A Lightweight Algorithm for Dynamic If-Conversion During Dynamic Optimization

Kim M. Hazelwood Thomas M. Conte
Department of Electrical and Computer Engineering
North Carolina State University
{kim_hazelwood,conte}@ncsu.edu

Abstract

Dynamic Optimization is an umbrella term that refers to any optimization of software that is performed after the initial compile time. It is a complementary optimization opportunity that may greatly improve performance on any computer system, but plays an especially important role in statically scheduled code. Several groups are working on developing dynamic optimization systems, yet the area of dynamic optimization algorithms can still benefit from further research. We introduce a lightweight algorithm that can be used in any modern dynamic optimizer to balance control flow and predication based on actual runtime behavior. In addition, we study the effectiveness of predicting overall runtime behavior based on a small sample size. Preliminary results show that if we skip the warm-up period of programs, profiles based on a small sample size of a particular run can be quite representative of overall runtime behavior (up to 98% correlation). This profile information can be used effectively in a number of dynamic optimizations. We found that our dynamic if-conversion algorithm can use this collated profile data to incorporate actual branch misprediction rates into the if-conversion decision process. This method acts as an effective means for balancing the results of static if-conversion, achieving speedup values of up to 14.7%, and can be easily incorporated into modern dynamic optimizers.

1. Introduction

Dynamic Optimization refers to any program optimization performed after the initial compile time. While typically not designed as a replacement for static optimization, dynamic optimization is a complementary optimization opportunity that leverages information that is not available until runtime. Dynamic optimization opens the doors for machine and user-based optimizations without the need for original source code.

The challenge of dynamic optimization rests mostly in the performance requirements for runtime optimization. The baseline goal of a dynamic optimization system is that the system must not slow a program down. While it may be the case that users will permit a slowdown if it is a one-time occurrence and the performance of the system is improved to the extent that the average runtime over subsequent runs is improved, there is a more defined need for simple, low-overhead optimization algorithms that are specialized for runtime implementation. The overall goal of such algorithms would be to require minimal effort in order to produce runtime speedup.

The purpose of this paper is to present a lightweight algorithm that can be implemented in modern dynamic optimizers. Dynamic if-conversion is a term used in this paper to refer to the dynamic version of static if-conversion. While not designed as a replacement for static if-conversion in any manner, dynamic if-conversion is a complementary, lightweight algorithm that can be used to effectively balance control flow and predication based on actual runtime behavior. The distinguishing feature of dynamic if-conversion is the fact that it takes into account the actual misprediction rate of a branch when deciding whether or not to convert it to a set of predicated instructions or reverse previously if-converted code. The algorithm is simple in order to allow for implementation in a system with stringent run-time overhead requirements. It was designed to be performed on architectures that support predication yet require static scheduling, such as EPIC and VLIW with predication architectures.

The paper also delves into the tradeoffs involved in the time at which optimizations based on branch behavior are employed. Due to the high overhead of dynamic profiling, dynamic if-conversion can make use of sample dynamic profile information gathered near the beginning of program execution for the dynamic if-conversion decision process. The accuracy of the sample profile data for several sampling heuristics is studied in this paper.

Several factors may lead to the decision to implement a dynamic optimization infrastructure. Often, decisions made by a static compiler are not well suited for the

runtime behavior of the program. This may be due to a variance in the usage patterns of the client, a change or upgrade in hardware (such as the pipeline structure) since the initial compile time, or simply an overaggressive static compilation decision. The current software development cycle does not provide an opportunity for changing or correcting such decisions made by the static compiler. Dynamic optimization can provide this flexibility in the software development cycle.

While most of the processors of the past rely on dynamic scheduling of instructions, an upcoming generation of processors contains a simpler processor core that relies on static scheduling. Scheduling programs once, and even before the program executes, increases the importance of accurate profile data. Inaccurate profiles result in poor scheduling decisions that will affect every run of a program [24]. Representative profile information is difficult to produce because of the large variance among users, and over time for the same user. This sheds doubt on the effectiveness of a unified set of profile data. Using dynamic optimization, code can be rescheduled to better represent actual runtime behavior for every run of a program.

Another justification for a dynamic optimization system concerns the large number of off-the-shelf software packages that are purchased every year. The varying level of compile technology available in the compilers used by software vendors has resulted in an unknown level of optimization in off-the-shelf software products. Consumers have no way of ensuring that high optimization levels are employed in the products they purchase. Dynamic optimization allows even the latest optimization technology to be introduced to existing software, therefore ensuring optimal performance.

Yet another reason for choosing to implement dynamic optimization concerns processor upgrades within a processor family and cross-generation compatibility between processor families [5][6][24]. A program compiled for a given instruction-set architecture (ISA) can be run on any processor implementing that ISA. The program is only optimized, however, for execution on a processor with an identical execution pipeline, identical function units, and identical instruction availability. Variances in any of these attributes result in a program that is not employing the available functionality of the processor. Dynamic optimization provides for customized optimization for every processor within an ISA from a single executable.

The remainder of the paper is organized as follows. Section 2 introduces some of the modern dynamic optimizers that could potentially employ dynamic if-conversion, discusses the problems with implementing static-only if-conversion, and introduces the simulation environment that was used to obtain our results. A study of the optimal time to perform certain dynamic optimizations is discussed in Section 3 including a full description of two case studies into the matter. The dynamic if-conversion and reverse if-conversion algorithms and results are presented in Section 0. Finally, Section 5 concludes the paper and discusses future work.

2. Background

Many research groups are currently working on developing infrastructures for dynamic optimization in one of its many forms. Some groups work exclusively within a given ISA, gearing their optimizations toward reoptimizing for differing features of processors within a processor family or user behavior [4][9][17][24][28]. Others work between ISAs. In this case, the final product does the job of translating or emulating one instruction set on a different instruction-set architecture [6][11][14].

