Appears in Proceedings of the 2011 International IEEE International Symposium on Workload Characterization (IISWC), pp. 172–183, Austin, Texas, November 2011

Program Interferometry

Zhe Wang and Daniel A. Jiménez
Department of Computer Science
University of Texas at San Antonio
{zhew,dj}@cs.utsa.edu

Abstract

Modern microprocessors have many microarchitectural features. Quantifying the performance impact of one feature such as dynamic branch prediction can be difficult. On one hand, a timing simulator can predict the difference in performance given two different implementations of the technique, but simulators can be quite inaccurate. On the other hand, real systems are very accurate representations of themselves, but often cannot be modified to study the impact of a new technique.

We demonstrate how to develop a performance model for branch prediction using real systems. The technique perturbs benchmark executables to yield a wide variety of performance points without changing program semantics or other important execution characteristics such as the number of retired instructions. By observing the behavior of the benchmarks over a range of branch prediction accuracies, we can estimate the impact of a new branch predictor by simulating only the predictor and not the rest of the microarchitecture. We call this technique Program Interferometry *based on its similarity to astronomical optical interferometry.*

Using measurements of the Intel Xeon E5440 Processor, we quantify the impact of branch prediction on a set of benchmarks, developing regression models that estimate the performance given by changes in the branch predictor. We incorporate these models into a simulator allowing us to estimate the impact of several branch predictors.

This first study in program interferometry points the way to future work on estimating the impact of other microarchitectural structures. We demonstrate the potential for interferometry to estimate the impact of L1 and L2 caches by perturbing data layouts.

1 Introduction

Astronomers used the earliest telescopes to view the universe from a single point of view. Their observations were dim and blurry, limited by the tiny amount of light that their small telescopes could collect and the effects of atmospheric turbulence. However, in recent years, astronomers have used a technique called optical interferometry to combine the observations of many telescopes from many different points of view to obtain images with a much higher resolution [1].

Similarly, by sampling and observing many points in a space of program performance, we can get a much better understanding of program behavior. This paper presents a technique called *Program Interferometry*, based on perturbing placement of code and data. Many executable versions of a program are produced by pseudo-randomly re-ordering procedures and objects files. Similarly, the memory allocator places objects pseudo-randomly on the heap. A given random placement of code and data can be repeated by using the same key for the pseudo-random number generator so that runs are reproducible. Each code and data placement is semantically equivalent, but because the instruction addresses are different, different conflicts will arise among microarchitectural structures such as the branch predictor and instruction cache [23]. The situation is isomorphic to one in which we keep the code and data placement constant, but change the hash functions for microarchitectural structures. Thus, we may measure the performance impact of changing these structures.

1.1 Performance Variance

Figure 1 shows the percent difference from average performance as measured by cycles-per-instruction (CPI) caused by 100 random but plausible code reorderings for the SPEC CPU 2006 benchmarks. The graph is a violin plot, showing the probability density at each CPI value, i.e., the thickness at each CPI value is proportional to the number of CPIs observed in that neighborhood. Mytkowicz *et al.* use violin plots to strikingly illustrate the impact of performance variance on experimental methodology [23]. Clearly, some benchmarks are greatly affected by differences in instruction addresses while some are less sensitive.

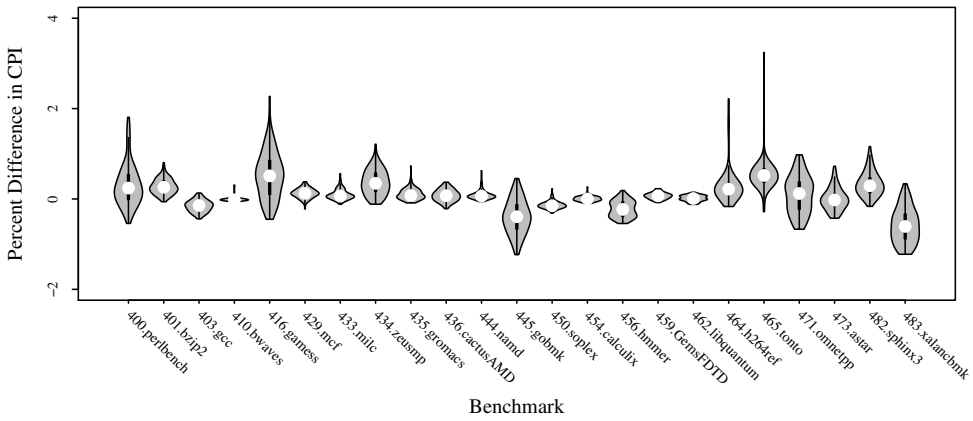

Figure 1. Violin plots for SPEC CPU 2006 percentage performance variation with code reordering.

1.2 Varying Branch Prediction Accuracy

Figure 2 demonstrates the potential of program interferometry. Each of the 100 points represents an executable with a different code reordering of the SPEC CPU 2006 benchmarks `400.perlbench` and `471.omnetpp` running on `ref` inputs. Performance monitoring counters enable collecting the cycles-per-instruction (CPI) and branch mispredictions per 1000 instructions (MPKI) of each run. The plot shows actual measurements as well as a least-squares regression line estimating the linear relationship between MPKI and CPI. They also show 95% confidence intervals and 95% prediction intervals.

1.3 Varying Cache Misses

Since instructions account for relatively few cache misses, we augment code reordering with data reordering in memory. The data reordering is done using a specially crafted memory allocator that randomizes the placement of heap-allocated data. Heap randomization has been proposed for memory safety [2], but here we use it to elicit performance variance. Figure 3 shows that performance varies linearly with L1 and L2 cache misses for the SPEC CPU 2006 benchmark `454.calculix`. The figure also shows confidence and prediction intervals for a performance model based on cache effects. The experiments were done using heap randomization combined with code reordering.

1.4 Focus on Branch Prediction

In this paper, we focus on measuring the impact of the branch predictor on performance. Future work will focus on the other microarchitectural structures affected by code and data placement such as the instruction and data caches, instruction decoders, L2 cache, etc.

As an example of the usefulness of program interferometry to branch predictor design, linear regression allows us to make the following predictions for `400.perlbench` with 95% probability: 1) A perfect branch predictor would yield a CPI of 0.517 ± 0.029, an improvement of $26.0\% \pm 4.2\%$. 2) Halving the average MPKI from 6.50 to 3.25 would improve CPI by $13.0\% \pm 2.2\%$ from 0.70 to 0.61 ± 0.022. 3) A 10% improvement in CPI due to branch prediction improvement would require a 38% reduction in mispredictions.

