar-based out-of-order microprocessor timing simulator (step 5). We chose an aggressive, but energy conscious configuration for the simulator: 4-wide superscalar with a 128-entry RUU. We used a combined two-level (4096-entry, 8-bit history) and bimodal (2048) predictor, with a 1024-entry chooser. The branch target buffer, return address stack, and memory dependence predictor are simulated as perfect, and the minimum branch misprediction penalty is 12 cycles. We simulated a 64KB, 4-way L1 I-cache, a 2-ported 32KB, 2-way L1 D-cache, backed up by a unified 1MB 4-way L2 cache and a 200-cycle main memory. CPU energy usage was estimated using Wattch [12] and its *cc_3* non-ideal clock gating measurement.

A. Path Length Determination

As different versions of code will have varying path lengths, it is important to ensure that each version is performing an equivalent amount of work. Because full benchmark runs were infeasible, we instead used the ability of our simulator to "mark" branches and used instances of an un-transformed branch to monitor the progress of the benchmark. We used these branch executions to designate corresponding start and end points in the baseline and if-converted code. For each experiment, we picked three trace intervals in which the branch is active that span the range of the branch's misprediction rates; for example, the first branch in Figure 9 has misprediction rates of 23.0, 23.4, and 23.5% in the first, second, and third intervals, respectively. The runs of the baseline code are roughly 100 million instructions long; the path length of the if-converted code depends on the degree of instruction overhead. 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.

V. Experimental Results

Because our goal in this study was to explore the impact of branch transformations on performance and energy, we have chosen to present results where we transform a single branch (or group of branches) for each experiment. In contrast to measuring aggregate results for many transformations in a benchmark, we feel this gives 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 represents the subset of Spec2000 integer benchmarks that runs

in our simulation infrastructure: `gzip`, `mcf`, `crafty`, `parser`, `bzip2`, and `twolf`.

We first present hammock results to demonstrate the energy implications of if-conversion (Section V-A), but the bulk of this section focuses on the less-explored loop unrolling (Section V-B) and conjunctive branch combining (Section V-C). For each experiment, we collect four data points that are plotted together in the same graph. The first three columns compare the number of cycles, path length, and number of instructions fetched to those measurements in the baseline configuration. The fourth column compares the energy consumption to that of the baseline code. In all cases, the results are normalized to an optimized binary without profile-directed if-conversion.

A. Hammock Evaluation

To demonstrate the performance and energy benefits of if-conversion, we first consider the four hammocks shown in Figure 9. These hammocks are taken from the benchmark `twolf`, which had the highest incidence of mispredictions from simple hammocks (39%, see Figure 1), but they are representative of the hammocks selected by our heuristic. Our first observation is that a very wide processor is not necessary to exploit if-conversion; our 4-wide machine achieves a significant benefit. Even these hammocks only make up 1-3% of the instructions executed in the sampled interval (coverages are provided in the caption of Figure 9), if-conversion achieved speedups of 1-3%, basically a return linear with coverage.

A second observation is that energy savings generally lags the speedup achieved by if-conversion. If-conversion generally represents a trade-off between a reduction of branch mispredictions and an increase in the number of instructions retired. Because much of the dynamic instruction count increase is in the form of instructions that can be executed in parallel with the existing critical path, much of the savings due to eliminating branch mispredictions translates directly to reduced execution time. In contrast, energy consumption for a given machine is roughly linear with the number of instructions executed [14]. While if-conversion reduces the total number of instructions fetched (the third bar in Figure 9), it is not reduced by as much as performance is improved. Furthermore, this reduction is partially offset by the increase in correct path instructions (column two), yielding only a modest energy improvement in many cases. In general, we find that energy consumption is well correlated to the number of instructions fetched.

B. Loop Peeling Evaluation

In this subsection, we present two examples that illustrate the following observations about loop peeling. First, loop

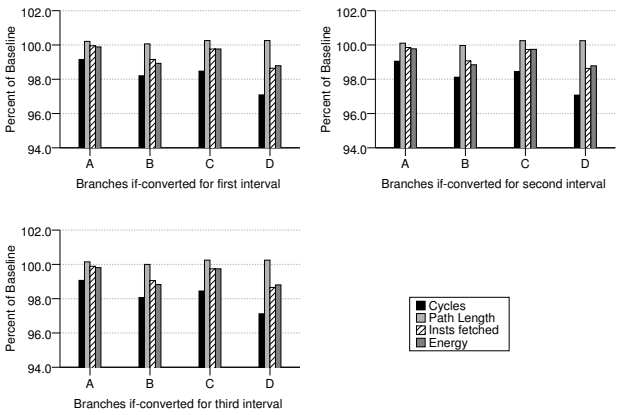

Fig. 9. Results for if-converting simple hammocks in `twolf`. In the intervals we measured, the coverage for hammocks A, B, C, and D averaged 1.13%, 1.62%, 2.10% and 2.80%, respectively. Hammock coverage is measured as a percent of the overall dynamic instructions in the interval.

peeling generally enables significant optimization opportunities of constant propagation and dead code elimination. The inability to anticipate these effects prevented us from selecting the optimal degree of loop peeling. Nevertheless, because our heuristic is conservative, it identified peeling factors that resulted in a performance gain in almost every case. Second, we observed an even weaker correlation between speedups and energy savings than we did in hammocks. In fact, energy consumption correlates more closely with path length than with the number of mispredictions saved, suggesting that loop peeling is effective for saving energy insofar as it enables the path length to be reduced through constant propagation and dead code elimination.