In this work, lightweight, dynamic algorithms that can be employed in any modern dynamic optimization system are introduced. A new dynamic optimization infrastructure is not proposed because the algorithms presented in this paper are well suited for implementation in the systems that currently exist. Because seamless integration of the algorithms with current infrastructures is expected, it is important to first become familiar with several dynamic optimization infrastructures that are under development in the research community.

Dynamo, a dynamic optimization system created at Hewlett-Packard Laboratories is a native instruction interpreter that optimizes program fragments (executable program traces) and stores them in a software-based fragment cache at runtime [4]. Based in the HP-UX environment, Dynamo is transparent to a user, yet achieves notable program speedup even without the use of runtime optimizations. The base speedup is due to reduced instruction cache misses for hot program fragments found in the fragment cache. It is shown that programs originally compiled with level-2 optimization (-O2) running under the Dynamo dynamic optimization system perform as well as the same program compiled with level-4 optimization (-O4) running without dynamic optimization [4].

Morph, a dynamic optimization system developed at Harvard University consists of several components that perform the tasks of observing program behavior and triggering optimization for Digital UNIX [28]. An extension to the back-end compiler produces a shared library containing useful information that may aid in later reoptimization. A Morph manager observes data gathered by a low-overhead profiling system built into Digital UNIX and decides when to invoke optimization.

Researchers at the University of Washington have developed an infrastructure for dynamic compilation called DyC [9]. DyC consists of a declarative annotation language and corresponding compiler. DyC's static compiler produces an executable that contains a combina-

tion of statically compiled code and a run-time specializer for the portions of code where dynamic com-pilation will occur. The run-time specializer allows code to be optimized for various instances of run-time variables. A dynamic template is created containing holes that will be filled in at dynamic compile time, once the runtime values are known. Custom dynamic compilers are implemented within the code itself, which trigger the dynamic compilation and throw away the corresponding template.

Other research endeavors that may fall under the umbrella of dynamic optimization include, but aren't limited to, DAISY [6] and Crusoe [14]. DAISY is a set of hardware features used to emulate code from existing architectures on VLIW and other ILP machines. Because code is dynamically translated and stored in a hardware-based cache, dynamic optimizations may be applied to the cached code. Crusoe consists of a low-power VLIW processor and code-morphing software that performs the task of translating x86 code to the native instruction set on the fly. Dynamic optimizations are easily implemented in the code-morphing software.

Each of the dynamic optimizers mentioned in this section would require minimal adjustment to employ the dynamic if-conversion algorithm presented later in the paper. The justification for implementing this runtime optimization requires an understanding of the limitations of the current practice of static-only if-conversion.

2.1. The Problems With If-Conversion

If-conversion replaces a branch and its control dependent paths with guarded (or predicated) execution [2][21]. Rather than evaluating a branch condition and conditionally executing only one control-dependent path, both paths are executed and predicates control which results are used. It is a means for converting control dependences into data dependences.

```
        BEFORE:
(1) if (cond) Branch L1
(2)  r2 = MEM[A]
(3)  r1 = r2 + 1
(4)  r0 = MEM[r1]
(5) L1 : r9 = r3 + r4

        AFTER:
(1) p1, p2' = cond
(2) r2 = MEM[A] <p2>
(3) r1 = r2 + 1 <p2>
(4) r0 = MEM[r1] <p2>
(5) L1 : r9 = r3 + r4
```

Figure 1. A standard if-conversion example.

Much of the current research involving if-conversion is based on the work done at Hewlett-Packard Laboratories. Park and Schlansker present a basic if-conversion algorithm called the RK algorithm [21]. The R function specifies which instructions should be based on the same predicate value, while the K function indicates the conditions under which a predicate should be set to true.

If-conversion avoids the penalty incurred when a branch is otherwise mispredicted. It is not a good idea to if-convert all branches; in certain instances, it is better to branch over a large set of instructions rather than conditionally executing them. If-conversion works best when attempts are made to reach a balance between control flow and predication [3]. Yet a perfect balance is not attainable at static compile time because little information is known about the actual behavior of a particular branch.

Mahlke et al. introduced compiler support for predicated execution using the hyperblock [19]. The hyperblock is used to increase the scheduling scope by allowing the inclusion of speculative execution to predicated instructions while also allowing for selective inclusion of basic blocks based on frequency and size. Mahlke then studied the impact of predicated execution on branch prediction and discovered that 56% of dynamic branch mispredictions are eliminated with predication support [18]. Our work may be viewed as an extension of the work done by Mahlke into the dynamic domain.

In addition, several other researchers have delved into if-conversion. One example is the work of Klauser et al. [15]. They describe a method for dynamically introducing predication to architectures that do not already support it. While their main objective was to provide hardware support for the dynamic introduction of predication into programs, our main goal is to use software and an ISA that already supports predication.

2.2. Our Simulation Environment

The results of this paper are obtained using an EPIC-style execution-driven simulator operating on the SPECInt95 benchmarks using the reference inputs. The LEGO back-end compiler [10][16], which is based on the HPL PlayDoh Architecture [13], was used to schedule the benchmarks. Treegion scheduling [10] was used, combined with static if-conversion and most modern optimizations that would be present in an EPIC compiler such as loop unrolling, operation combining, aggressive list scheduling, upward and downward code motion, and support for multi-way branching. The branch predictor modeled is a hybrid predictor (with four hybrid counters) containing a PAS predictor (with 2^{12} branch history registers, each containing 10-bits of history information and 2-bit prediction counters) and a Gshare predictor (with 2^{12} rows in the Gshare table, 10-bits of history and 2-bit prediction counters) [7][27]. The predictor has an average accuracy of 95% on the SPECInt95 benchmarks and a misprediction latency of 10 cycles [12].