1.5 Using Current Microarchitectures to Improve Future Microarchitectures

This paper presents a technique that can estimate the performance improvement of changing the branch predictor in a current microarchitecture. This is a good way to explore new microarchitectural ideas in the absence of clear information about what future microarchitectures will look like. An alternative would be to use cycle-accurate simulators with best-guess estimates of future microarchitectural structures. However, it is not clear to researchers what future microarchitectures will be like. The return of Intel from the more complex Netburst to the simpler P6-inspired Intel Core 2 is an example of this uncertainty. The trend in 2001 was toward deeper and deeper pipelines, so contemporaneous branch prediction papers simulating pipeline depths of up to 40 were way off the mark. Also, simulators are notoriously inaccurate with respect to real systems because many of the details of real systems are difficult or impossible to model or even to know about [5]. Earnest efforts at simulation are subject to bugs that can invalidate research conclusions made with them [6]. Thus, demonstrating that a new branch predictor (or other optimization) can improve an existing microarchitecture is another way to have confidence in that optimization's contribution to unknown future microarchitectures.

2 Related Work

In this section we discuss related work.

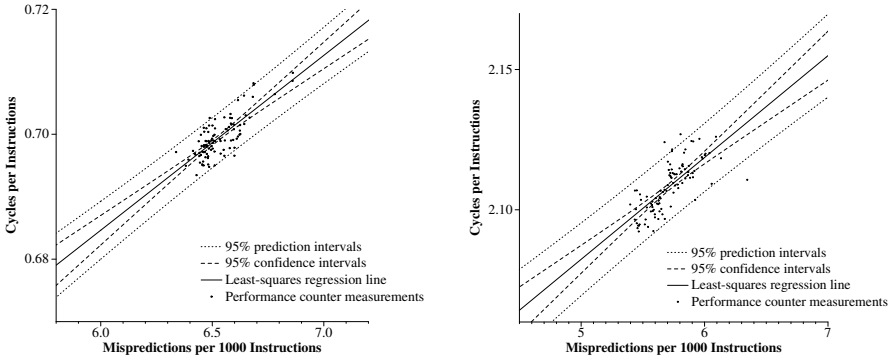

Figure 2. Performance changes with branch prediction accuracy for `400.perlbench` and `471.omnetpp`.

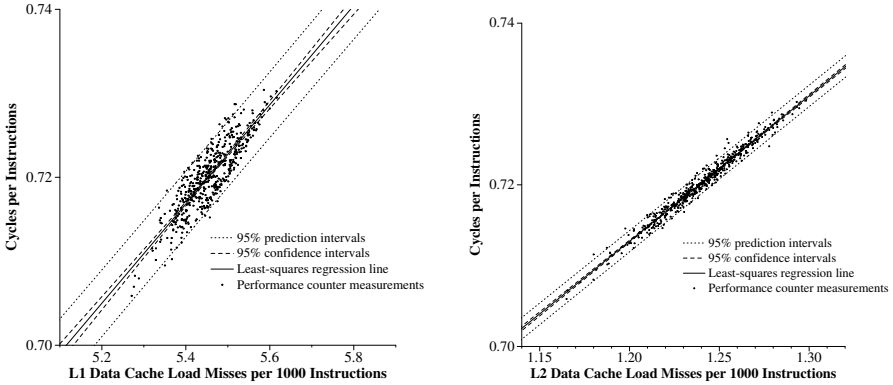

Figure 3. Using heap randomization along with code reordering to model cache effect on performance. Performance changes with (a) L1 and (b) L2 cache misses for `454.calculix`.

2.1 Eliciting Performance Variance

Mytkowicz *et al.* introduce the technique of object file reordering for showing that different link orders of object files, as well as other seemingly random and harmless details of an experimental setup, can yield significantly different performance [23]. That work indicts the architecture and programming languages community for falling victim to measurement bias, i.e., allowing oneself to believe that some observed improvement in program behavior is due to one's own technique rather than a happy coincidence of experimental factors. Our work was partly inspired by Mytkowicz *et al.*. We choose to see the phenomenon they exposed as an interesting opportunity to develop a tool to examine microarchitectural behavior.

Rubin *et al.* propose a framework to explore the space of data layouts using profile feedback to find layouts that yield good performance [26]. They point out that the general problem of optimal data layout is NP-hard and poorly approximable. The space of data layouts is similar to the space of code reorderings, and the impact of data layouts on the data cache is similar to the impact of code placement on the branch predictor and instruction cache.

2.2 Impact of Code Placement on Performance

The impact of code placement on performance has not gone unnoticed in the academic literature. Many code-improving transformations have been proposed based on code placement. Hatfield and Gerald [7], Ferrari [9], McFarling [19], Pettis and Hanson [24], and Gloy and Smith [10] present techniques to rearrange procedures to improve locality using profiling. Mytkowicz *et al.* exploit the kind of performance variance described in this paper to optimize programs [16]. Calder and Grunwald present *branch alignment*, an algorithm that seeks to minimize the number of taken branches by reordering code such that the hot path through a procedure is laid out in a straight line [3]. Young *et al.* present a near-optimal version of branch alignment [31]. Jiménez proposes a technique to use code placement to explicitly avoid branch mispredictions due to conflicts in the predictor tables [13]. Knights *et al.* propose exploiting fortuitous object code orderings to improve performance [16].

From the microarchitecture side, a trace cache is a specialized instruction cache that exploits instruction locality

by organizing instructions in the order they are executed, rather than in their static program order[25]. With a trace cache, branch prediction and instruction fetch can be made somewhat immune to the effect of code placement when there is a high hit rate in the trace cache. The Intel Netburst microarchitecture in the Pentium 4 processor line featured a micro-op trace cache [12].

Our technique is not an optimization, but a tool for peering inside the microarchitecture using code placement. If thoughtful code placement optimizations like those mentioned above were widely adopted, our results would show less variance in execution behavior and less confidence in the regression lines. Nevertheless, most production code is not optimized with code placement in mind; thus, our results are widely applicable to real systems.

2.3 Estimating Performance

We use linear regression to estimate processor performance. Other work has also used regression to estimate program behavior for simulators as well as real systems.