Loop E is a single basic block loop in an implementation of `quicksort` that is responsible for 6% of the overall branch mispredictions in `mcf`. It exhibits the case where its guarding branch is frequently mispredicted. Our loop peeling heuristic advises to if-convert this guarding branch, but to not otherwise peel any iterations, resulting in the speedup shown in column 0 of Figure 10. Although performance is improved, energy is slightly increased. Roughly the same performance, but with lower energy consumption, can be achieved by peeling the first iteration; performance can be improved with two peels. This was not anticipated by the heuristic, because it overestimates the overhead of peeling. Note the drop in path length from 0 peels to 1 peel that results from optimizations enabled by the control-flow transformation.

Our second example, loop F from `twolf`, is an interesting example that shows how loop peeling benefits from constant propagation. Figure 11 shows there is a *savings* in path length even for peeling factors that exceed the

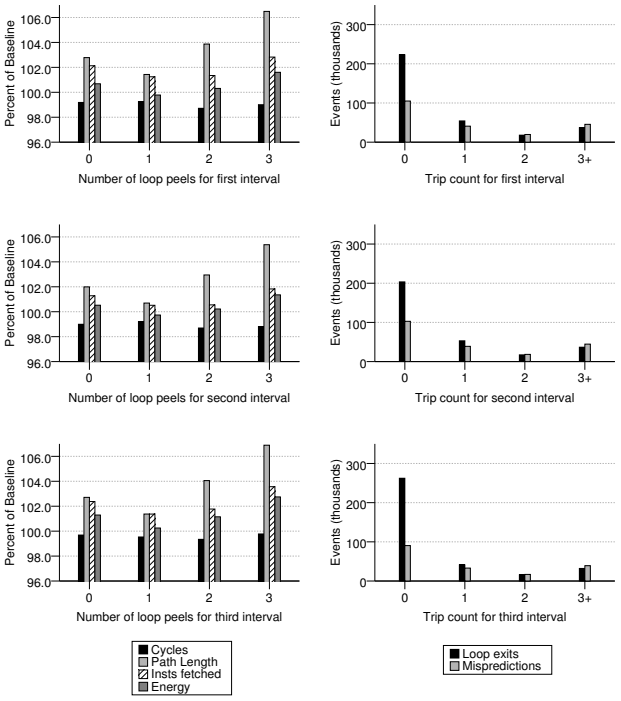

Fig. 10. Loop peeling factors for loop E in `mcf`. The instructions in this loop are responsible for around 4.3% of the dynamic instructions in the trace intervals we simulated.

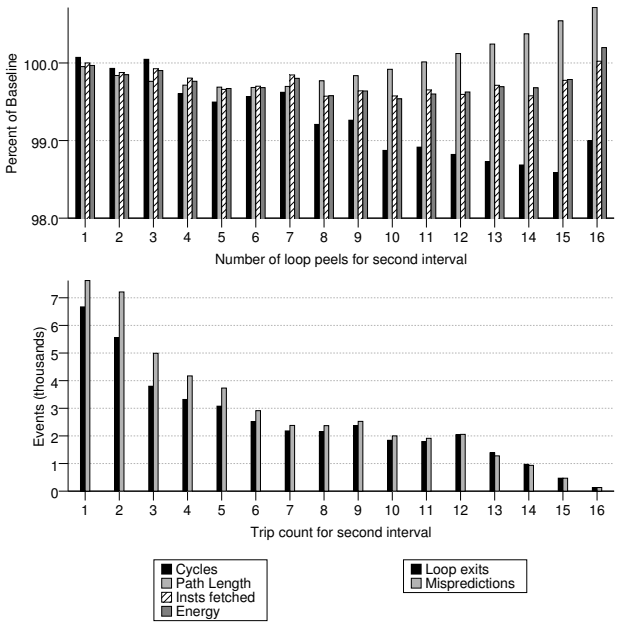

Fig. 11. Loop peeling factors for loop F in `twolf`. As all three intervals were almost identical, we show only one interval to conserve space. The instructions in this loop are responsible for around 1.6% of the dynamic instructions in the trace intervals we simulated. The trip count for this loop never exceeds sixteen.

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 optimization eliminates three add instructions per iteration for a loop body that began with only seven instructions. As a result, both performance and energy are improved. Because our heuristic could not anticipate the path length improvements it chose to peel eight iterations, but performance continues to improve out to fifteen peels.