3. Deciding When to Dynamically If-Convert

A challenging aspect of dynamic optimization is deciding if and when to optimize a portion of code. Dynamic optimizations should be performed as soon as the profile information is available, yet it must be ensured that the profile information is indicative of overall program behavior. To further complicate the decision-making process, time should not be wasted optimizing rarely executed code. Instead, hot spots should be located in order to focus on optimizing active portions of code. These uncertainties warrant a study of the most favorable optimization opportunities.

The dynamic if-conversion algorithm relies heavily on the misprediction rate of the branches during the current execution. Not only is it important to gather accurate branch misprediction rates, but the profile data must also be gathered early enough for the program to benefit from dynamic if-conversion. Because dynamic profiling might not be free (as in the case of Dynamo), the misprediction rate of a branch is sampled and dynamic if-conversion decisions are based on that sample rate. Therefore, an important question is: *How representative of the overall misprediction rate is a sample misprediction rate?* The answer to this question varies depending on the heuristic used to collect the statistics.

3.1. Sampling Based on First *N* Occurrences

One scheme for gathering sample misprediction data is fairly straightforward. The misprediction rate for the first n occurrences of a branch is tracked, then it is assumed that the behavior of that branch will follow the same trend in the future. Figure 2 shows the percent difference between sampled and full misprediction rates, averaged over all branches for the SPECInt95 benchmarks.

The results in Figure 2 were determined by comparing the misprediction rate of various sample sizes (25, 50, 75, 100, 125, 150) to the actual misprediction rate of each branch throughout the entire run. The sample sequences are successive outcomes of a particular branch taken at the beginning of program execution.

The rates were viewed as a set of Bernoulli trials [8] and Bernoulli distributions were determined. A Bernoulli distribution is a means for representing a trial with two possible outcomes, in our case a misprediction or a correct prediction. Next, the two distributions – the overall outcome and the sample outcome, were compared using the Kolmogrov-Smirnov test [1]. A 99% confidence interval was chosen to determine the difference between the two distributions.

Figure 2 shows that the average difference in the misprediction rate of the SPECInt95 benchmarks for a sample size of 25 branch occurrences, when compared to the overall behavior of the branch, is around 7%. This would equate to a misprediction rate of 17% in the sample run as compared to a 10% misprediction rate overall. If the sample size were increased to 150 branch occurrences, the average difference drops down to around 2%.

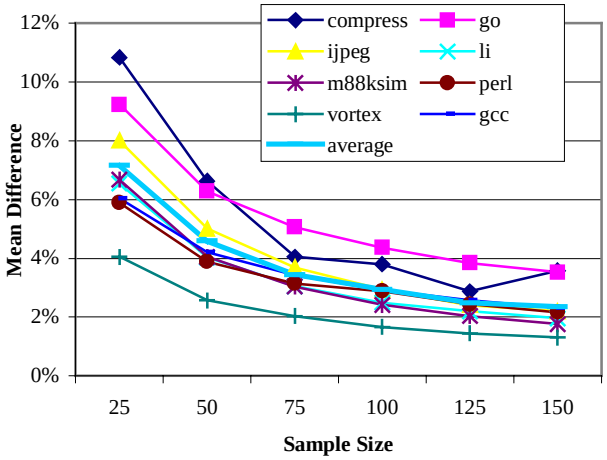

Figure 2. The mean difference between sample (first *n* occurrences) and actual misprediction rates.

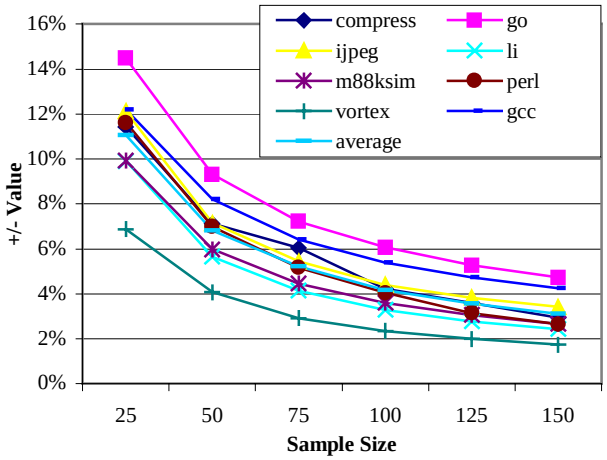

Figure 3. Error values corresponding to a 99% confidence interval.

A notable factor is that differences shown in Figure 2 have a delta error value when compared to the behavior of a single branch because they are averaged over all branches. These error values are shown in Figure 3. The error values in Figure 3 encompass a 99% confidence interval. For example, we are 99% certain that a particular branch in the vortex benchmark will result in a misprediction rate within 4% +/- 7%, or between -3% and 11% of the averages shown in Figure 2 (for a sample size of 25). Figure 3 allows us to calculate the worst-case scenario that can be expected when comparing sample misprediction rates to actual rates for any given branch.

An important conclusion that can be drawn from Figure 2 and Figure 3 is that choosing the first n branch

occurrences as the sample for estimating overall branch misprediction rate can be quite inaccurate for small values of n such as 25 or 50. In the worse case, inaccuracies of up to 26% may be seen. Sample sizes of 75 or greater result in error values of less than 4% on average with an error of less than 4% (or between 0% and 8%). Yet as the sample size is increased, the benefits gained from dynamic optimizations are reduced because the point at which the optimizations are performed is delayed.