2.3.1 Estimating Simulation Results with Regression

Lee and Brooks propose using regression modeling to estimate processor performance and power under a given microarchitectural configuration after sampling a small portion of the microarchitectural design space through simulation. Performance and power are accurately predicted with an error of about 4% on average. Joseph *et al.* propose non-linear [15] regression techniques such as neural networks for estimating CPI given a set of microarchitectural parameters. The technique predicts CPI with an error of 2.8% on average. Both of these proposal are intended to reduce the number of points in a processor design space that must be simulated to find parameters that give good performance.

Our technique differs in that we are modeling the behavior of a real system rather than a simulation design space. Simulators can be inaccurate with respect to real systems [5, 6]. On the other hand, real hardware is a perfectly valid model of itself. Through careful measurement, the performance impact of changing a single microarchitectural feature such as branch prediction can be estimated accurately using the hardware itself to model to rest of the microarchitecture.

2.3.2 Estimating Behavior of Real Systems

Contreras and Martonosi use performance monitoring counters to develop a linear power model of the Intel XScale processor [4]. This approach can enable a technique capable of quickly estimating future power behavior and adapting to it at run-time. Our technique is similar in that it uses performance monitoring counters to develop a model of program behavior. However, we focus on modeling the behavior of one program at a time to get very precise information about the change in performance in response to a small change in the behavior of microarchitectural structures, i.e., our work concentrates on a much finer level of granularity, and we focus on performance instead of power.

3 Performance is Strongly Linear in Mispredictions

Before attempting to apply linear regression to estimating performance from performance-related events, we must demonstrate that there is a linear relationship. This section explores the relationship between branch mispredictions and CPI through cycle-accuracy simulation.

3.1 Potential Causes of Non-Linearity

Branch mispredictions and other performance-related events are not necessarily independent from one another. For example, some branch mispredictions might cause prefetching into the cache, and others might cause cache pollution. It is possible that, as branch prediction accuracy changes, the nature of the prefetching or pollution changes as well, causing the performance impact of branch mispredictions to change non-linearly.

3.2 Demonstrating Linearity

Program interferometry can elicit only a small range of branch prediction accuracies. We use simulation to extend the range for the purpose of demonstrating linearity. We use the MASE simulation infrastructure [17] to explore the relationship between MPKI and CPI. MASE is a cycle-accurate simulator for the Alpha AXP instruction set that includes several branch prediction models, including perfect branch prediction. We configure MASE with parameters as similar as possible to those of Intel Xeon. We choose all of the benchmarks from SPEC CPU 2000 and SPEC CPU 2006 that will compile and execute under MASE without errors. MASE simulates 145 different branch predictor configurations with varying accuracies, as well as a perfect branch predictor. Each simulation runs for one billion instructions from the single simpoint that best characterizes its typical behavior [28]. We use linear regression with the results from non-perfect branch predictors to estimate the performance for the perfect predictor, and compute the percent error to the performance given by perfect branch prediction.

Figure 4 shows the percent error between estimated and actual perfect prediction for each benchmark, ordered from lowest to highest error. The average percent difference was 1.32%. The two worst benchmarks, `252.eon` and `178.galgel`, show some non-linear behavior and percent differences of 6.0% and 7.5%, respectively. All other benchmarks have percent errors below 3.0%, and 3/5 of them have percent errors below 1%.

The figure also shows the percent error when estimating the performance of the most accurate predictor from the academic literature: L-TAGE [27]. For most benchmarks,

L-TAGE is highly accurate but often far from perfect. Thus, its performance will be closer to that of the other predictors than the performance of the perfect predictor. In this more realistic case, the percent error is very low. The average error is less than 0.3%, and the highest error is less than 1%. Thus, for estimating the performance of even the most aggressive branch predictors, linear regression is highly accurate.

Figure 5 illustrates the strong linear relationship between MPKI and CPI under simulation. The two graphs show the simulated MPKI and CPI pairs with regression lines, with CPI normalized to the performance with perfect branch prediction. The point (0,1) represents the MPKI and CPI for perfect prediction. Figure 5(a) shows the results for `473.astar`, `401.bzip2`, and `458.sjeng`. It is apparent from the graph that there is a linear relationship between MPKI and CPI. Even for `458.sjeng`, the benchmark with the fifth highest error, the percent error of 2.7% at (0,1) is barely perceptible. Figure 5(b) shows the results for `456.hmmer`, `252.eon`, and `178.galgel`. These are the three benchmarks with the worst percent errors. The error is perceptible from the graph, but is still very small. The error for perfect prediction is the worst case for linear regression; in practice, as in the case of estimating the performance of L-TAGE, we expect the error to be well below 1%.

We conclude that in the great majority of cases, the relationship between MPKI and CPI is strongly linear, and that it is appropriate to estimate this relationship for a real machine using linear regression. With this knowledge we can rely on statistical tools to quantify the impact of noise and reduced range from real systems.

4 Program Interferometry

In this section we describe the technique of program interferometry. The basic idea is to execute code under many different reorderings, causing a wide variance in performance due to different accidental collisions in microarchitectural structures. By measuring the resulting adverse microarchitectural events, we can build a performance model for the program and microarchitecture.

4.1 Instruction Addresses in Microarchitectural Structures

Program interferometry exploits the fact that several microarchitectural structures use a hash of instruction and data addresses. For example:

1. A 128-set instruction cache with 64 byte blocks would likely use bits 6 through 12 of the instruction address as the set index.

2. A branch direction predictor might index a table of counters using a combination of branch history and branch address bits.

3. A branch target buffer (BTB) or indirect branch predictor would use lower-order bits of the branch address to index a table of branch targets.

Sometimes addresses will accidentally collide in some microarchitectural structure. For example, conflict misses in the instruction cache occur when the number of blocks mapping to a particular set exceeds the associativity of the cache. Although this phenomenon has been studied in academic research, most compilers do not optimize to protect against these kinds of conflicts.

Compiler writers are aware of uses of instruction addresses and write compilers to exploit these uses. For instance, a common heuristic is to align the target of a branch on a boundary divisible by the number of bytes in a fetch block to allow the fetch beginning at that target to read the maximum number of instruction bytes in one cycle.

4.2 A Wide Range in Performance

These accidental conflicts result in adverse microarchitectural events such as branch mispredictions, cache misses, BTB misses, etc. A particular code and data placement will result in a particular number of accidental collisions with a particular impact on performance. A different layout will result in a difference impact on performance. By exploring a wide range of layouts, we can force a wide range of adverse performance events to take place and explore a wide range of performances.