C. Conjunctive Branch Evaluation

```
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));
}
```

Fig. 12. Source listing of function *bea_is_dual_infeasible* in `mcf`

The effectiveness of our conjunctive branch merging heuristic can be well demonstrated by an example from `mcf`. The *bea_is_dual_infeasible* function (shown in Figure 12), contains two conjunctive branch pairs. The misprediction rates of these branches roughly correspond to their branch biases (shown in Figure 13). Merging branches W and X results in only 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 (see Figure 13). Furthermore, 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.

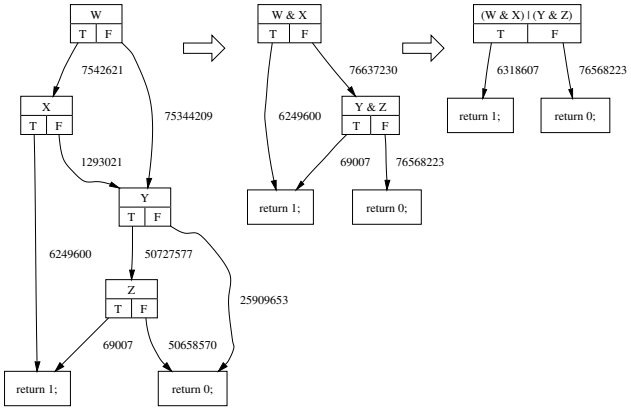

Fig. 13. Improving branch bias by combining branches. Combining branches Y and Z provides the most significant improvement in net bias.

As a result, our heuristic selects only the YZ branch pair for if-conversion, as the others are not expected to save

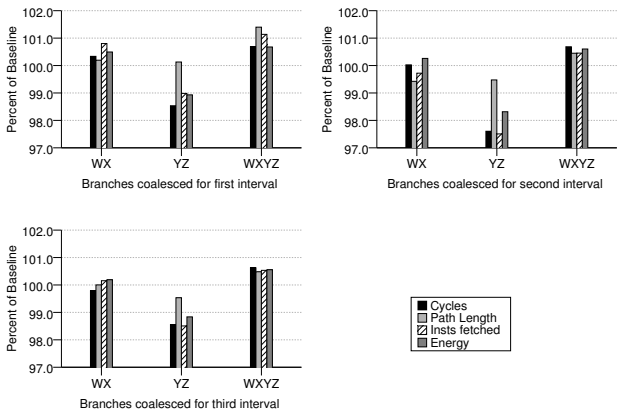

Fig. 14. Conjunctive branch evaluation. The instructions in this function are responsible for around 7% of the dynamic instructions in the trace intervals we simulated.

enough mispredictions to amortize the instruction fetching and execution overhead. When we run the transformed code, we observe that the prediction rates for the combined branches are in fact well predicted by their bias: the branch WX has bias of 92.5% and a prediction rate of 91.4%, the YZ branch has a bias of 99.9% and a prediction rate of 99.9%, and the WXYZ branch has a bias of 92.3% and a prediction rate of 91.3%. Figure 14 confirms that combining branches Y and Z offers a 1-2% reduction in cycles and energy, while the other combinations offer negligible improvements.

VI. Conclusion

In this paper, we made four primary contributions: 1) we explored if-conversion beyond hammocks, describing heuristics and experimental results for if-converting loop back edges and conjunctive branches, 2) we demonstrated that together these if-convertible regions cover a significant fraction of mispredictions (over 50% on average) in most benchmarks and such if-conversion is a viable technique even in energy-conscious superscalar processors, 3) we make the observation that the bias of a hard-to-predict branch serves as a good proxy for its prediction rate, and 4) we demonstrated that the energy benefits of if-conversion are generally not as profound as its performance improvements due to an increase in path length. Energy benefits that accrue do so at a higher performance point, yielding significant improvements in energy-delay metrics. If voltage-frequency scaling is available, it could be used to translate the performance benefits into energy savings at the baseline performance level.

VII. Acknowledgments

We thank Charles Tucker for his assistance in collecting the power numbers presented in this work and the anonymous reviewers for their feedback on this work. This work was supported in part by NSF grants CCR 03-11340, CCR 03-11340 REU, CCF 03-47260, CCF 04-29561, NSF Instrumentation grant EIA-0224453, a gift from the Intel Corporation, and an equipment grant from HP.

References

[1] 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 *25th Annual International Symposium on Microarchitecture*, 1992.

[2] 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.

[3] 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.

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

[5] 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.

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

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

[8] A. Cristal, O. J. Santana, M. Valero, and J. F. Martinez, "Toward kilo-instruction processors," *ACM Transactions on Architecture and Code Optimization*, vol. 1, pp. 389–417, December 2004.

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

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

[11] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An infrastructure for computer system modeling," *IEEE Computer*, vol. 35, pp. 59–67, Feb. 2002.

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

[13] 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.

[14] M. Valluri and L. John, "Is compiling for performance == compiling for power," in *The 5th Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT-5)*, January 2001.

[15] 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.