3.2. Adaptive Warmup Exclusion

Another approach for sampling misprediction rates involves recognizing an end-of-warm-up (confidence) condition and only then beginning to collect misprediction statistics. Section 3.1 described a sampling heuristic that compared the first *n* outcomes of a particular branch to its overall outcome statistics for the entire program duration. The problem with this approach is that the data was gathered at the start of the program. The start of any program typically has a high misprediction rate as the branch predictors learn branch patterns. This behavior may not be representative of the entire program behavior because the misprediction rate of a particular branch often decreases notably after the warm-up period, as shown in Figure 4 for several hard-to-predict branches in SPECInt95.

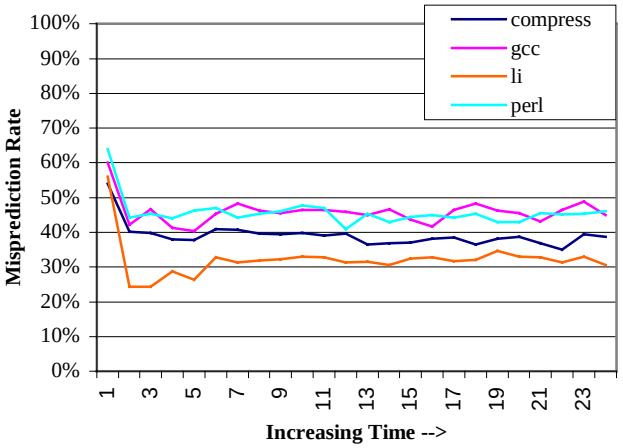

Figure 4. The misprediction rate over time (in increments of 50 occurrences) for the first hard-to-predict branch encountered for four benchmarks in SPECInt95. A branch is considered hard-to-predict if the final misprediction rate exceeds 30%.

The adaptive warmup exclusion sampling heuristic attempts to recognize the stabilization of the branch predictors on a branch-by-branch basis. Only when a branch reaches the end of its warmup period does the profiling of misprediction statistics for the branch begin. Equation 1 describes the heuristic for detecting an end-of-warmup condition. This equation attempts to recognize an end-of-warmup condition as the point at which the branch misprediction rate settles to within a threshold value (10% for example) of the previous rate. The results of testing this heuristic with a threshold value of 10% are shown in Figure 5. The graph shows us that by ignoring the warm-up period of the branch predictors, our average sampling accuracy improves by 43% (dropping from 7% difference to 4% difference).

$$| P_{MISS\_A} - P_{MISS\_B} | < T \qquad (1)$$

P_{MISS_A} = last misprediction rate
P_{MISS_B} = this misprediction rate
T = threshold

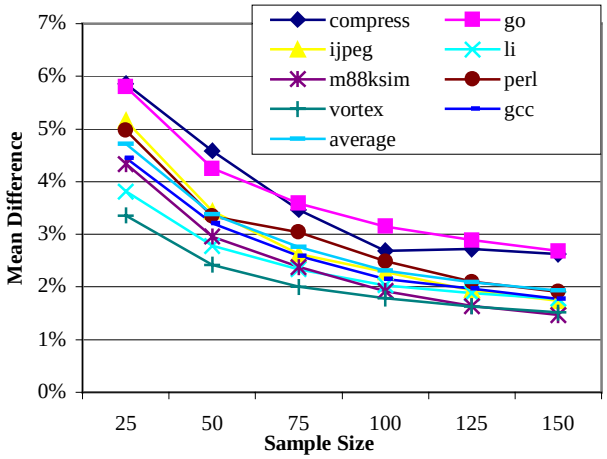

Figure 5. The mean difference between sample (first *n* occurrences ignoring warm-up period) and actual misprediction rate.

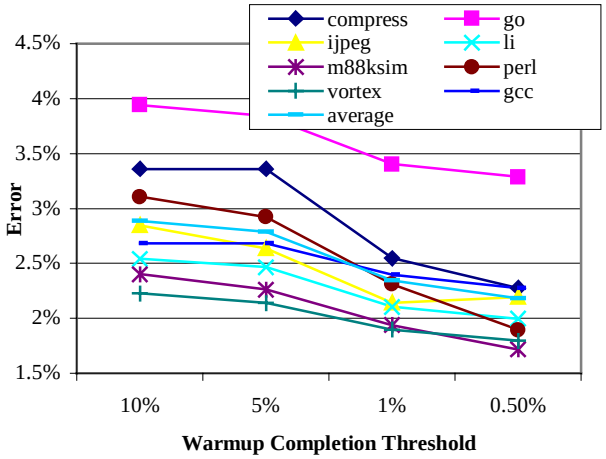

Figure 6. The effect of varying the warmup completion threshold value.

Figure 6 shows the effect of holding the sample size at a single value (50 occurrences) and varying the threshold value from 10% to 5% 1% and 0.5%. As the graph

indicates, varying the threshold value results in minimal changes to the mean difference between the sample and actual misprediction rate. Figure 7 shows the average number of branch occurrences required to trigger the end-of-warmup condition.

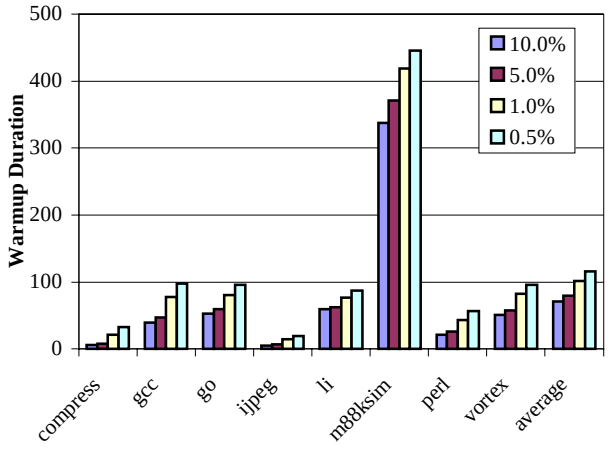

Figure 7. Number of branch occurrences before reaching end-of-warmup condition.