4.3 Astronomical Analogy

This work draws inspiration from astronomical optical interferometry, where multiple telescopes at different locations collect an image of an astronomical object. These images are combined into a much more complete picture than any one of the telescopes by itself could produce. The collection of telescopes acts as one large and more powerful telescope. In our work, each combined executable is like a single telescope focused on an astronomical object. Each executable has its own set of accidental collisions in microarchitectural structures, which may be compared with atmospheric turbulence present above that telescope. By sampling many different semantically equivalent executables, each with its own accidental collisions, we get a much better picture of the behavior of the program in terms of the impact of instruction-address-sensitive microarchitectural structures on performance.

4.4 Causing Collisions

To generate many random but plausible code layouts, we extend the technique of Mytkowicz *et al.*, i.e., object-file reordering. We compile each benchmark once, lowering it to assembly language files. Then we produce executables with hundreds of different code reorderings. We then reorder procedures within assembly files, assemble the files,

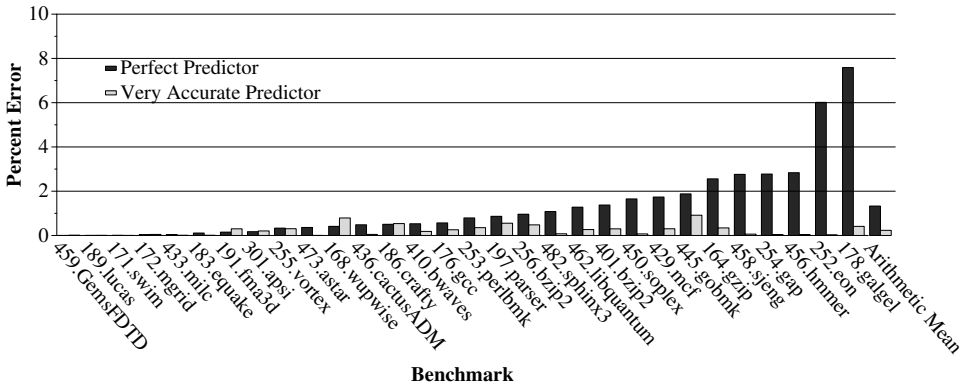

Figure 4. Percent error estimating perfect and L-TAGE branch prediction CPI with linear regression from imperfect predictors.

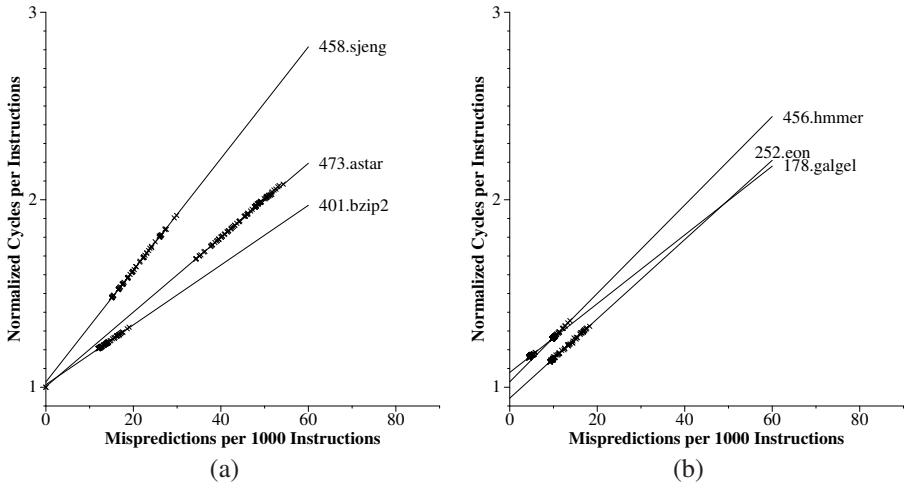

Figure 5. Linear regression lines relating mispredictions to performance for (a) highly linear benchmarks and (b) less linear benchmarks.

and then link with different randomly-generated order of the object files. The linker lays code out in the order in which it is encountered on the command line, so each random procedure and object-file ordering results in a different code layout. In the experiments illustrated in 1.3, we use a custom memory allocator based on DieHard [2] that essentially assigns random addresses to heap-allocated object to elicit perturbations to due conflict misses in the data caches. We execute each resulting executable five times, collecting performance monitoring counter information such as number of instructions committed, number of branch mispredictions, number of clock cycles, etc. We take the performance monitoring counter statistics that gave the median performance. Details of our infrastructure are given in Section 5

4.5 Making Predictions

Once the performance monitoring counter information has been collected, we can begin using statistical tools to build a performance model. We use least-squares linear regression to estimate the relationship between various microarchitectural events and performance outcomes. For instance, for the plots in the Introduction, we found a regression line of $\text{CPI} = 0.02799 * \text{MPKI} + 0.51667$. That is, we use the MPKI to predict the CPI. For a range of MPKI values, we also 95% computed confidence intervals and prediction intervals. A 95% confidence interval has a 95% chance of containing the true regression line, i.e., of all the data collected, the line that best illustrates the linear relationship between CPI and MPKI has a 95% chance of being in that confidence interval [20]. The larger 95% prediction interval has a 95% chance of containing all of the observations (i.e. CPIs) that would be encountered in a given domain (i.e. set

of MPKIs).

Linear regression only works if we may confidently assume that there is a linear relationship. Under normal circumstances CPI and MPKI do indeed have a linear relationship: for each benchmark, there is an average misprediction penalty, and each extra misprediction increases the number of cycles by this penalty. In Section 3 we verify this assumption.

4.6 When Things Go Wrong

Some benchmarks do not give a wide range in performance under code reordering, or the range in performance cannot be explained by events related to the instruction address. For each type of prediction we would like to make for a given benchmark, we first determine whether there is significant correlation between the dependent variable and independent variables. We use Student's t-test with the null hypothesis "there is no correlation," i.e., if we cannot reject the null hypothesis, then we cannot say whether there is any correlation between the events observed [20]. For the 23 SPEC CPU 2006 benchmarks that compiled in our infrastructure, estimating CPI with MPKI, the null hypothesis was rejected at $p = 0.05$ or less for 20 benchmarks. In other words, for the great majority of the benchmarks, we determined that there was at least a 95% chance that program interferometry found significant correlation between CPI and MPKI. For the other benchmarks, there was not enough range of MPKI to predict CPI.

5 Experimental Methodology

This section describes the experimental methodology used for this paper.