This study of the dynamic sampling heuristics provides insight into the effectiveness and accuracy of dynamic optimizations that are performed based on sampled statistics. If the sampling heuristic is inaccurate, any dynamic optimizations that make transformations based on sample data are likely to be inaccurate or unnecessary and may hurt overall program performance.

4. The Dynamic If-Conversion Algorithm

Dynamic optimizations may be classified as either heavyweight or lightweight [4]. Heavyweight optimizations make the largest impact on program performance. Not surprisingly, heavyweight optimizations take the largest amount of time to perform, and therefore may not be feasible to perform at runtime. Lightweight optimizations, on the other hand, can typically be performed at runtime with a minimal impact on overall performance.

As mentioned earlier, dynamic if-conversion is a lightweight, runtime equivalent of static if-conversion. It can be used to if-convert branches that are performing poorly during a particular run. It can also be used to ensure that if-conversion is performed at all, as some compilers may not have employed static if-conversion at compile time.

Dynamic if-conversion is based on a one-time opportunity for each control-flow sequence. Sample profile data is used to make one decision regarding dynamic if-conversion for a particular branch or predicated trace. While the algorithms may undo static if-conversion decisions, they do not undo dynamic if-conversion decisions. For example, static branches may be converted to dynamic predication and static predication may be converted to dynamic branches, but once converted no attempts are made to reverse the decision. Since no attempts to undo dynamic if-conversions are made, the machine state doesn't risk becoming unstable due to continuous if-conversion and reverse if-conversion. Therefore, a bailout mechanism is unnecessary for this particular model.

4.1. Dynamic Forward If-Conversion

An opportunity for branch-to-predicate conversion, or dynamic forward if-conversion, lies in branches that exhibit hard-to-predict behavior. Actual branch behavior is not known until runtime, and it may vary from one run to the next [24]. A static decision whether or not to predicate sections of code will clearly be inferior to one that takes into account the current behavior of a branch during a particular run.

Deciding when to dynamically convert a branch to a set of predicated instructions is a matter of estimating the cost-effectiveness of doing so. First of all, the branch must be a forward branch. This restriction is put in place because backward loops typically do not benefit from if-conversion, as this would require completely unrolling the loop [3]. Not only are loops highly predictable, but they always require a branch before the epilog. This branch cannot be converted to predication. In the case of loop unrolling, the unrolled portions of a loop that end up benefiting from branch conversion become forward branches in the unrolling process.

The second requirement is that a candidate branch must be on a hot path. The cost of converting a branch to predicated code is typically amortized over the subsequent uses of the section of code. Therefore, time and effort should not be wasted on a branch that is not likely to be encountered frequently.

$$\mathbf{P_{MISS} * L_{MISS} \geq P_{HIT} * L_{HIT} * (1+error)} \qquad \mathbf{(2)}$$

P_{MISS} = odds of mispredicting branch
L_{MISS} = misprediction penalty
P_{HIT} = odds of correctly predicting branch
L_{HIT} = cycles to execute predicated instructions

Deciding whether or not to convert a branch to a set of predicates is a matter of comparing the average latency of either option. Equation 2 describes the dynamic if-conversion decision. The left side of the equation estimates the average penalty for leaving the branch as is. The right side estimates the overall cost of removing the branch and predicating the following set of instructions. The error value mentioned in the right side of the equation compensates for the error involved in determining the branch misprediction rate as described in Section 3.1 and Section 3.2. Since only a sample of the misprediction rate

has been gathered when Equation 2 is evaluated, the error term compensates for the error with a 99% confidence interval.

Branches over very large sections of code will typically not be considered for forward if-conversion because the cost of executing all of the predicated instructions on the off path far outweighs the benefits of removing the branch misprediction penalty. In the case of IA-64, where the branch misprediction penalty is 10 cycles [12], most of the converted branches will have a target address that is within 10 cycles of the branch instruction itself. Higher distances are permitted in the case of unusually high misprediction rates. The maximum value of the allowable branch distance is shown in Equation 3.

$$A_T - A_B > 0 \qquad (3)$$
$$A_T - A_B < L_{MISS} * P_{MISS} * S_{INSTR}$$

A_T = target address A_B = branch address
L_{MISS} = miss penalty P_{MISS} = miss rate
S_{INSTR} = instruction size

The basic concept is that the latency of a few predicated instructions may be much smaller than the latency of a potential branch misprediction. And if a branch misprediction is likely to occur, it is better to convert that branch to predicated instructions. The potential for a branch misprediction is then eliminated and, often, the predicated instructions can be scheduled into the holes (NOPs) of the existing schedule.

Figure 8 and Figure 9 show the speedup gained from employing dynamic if-conversion as well as the actual number of branches that were converted to predicates for SPECInt95. The overhead of dynamic if-conversion varies depending on the implementation details of the dynamic optimizer. The results were produced including an estimate of the additional overhead that dynamic if-conversion would incur over the base overhead of dynamic optimization.

There was a great deal of variance in the number of branches converted (from 1 to 215) and the corresponding speedup (from 0.16% to 14.7%), yet no slowdown was detected. On average, 46 branches were converted to predicates for a speedup of 2.5%. Since 95% of SPECInt95 branches are predicted correctly using the PAS/Gshare predictor, dynamic if-conversion is designed to reduce the latency of 5% of mispredictions. Figure 10 shows that on average, 25% of branch mispredictions are eliminated through the use of dynamic if-conversion.

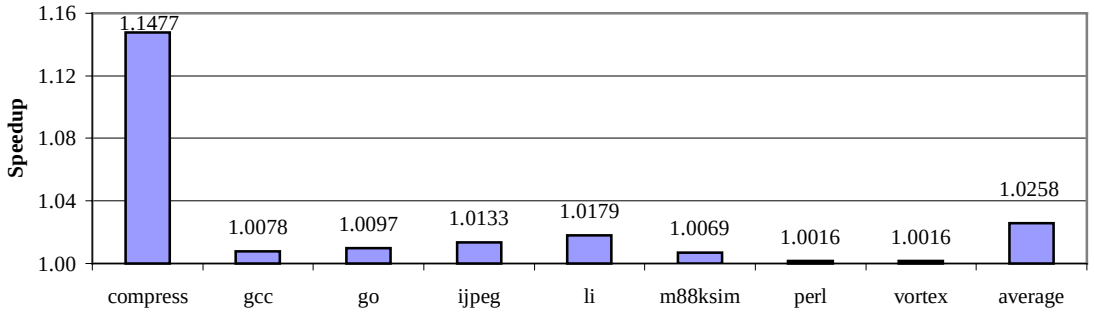

Figure 8. Speedup resulting from dynamic if-conversion.

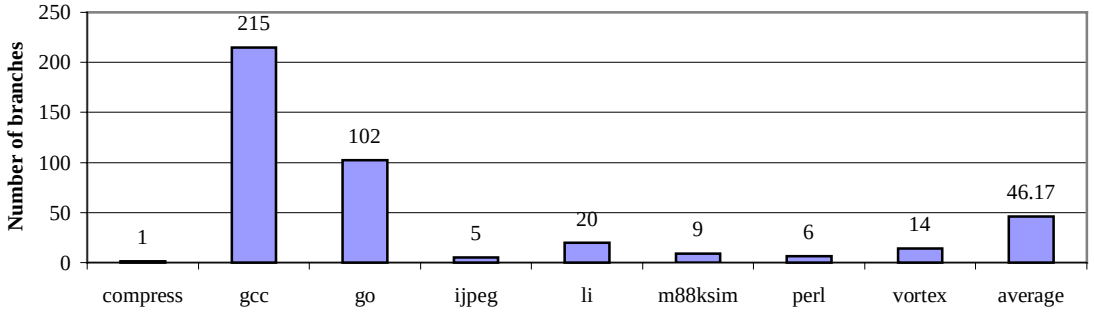

Figure 9. Number of branches dynamically converted to predicates for SPECInt95.

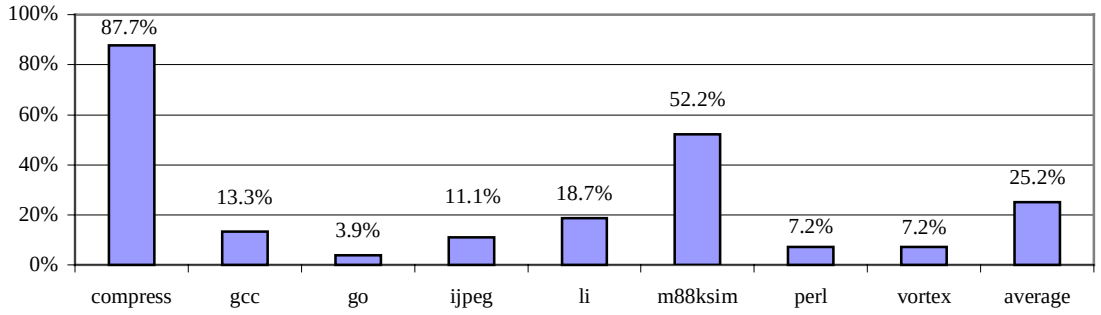

Figure 10. Percentage of branch misprediction eliminated by dynamic if-conversion.

4.2. Dynamic Reverse If-Conversion

While dynamic if-conversion is based on the notion that some branches were not converted to predicated instructions at compile time, the opposite scenario could also be true. The compiler may have converted certain basic blocks to sets of predicated instructions, yet those predicates may turn out to be quite biased for a given run. In this instance, it would be better if a branch guarded that basic block to allow those instructions to be skipped during execution.

Dynamic predicate-to-branch conversion, or dynamic reverse if-conversion, converts sequences of predicated instructions that are guarded by biased predicates back into branches at runtime. The idea here is that the predicates guarding the instructions evaluate to true so rarely that it would be better to branch over the particular instructions rather than to leave them as predicated instructions.

Equation 4 represents the dynamic reverse if-conversion criteria. It is very similar to the branch-to-predicate conversion algorithm in that it weighs the cost of each option – leaving the instructions as predicated instructions or converting them to branches.

$$\mathbf{P_{PRED'} * L_{PRED} \geq P_{MISS} * L_{MISS}} \qquad \mathbf{(4)}$$

$P_{PRED'}$ = odds of false predicate
L_{PRED} = number of predicated cycles
P_{MISS} = odds of mispredict
L_{MISS} = misprediction penalty

The left side of the equation calculates the penalty of leaving the instructions as a set of predicated instructions, while the right side attempts to predict the penalty of converting the predicates to a branch. The odds of misprediction are not known for a set of predicates because the branch predictor is not used for predicated instructions. Therefore, the average misprediction rate for the current branch predictor is used.

Figure 11 and Figure 12 show the speedup achieved by employing the dynamic reverse if-conversion algorithm for SPECInt95 along with the number of predicate traces converted to branches for each benchmark. From the graphs, we see an average speedup of 5% for converting an average of 27 predicate traces back into branches. The speedup in Figure 11 results from removing relatively long sequences of instructions that had been guarded by a typically-false predicate. Rather than enduring the performance penalty of continually executing instructions that are predicated on a false predicate, the predication was converted to control flow, and a branch was placed before the instructions. This branch allowed the control flow to essentially skip over the offending instructions, resulting in the speedup seen in Figure 11.