5.1 Compiler

We use the Camino compiler infrastructure [11]. This system is a post-processor for the Gnu Compiler Collection (GCC) version 4.2.4. C, C++, and FORTRAN programs are compiled into assembly language, the assembly language is instrumented by Camino, and the result is assembled and linked into an executable. Camino features a number of profiling passes and optimizations, but for this study we implement and use only the profiling and instrumentation pass described below. All of the executables produced for this study target the x86_64 instruction set.

5.2 Benchmarks

We use the SPEC CPU 2006 benchmarks for this study. Of the 29 benchmarks, 23 compile and run without errors with our compiler infrastructure. These benchmarks are listed in the x-axes of several graphs in later sections.

5.3 Generating Random Code Reorderings

Each benchmark is compiled once from C/C++/FORTRAN into assembly. The Camino infrastructure is then used to reorder procedures within files and then assemble the files into object code files. The resulting object files are randomly reordered and linked to make an executable. Camino accepts a seed to a pseudo-random number generator to generates pseudo-random but reproducible orderings of procedures and object files.

5.4 System

We perform our study using four Dell systems with identical configurations running the 64-bit version of Ubuntu Linux 8.04 Server and a custom compiled kernel with performance monitoring counter support. Each system contains two quad-core Intel Xeon E5440 processors. The Intel processor 5400 Series are based on 45nm Enhanced Intel Core Microarchitecture. Each processor has 16GB of SDRAM and 12MB second level cache. Each core in the Intel Xeon E5440 processor has 32KB instruction cache and a 32KB data cache. The branch predictor of the Intel Xeon E5440 is not documented, but through reverse-engineering experiments we have determined that it is likely to contain a hybrid of a GAs-style branch predictor and a bimodal branch predictor [30, 29, 8].

5.5 Running with Performance Monitoring Counters

We measure a number of performance monitoring counters using the `perfex` command found in the PAPI performance monitoring package [22]. The Intel Xeon processor allows up to two user-defined microarchitectural events to be counted simultaneously. We are interested in more than two events, so we make multiple runs of each benchmark to collect all of the desired counters. We group the counters into three sets of two. For each set we run each benchmark five times and take the measurements given by the run with the median number of cycles. Only the microarchitectural events that occur while user code is running are counted, thus the impact of system events is minimized. We collect the following statistics: 1) Retired branches mispredicted, 2) Retired x86 instructions excluding exceptions and interrupts, 3) L1 instruction cache misses, 4) L2 cache misses, and 5) Elapsed clock cycles.

From these counters, we can derive other statistics such as cycles-per-instruction (CPI), branch mispredictions per 1000 instructions (MPKI), various cache miss rates, etc.

Although each system is configured identically and each core has the same microarchitecture, we use the Linux `taskset` command to make sure that each benchmark always runs on the same core to eliminate the effect of possible slight differences among the cores. Each run is performed on an otherwise quiescent system with as many sys-

tem services stopped as possible without compromising the ability to access remote files and log in remotely. Stack address randomization, a security feature that resists stack-smashing attacks, is disabled to minimize performance variance not due to code placement.

5.6 Simulation

We develop several branch predictor simulators. We implement these as a tool in Pin [18]. We then run pin on the same executables that we run natively. Our Pin tool instruments each branch with a callback to code that simulates a set of branch predictors. The tool counts the number of branches executed and the number of branches mispredicted for each predictor simulated.

5.7 Timing Concerns

Many of the SPEC CPU 2006 benchmarks run for over 30 minutes on the first `ref` input. For this study, we have executed each of the 23 benchmarks at least 100 times on a set of 4 computers. To facilitate this study, we instrument the benchmarks such that under native execution they run for up to approximately two minutes each. To do this, we implement a two-pass profiling and instrumentation pass in the Camino compiler. The first pass inserts instrumentation that collects information about each procedure. The benchmark is allowed to run for two minutes. Then the collected information is analyzed to find a procedure with a low dynamic count that is also executed near the end of the two-minute run. The second pass of the compiler instruments only that procedure such that when it is executed the same number of times as before, the program is ended. The first instrumentation has low overhead, thus the resulting executable runs for approximately two minutes. The second instrumentation affects a low-frequency procedure and takes two x86 instructions, thus it has negligible overhead. All of the executables in this study are compiled from this second instrumentation, or are from benchmarks that naturally run for less than two minutes. Because we are counting procedures and not elapsed time, each run of a benchmark executes the same number of user instructions.

5.8 Statistics

We make use of some statistical techniques in this study. We briefly review these techniques:

1. Correlation coefficients. Also known as Pearson's r, correlation coefficients range from -1.0 to 1.0 and measure the correlation between two random variables. A higher magnitude for r means a higher degree of correlation. A negative value of r means that two variables are negatively correlated. For instance, we find that the MPKI and CPI of `473.astar` have a sample correlation coefficient of 0.80 for 100 random orderings. Thus, MPKI and CPI have a rather strong correlation for this benchmark, but other factors also affect performance.

2. Coefficient of determination. The sample coefficient of determination, computed as r^2 where r is the correlation coefficient, gives the fraction of dependence of a given observation on an underlying factor. For instance, the coefficient of determination between MPKI and CPI for *473.astar* is 0.65. Thus, 65% of the variability in CPI observed in `astar` can be attributed to branch mispredictions.

3. Linear regression. We develop performance estimators using linear least-squares regression that finds a best-fit equation of a line between two variables, e.g. between MPKI and CPI. We also use multi-linear least-squares regression to produce an estimator for performance in terms of several observed variables.

4. Hypothesis testing. We use Student's t-test for hypothesis testing. We formulate a null hypothesis, e.g. "there is no correlation between CPI and MPKI" then use hypothesis testing to see if the null hypothesis can be rejected. We consider a result significant if the null hypothesis can be rejected with $p = 0.05$, i.e., the probability that the null hypothesis is not true is 95%. Student's t-test gives a meaningful result in the presence of normally distributed data. The observed CPI of most of the benchmarks roughly follow a normal distribution, thus in most cases hypothesis testing can give us additional confidence in our results.

5. Confidence intervals and prediction intervals. For the linear regression lines, we plot 95% confidence intervals and 95% prediction intervals. A 95% confidence interval has a 95% chance of containing the true regression line, i.e., of all the data collected, the line that best illustrates the linear relationship between CPI and MPKI has a 95% chance of being in that confidence interval. The larger 95% prediction interval has a 95% chance of containing all of the observations (i.e. CPIs) that would be encountered in a given domain (i.e. set of MPKIs).