4.3. Possible Implementation

Because the algorithms presented in this paper are lightweight, they can easily be implemented in a modern dynamic optimization system with small overhead. Since many dynamic optimizers already perform some sort of hot-path detection, we can be assured that time will not be wasted performing the optimizations on rarely executed instructions.

While the algorithms do require some sort of profiling structure to be in place, they are not limited to a pure hardware or software solution. The requirement could be filled using schemes ranging from the built-in performance monitors planned for Itanium [12] to software structures such as the branch and trace counters in HP Labs' Dynamo [4].

Furthermore, incorporating the dynamic if-conversion algorithms can improve the performance of other algorithms already in place. For example, when combining dynamic if-conversion with aggressive dynamic rescheduling, the performance is better than additive. Dynamic if-conversion creates many more optimization opportunities for rescheduling because there is much more flexibility in scheduling predicated instructions than control-flow sequences.

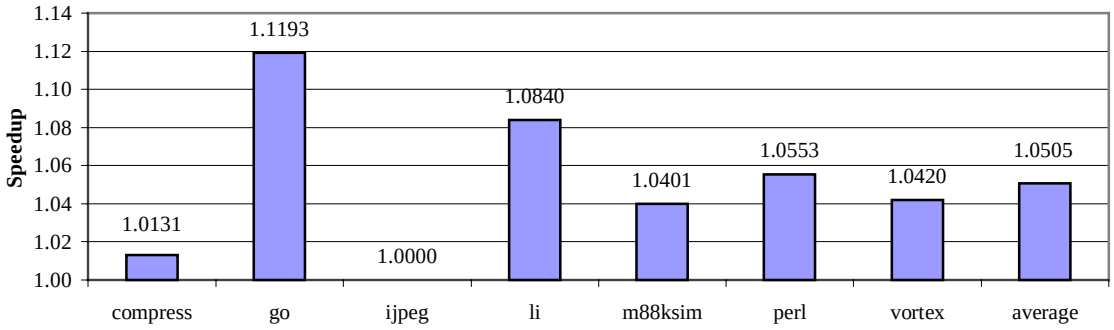

Figure 11. Speedup resulting from dynamic reverse if-conversion.

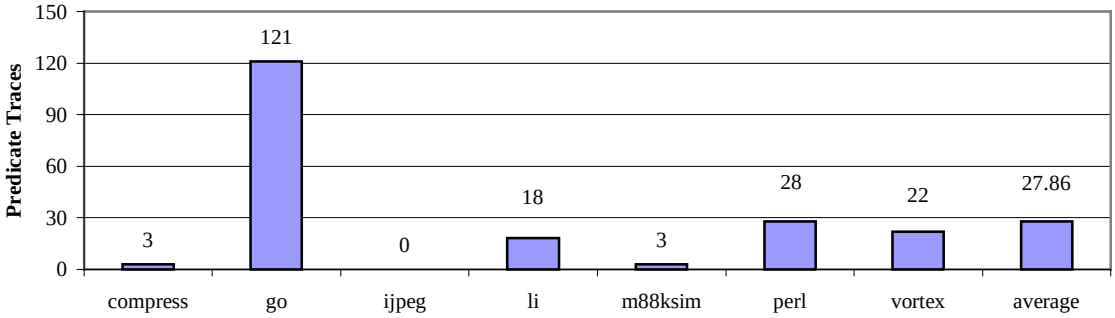

Figure 12. Actual number of predicates converted back to branches.

5. Conclusion

In this paper, an effective algorithm for incorporating dynamic if-conversion into a dynamic optimization system is introduced. We begin by determining an optimal means for tracking the misprediction rate of a branch and recognizing the need for dynamic if-conversion. As it turns out, if the warm-up period for a branch (defined as the time it takes for the branch misprediction rate to settle to a value within 1–10% of the previous value) is ignored, the error of our sample misprediction rates drops from 7% to 4% for a 25-event sample. For larger sample sizes, the error drops below 2%. The accuracy of the sample misprediction rates is important because the rates form the basis of our dynamic if-conversion algorithms.

Dynamic if-conversion was then introduced as a set of algorithms that take into account actual branch and predicate behavior to calculate the trade-offs of if-conversion at a particular runtime instance. The algorithms are lightweight – they can be implemented with minimal impact on system performance, and they are universal – they can be implemented in any dynamic optimization system available without the need for specialized hardware. By simulating the algorithms on an EPIC-style machine employing the latest branch prediction scheme, speedup values of up to 14.7% were observed.

This work opens the doors for many lightweight and heavyweight dynamic optimizations. This paper did not delve into dynamic rescheduling of dynamically if-converted instructions. However, performance can potentially be improved if aggressive dynamic rescheduling were used in conjunction with dynamic if-conversion. This is an interesting concept that should be explored further.

While this particular study focused on improving performance, a well-designed dynamic optimization system has great potential for venturing into other domains, such as power reduction or fault-tolerance. In the domain of power reduction, dynamic optimizations could be performed to lower the number of bit-flips between successive operations, thus reducing power use when necessary [26]. In the fault-tolerance domain, dynamic optimizations could insert recovery code for aggressive optimizations into the NOP slots of a schedule [23].

6. Acknowledgments

This research was supported through equipment and cash donations from Hewlett-Packard Company and an NSF CAREER award. We would like to thank the TINKER research group and the independent reviewers for feedback on earlier versions of this paper. We also wish to thank the developers of the many tools necessary for simulating an EPIC-style machine – Sergei Larin for Yula, a rebel-to-C

code generator; Matt Jennings for his instruction scheduler; Chao-ying Fu for his instruction tracer that served as the backbone of our simulations.