6 Estimating Performance by Counting Microarchitectural Events

This section shows the potential of program interferometry to predict performance. We develop and evaluate regression models for a number of benchmarks using several characteristics of program behavior such as branch prediction and cache misses.

6.1 Assigning Blame

Code reordering can elicit a wide range of CPIs for our benchmarks. Here, we determine how much blame to place

on certain microarchitectural structures for the performance variance. We focus on what we believe to be the microarchitectural events most likely to be affected by code placement: 1) Branch mispredictions. Conditional branch predictors use the address of an instruction to index one or more tables. Branches may conflict with one another in these tables leading to *aliasing* [21] causing branch prediction accuracy to suffer. 2) L1 instruction cache misses. The Intel Xeon Core has a 32KB 8-way set associative instruction cache. If nine or more frequently used blocks map to the same set, there will be frequent cache misses. 3) L2 cache misses.

We also use multi-linear regression to develop a combined model that takes into account all three of these events in the hope that a combined model will be more accurate than using one of the observations by itself.

Using r^2, the coefficient of determination, we can determine what portion of performance is due to a particular microarchitectural event. Figure 6 shows the cumulative r^2 for each of the three events, as well as r^2 for the combined regression model. On average, 27% of the CPI difference between different code reorderings can be explained by branch misprediction.Some benchmarks are more sensitive; for instance, 84.2% of the CPI variance of `462.libquantum` is due to branch mispredictions.

The average bar for the combined model does not reach exactly the same height as that of the sum of the three measurements. This is because the three measurements are not altogether independent of one another; for instance, in some cases, a branch misprediction might cause an L1 cache event, sometimes causing cache pollution and other times causing prefetching. It must be emphasized that the correlation we report between microarchitectural events and performance is with respect to code ordering. Other changes to the execution environment would show other correlations.

6.2 Establishing Statistical Significance

Clearly many benchmarks' performance show correlation with microarchitectural events. However, we must ask whether the correlation is statistically significant. We use Student's t-test to determine statistical significance. For each of the three measurements as well as the combined model we attempt to reject the null hypothesis that there is no correlation. The value $p \leq 0.05$ for the t-test is traditionally accepted as proof of statistical significance. For the combined model we use the F-test $p \leq 0.05$ instead of the t-test, as the t-test is appropriate for single-variable linear regression models.

6.3 Number of Samples

For some benchmarks, the effect of code reordering on performance is harder to detect than for others. To establish statistical significance for as many benchmarks as possible, we sample a number of code reorderings in multiples of 100 until the benchmark is able to reject the null hypothesis, or until by inspection we determine that the benchmark is unlikely to reject the null hypothesis with a much larger number of samples. Most benchmarks reject the null hypothesis within the first 100 samples. Some take 200 samples, and a few require 300 samples. We do not discard any data when building or testing our regression models: we use the data from each reordering.

6.4 Blame the Branch Predictor

Of the 23 benchmarks, 20 show significant correlation between CPI and branch prediction. No other measurement consistently shows statistically significant correlation with CPI. The combined estimator does not increase the number of benchmarks showing significant correlation, and indeed two benchmarks that show significant linear correlation with MPKI through the t-test fail to reject the null hypothesis for the F-test with the combined model and multiple linear regression. Thus, in this paper we focus our attention on branch prediction.

6.5 Other Measurements

The code reordering methodology for program interferometry clearly elicits a large impact on branch prediction. As demonstrated in 1.3, a randomizing memory allocator can manipulate program behavior to elicit variance in the memory system. In future work we will study the impact of other events dependent on code and data placement. For instance, the number and type of x86 instructions in a fetch block has a large impact on the efficiency of decoding, but at this point it is not clear how to measure that impact.

6.6 A Linear Performance Model

We use least-squares linear regression to derive branch prediction performance models for the Average Model and each of the benchmarks that passed the hypothesis testing phase. For each benchmark, we find the best fit of the observed data to a regression line $y = mx + b$ where y is CPI and x is MPKI. The slope (m) gives the cost for performance of one additional MPKI and the y-intercept (b) gives the predicted average CPI for perfect branch prediction, i.e. 0 MPKI.

We also derive 95% confidence intervals and 95% prediction intervals for the regression lines. Figure 2 in the Introduction shows the regression line and intervals for `400.perlbench` and `471.omnetpp`. The confidence interval has a 95% chance of containing the true regression line for the data observed. The much wider prediction interval has a 95% chance of containing future observations. Thus, we can be 95% sure that the CPI of `471.omnetpp` with perfect branch prediction would be between 1.86 and 1.94. Table 1 shows the slopes and y-intercepts found by linear regression for each benchmark. It also shows the high and low prediction intervals for perfect prediction.

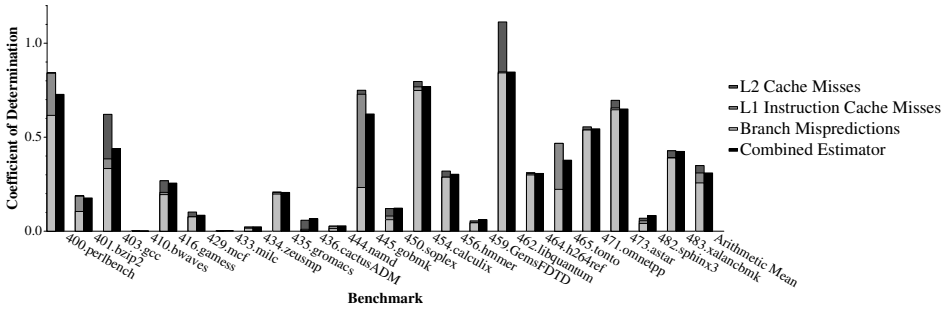

Figure 6. Coefficient of determination showing how much of each type of event accounts for overall performance.

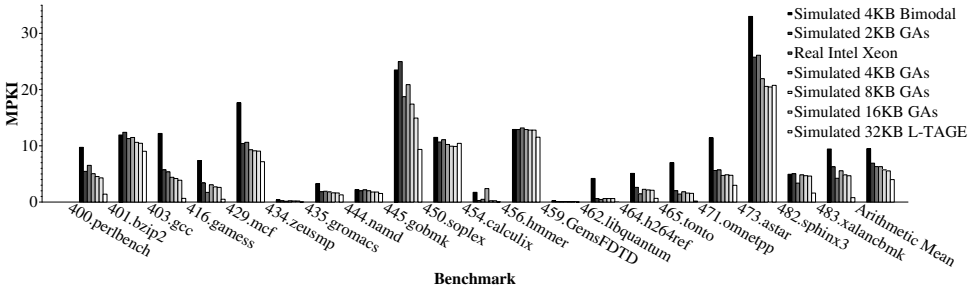

Figure 7. MPKI of real and simulated branch predictors.

Benchmark	Slope	y-intercept	Low	High
400.perlbench	0.028	0.517	0.488	0.546
401.bzip2	0.017	0.596	0.485	0.708
403.gcc	0.028	1.839	1.796	1.882
416.gamess	0.041	0.548	0.519	0.577
429.mcf	0.019	4.675	4.531	4.819
434.zeusmp	0.373	0.863	0.813	0.913
435.gromacs	0.020	0.811	0.795	0.827
444.namd	0.033	0.620	0.551	0.689
445.gobmk	0.019	0.643	0.515	0.771
450.soplex	0.016	1.822	1.741	1.904
454.calculix	0.023	0.461	0.460	0.463
456.hmmer	0.041	0.203	0.032	0.375
459.GemsFDTD	0.516	1.229	1.189	1.269
462.libquantum	0.022	1.432	1.431	1.433
464.h264ref	0.032	0.466	0.451	0.481
465.tonto	0.027	0.632	0.617	0.647
471.omnetpp	0.036	1.901	1.860	1.941
473.astar	0.022	2.373	2.289	2.456
482.sphinx3	0.036	0.916	0.798	1.034
483.xalancbmk	0.029	1.914	1.881	1.947

Table 1. Least-squares regression model relating branch prediction to performance. Shows high and low prediction intervals for perfect prediction i.e. 0 MPKI.

7 Estimating Branch Prediction Performance

This section presents results of simulation experiments using program interferometry to predict the performance impact of changes to the branch predictor. We use the performance model derived with program interferometry to predict the performance given by several predictors.

7.1 Branch Prediction Simulation

The Pin tool instruments each branch with a callback to code that simulates a set of branch predictors. The tool counts the number of branches executed and the number of branches mispredicted for each predictor simulated.

7.2 Impact of Mispredictions on Performance

We explore only those benchmarks that were demonstrated in the previous section to be suitable for program interferometry. Figure 7 shows the average MPKI for various branch predictors simulated with Pin as well as the average MPKI from the real Intel Xeon branch predictor. These data are averaged over 100 different pseudo-randomly generated code reorderings. For each benchmark, these are the same first 100 reorderings used for the performance monitoring counter measurements. Pin runs only once for each reordering; since we control the initial conditions of the simulator and Pin is not affected by system-level events, there is no

variance in the simulation result. We simulate GAs branch predictors [30] ranging in size from 2KB to 16KB to explore the effect of decreasing or increasing the hardware budget for the branch predictor. The average MPKI over all benchmarks and code reorderings for the real branch predictor is 6.306, compared with 5.729 for a simulated 8KB GAs predictor. A 16KB simulated GAs branch predictor yields 5.542 MPKI.

Figure 8 shows the predicted CPI for the various branch predictors as well as a perfect (0 MPKI) predictor using the performance model derived in the previous section. Each point in the graph shows a marker superimposed on error bars giving the 95% prediction interval for the benchmark's regression model. For the real branch predictor, the error bars indicate the tighter confidence interval since the data are observations and not predictions. Most of the benchmarks have reasonable prediction intervals even for the perfect predictor.

7.2.1 Perfect Branch Prediction

The real branch predictor yields an average CPI of 1.387 ± 0.012. The estimated CPI for perfect prediction is 1.223 ± 0.061. Thus, the performance improvement going from the current predictor to perfect prediction would be between 7% and 16%, with an average of 11.8%.

7.2.2 Academic Predictor

The L-TAGE branch predictor is currently the most accurate branch predictor in the academic literature [27]. We simulate this predictor using Pin and estimate the CPI yielded using our regression models. On average, L-TAGE yields 3.995 MPKI, compared with 6.306 MPKI for the real Intel predictor, an improvement of 37%. Our regression model estimates that this predictor would yield an average 1.320 ± 0.03 CPI, an improvement of between 2.4% to 6.8%, with an average of 4.8%. Several different sized GAs predictors are also shown. GAs predictors are simple global predictors uses in current microprocessors. The accuracy of GAs improves as its size grows.

7.2.3 Practical Concerns

We do not suggest that Intel should or should not replace their predictor with some other predictor. There are other concerns such as access latency to the prediction table that would guide such a decision. Our tool allows exploring the performance impact of hypothetical predictors before the decision is taken to spend design effort to accommodate them in a microarchitecture. For instance, it is possible that Intel could spare an extra 24KB for the L-TAGE branch predictor, but that the access latency and design complexity for such a structure might exceed the time allowed for branch prediction resulting in an unacceptable pipeline bubble. The design effort to include latency mitigating techniques [14] might not be worth the improvement in performance or delay in time to market. Nevertheless, our tool allows a quick way of evaluating many potential branch predictors for a given microarchitecture.

8 Conclusion

We have presented program interferometry, a technique for developing a performance model for programs running on a given microarchitecture based on the effect of code and data placement on certain microarchitectural structures. We have shown that program interferometry in many cases can give tight bounds on performance estimates for different branch prediction accuracies, allowing accurate evaluation of new branch predictors for existing microarchitectures without implementing a cycle-accurate simulator. In future work we will extend this technique to other structures.

References

[1] John E. Baldwin and Christopher A. Haniff. The application of interferometry to optical astronomical imaging. *Philosophical Transactions of The Royal Society*, 360(1794):969–986, May 2002.

[2] Emery D. Berger and Benjamin G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 158–168, New York, NY, USA, 2006. ACM.

[3] Brad Calder and Dirk Grunwald. Reducing branch costs via branch alignment. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[4] Gilberto Contreras and Margaret Martonosi. Power prediction for Intel XScale®processors using performance monitoring unit events. In *ISLPED '05: Proceedings of the 2005 international symposium on Low power electronics and design*, pages 221–226, New York, NY, USA, 2005. ACM.

[5] Rajagopalan Desikan, Doug Burger, and Stephen W. Keckler. Measuring experimental error in microprocessor simulation. In *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*, pages 266–277, New York, NY, USA, 2001. ACM.

[6] Rajagopalan Desikan, Doug Burger, Stephen W. Keckler, Llorenc Cruz, Fernando Latorre, Antonio González, and Mateo Valero. Errata on "measuring experimental error in microprocessor simulation". *SIGARCH Comput. Archit. News*, 30(1):2–4, 2002.