7. References

[1] Allen, A.O. *Probability, Statistics, and Queueing Theory with Computer Science Applications*. 2nd Edition. Harcourt Brace Jovanovich, Boston. 1990.

[2] Allen, J.R., K. Kennedy, C. Porterfield, and J. Warren. "Conversion of Control Dependence to Data Dependence." *Proc. of the 10th ACM Symposium on Principles of Programming Languages*, 1983, pp. 177-189.

[3] August, D., W. Hwu and S. Mahlke. "A Framework for Balancing Control Flow and Predication." *Proc. of the Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture*, 1997, pp. 92-103.

[4] Bala, Vasanth, E. Duesterwald, S. Banerjia. "Dynamo: A Transparent Dynamic Optimization System." *Proc. of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, 2000, pp. 1-12.

[5] Conte, T.M. and S.W. Sathaye. "Dynamic Rescheduling: A Technique for Object Code Compatibility in VLIW Architectures." *Proc. of the 28th Annual International Symposium on Microarchitecture*, 1995, pp. 208-218.

[6] Ebcioglu, K. and E. Altman. "DAISY: Dynamic Compilation for 100% Architectural Compatibility." *Proc. of the 24th Annual International Symposium on Computer Architecture*, 1996, pp. 26-37.

[7] Evers M., P. Chang and Y. Patt. "Using Hybrid Branch Predictors to Improve Branch Prediction Accuracy in the Presence of Context Switches." *Proc. of the 23rd Annual International Symposium on Computer Architecture*, 1996, pp. 3-11.

[8] Gonick, Larry and Woollcott Smith. *The Cartoon Guide to Statistics*. HarperPerennial. New York. 1993.

[9] Grant, Brian, M. Philipose, M. Mock, C. Chambers and S. Eggers. "DyC: An Expressive Annotation-Directed Dynamic Compiler for C." Technical Report UW-CSE-97-03-03. University of Washington. 1999.

[10] Havanki, W.A., S. Banerjia and T. M. Conte, "Treegion Scheduling for Wide-Issue Processors." *Proc. of the 4th International Symposium on High-Performance Computer Architecture.* 1998, pp. 266-276.

[11] How FX!32 Works. White Paper. Available at http://www.digital.com/amt/fx32/fx-white.html

[12] *IA-64 Application Developer's Architecture Guide*. Intel Corporation. May 1999.

[13] Kathail, V., Schlansker, M., Rau, B. "HPL PlayDoh Architecture Specification." Hewlett-Packard Laboratories Technical Report, February 1994.

[14] Klaiber, Alexander. "The Technology Behind Crusoe Processors: Low-Power x86-Compatible Processors Implemented with Code Morphing™ Software." Transmeta Corporation. January 2000.

[15] Klauser, A., T. Austin, D. Grunwald, B. Carter. "Dynamic Hammock Predication for Non-predicated Instruction Set Architectures." *Proc. of Parallel Architectures and Compilation Techniques*, 1998, pp. 278 - 285.

[16] The LEGO Compiler. Available for download at http://www.tinker.ncsu.edu/LEGO

[17] Leone, Mark and Dybvig, R. Kent. "Dynamo: A Staged Compiler Architecture for Dynamic Program Optimization." Technical Report #490, Department of Computer Science, Indiana University. 1997.

[18] Mahlke, S., R. Hank, R. Bringmann, J. Gyllenhaal, D. Gallagher, W. Hwu. "Characterizing the Impact of Predicated Execution on Branch Prediction." *Proc. of the 27th Annual International Symposium on Microarchitecture*, 1994, pp. 217 - 227.

[19] Mahlke, S, D. Lin, W. Chen, R. Hank, R. Bringmann. "Effective Compiler Support for Predicated Execution Using the Hyperblock." *Proc. of the 25th annual Inter-national Symposium on Microarchitecture*, 1992, pp. 45 - 54.

[20] Mahlke, Scott. *Exploiting Instruction-Level Parallelism in the Presence of Conditional Branches*. Ph.D. thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1996.

[21] Park, J.C.H. and M. Schlansker, "On Predicated Execution." Technical Report HPL-91-58, Hewlett-Packard Software Systems Laboratory, May 1991.

[22] Romer, T., G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, B. Chen, "Instrumentation and Optimization of Win32/Intel Executables Using Etch." *Proc. of the USENIX Windows NT Workshop*, 1997, pp. 1 – 7.

[23] Rotenberg, Eric. "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors." *Proc. of the 29th Fault-Tolerant Computing Symposium*, June 1999, pp. 84-91.

[24] Sathaye, Sumedh. *Evolutionary Compilation for Object Code Compatibility and Performance*. Ph.D. thesis, Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC, 1998.

[25] Smith, Michael D. "Overcoming the Challenges to Feedback-Directed Optimization." *Proc. of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, January 2000.

[26] Toburen, Mark. *Power Analysis and Instruction Scheduling for Reduced di/dt in the Execution Core of High-Performance Microprocessors*. Master's Thesis. Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC, August 1999.

[27] Yeh, Tse-Yu and Yale N. Patt. "Two-level Adaptive Training Branch Prediction." *Proc. of the 24th Annual International Symposium on Microarchitecture*, 1991, pp. 51 - 61.

[28] Zhang, Xiaolan, Z. Wang, N. Gloy, J. Chen and M. Smith. "System Support for Automatic Profiling and Optimization." *Proc. of the Sixteenth ACM Symposium on Operating Systems Principles*, 1997, pp. 15 - 26.