[7] D.J.Hatfield and J.Gerald. Program restructuring for virtual memory. *IBM Systems Journal*, 10(3):168–192, 1971.

[8] M. Evers, P.-Y. Chang, and Y. N. Patt. Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches. In *Proceedings of the 23rd International Symposium on Computer Architecture*, May 1996.

[9] Domenico Ferrari. Improving locality by critical working sets. *Communications of the ACM*, 17(11):614–620, November 1974.

[10] Nikolas Gloy and Michael D. Smith. Procedure placement using Temporal-Ordering information. *ACM Transactions on Programming Languages and Systems*, 21(5):977–1027, September 1999.

[11] Chunling Hu, John McCabe, Daniel A. Jiménez, and Ulrich Kremer. The camino compiler infrastructure. *SIGARCH Comput. Archit. News Special Issue on the 2005 Workshop on Binary Instrumentation and Application*, 33(5):3–8, 2005.

[12] Intel Corporation. Intel Pentium 4 processor optimization. Technical Report Order Number: 248966, Intel Corporation, 2001.

[13] Daniel A. Jiménez. Code placement for improving dynamic branch prediction accuracy. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Langu age Design and Implementation (PLDI)*, pages 107–116, June 2005.

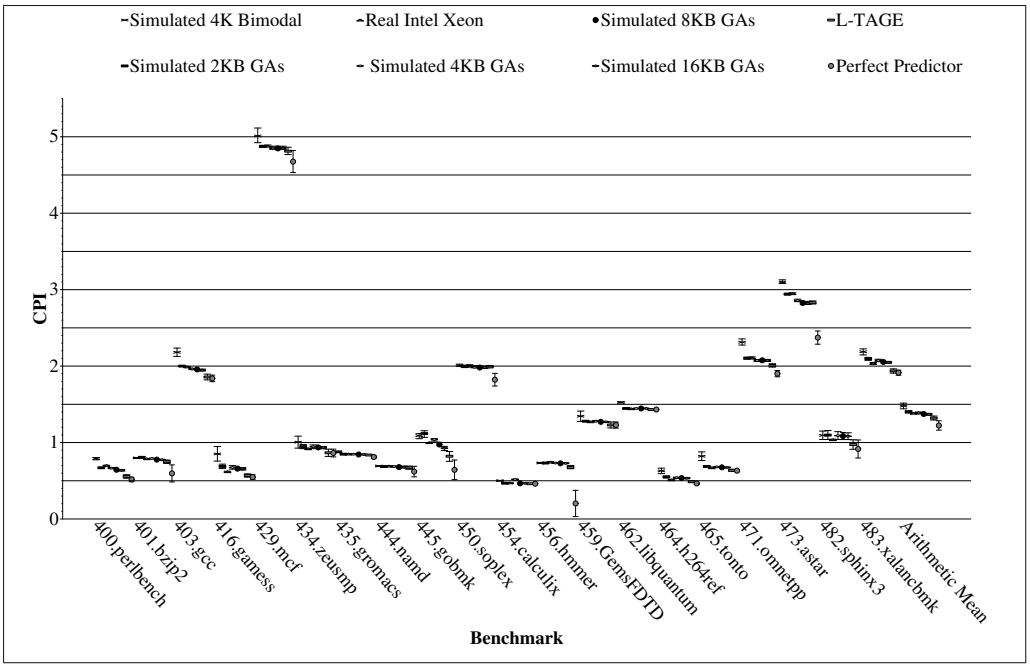

Figure 8. Predicted CPI of real and simulated branch predictors.

[14] Daniel A. Jiménez, Stephen W. Keckler, and Calvin Lin. The impact of delay on the design of branch predictors. In *Proceedings of the 33rd Annual International Symposium on Microarchitecture (MICRO-33)*, pages 67–76, December 2000.

[15] P. J. Joseph, Kapil Vaswani, and Matthew J. Thazhuthaveetil. A predictive performance model for superscalar processors. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 161–170, Washington, DC, USA, 2006. IEEE Computer Society.

[16] Dan Knights, Todd Mytkowicz, Peter F. Sweeney, Michael C. Mozer, and Amer Diwan. Blind optimization for exploiting hardware features. In *CC '09: Proceedings of the 18th International Conference on Compiler Construction*, pages 251–265, Berlin, Heidelberg, 2009. Springer-Verlag.

[17] Eric Larson, Saugata Chatterjee, and Todd Austin. MASE: A novel infrastructure for detailed microarchitectural modeling. In *Proceedings of the 2001 International Symposium on Performance Analysis of Systems and Software*, November 2001.

[18] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[19] Scott McFarling. Program optimization for instruction caches. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191. ACM, 1989.

[20] William Mendenhall, Dennis D. Wackerly, and Richrd L. Sheaffer. *Mathematical Statistics withh Applications, Fourth Edition*. PWS Publishers, Boston, MA, 1986.

[21] Pierre Michaud, André Seznec, and Richard Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 292–303, June 1997.

[22] Shirley Moore, David Cronk, Felix Wolf, Avi Purkayastha, Patricia Teller, Robert Araiza, Maria Gabriela Aguilera, and Jamie Nava. Performance profiling and analysis of dod applications using papi and tau. In *DOD_UGC '05: Proceedings of the 2005 Users Group Conference on 2005 Users Group Conference*, page 394, Washington, DC, USA, 2005. IEEE Computer Society.

[23] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 265–276, New York, NY, USA, 2009. ACM.

[24] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 16–27, June 1990.

[25] Eric Rotenberg, Steve Bennett, and James E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996.

[26] Shai Rubin, Rastislav Bodík, and Trishul Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 140–153, New York, NY, USA, 2002. ACM.

[27] André Seznec. A 256 kbits l-tage branch predictor. *Journal of Instruction-Level Parallelism (JILP) Special Issue: The Second Championship Branch Prediction Competition (CBP-2)*, 9, May 2007.

[28] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[29] James E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 135–148, May 1981.

[30] T.-Y. Yeh and Yale N. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th ACM/IEEE International Symposium on Microarchitecture*, pages 51–61, November 1991.

[31] Cliff Young, David S. Johnson, David R. Karger, and Michael D. Smith. Near-optimal intraprocedural branch alignment. In *Proceedings of the SIGPLAN'97 Conference on Program Language Design and Implementation*, June 1997.