

Program Interferometry

Zhe Wang and Daniel A. Jiménez

Department of Computer Science

The University of Texas at San Antonio

I. INTRODUCTION

This abstract 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 [2]. The situation is similar 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.

A. Varying Branch Prediction Accuracy

Figure 1 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.

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.

II. PROGRAM INTERFEROMETRY

A. Instruction Addresses in Microarchitectural Structures

Program interferometry exploits the fact that several microarchitectural structures use a hash of instruction and data addresses. Sometimes addresses will accidentally collide in some microarchitectural structure. Although this phenomenon has been studied in academic research, most compilers do not optimize to protect against these kinds of conflicts.

B. 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.

C. Causing Collisions

To generate many random but plausible code layouts, we extend the technique of Mytkowicz *et al.* [2] 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, 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.

D. 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 [1]. 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).

III. ESTIMATING PERFORMANCE BY COUNTING MICROARCHITECTURAL EVENTS

A. 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: branch mispredictions, L1 instruction cache misses, and LLC 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 2 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 reorderings is explained by branch misprediction.

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.

B. 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

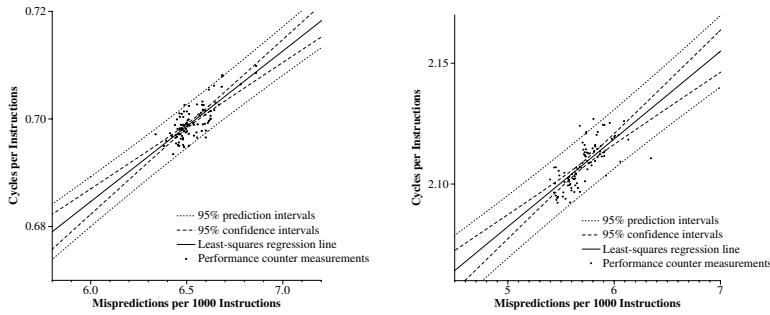


Figure 1. Performance changes with branch prediction accuracy for 400.perlbench and 471.omnetpp.

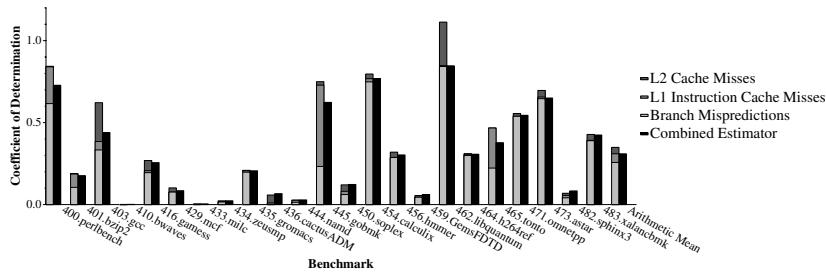


Figure 2. Coefficient of determination showing how much of each type of event accounts for overall performance.

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.

C. 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.

D. 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 derive 95% confidence intervals and 95% prediction intervals for the regression lines. Figure 1 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.

IV. ESTIMATING BRANCH PREDICTION PERFORMANCE

This section discusses 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. 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. We explore only those benchmarks that were demonstrated in the previous section to be suitable for program interferometry. The 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.

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%.

L-TAGE is currently the most accurate branch predictor in the academic literature [3]. 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%.

REFERENCES

- [1] William Mendenhall, Dennis D. Wackerly, and Richrd L. Sheaffer. *Mathematical Statistics with Applications, Fourth Edition*. PWS Publishers, Boston, MA, 1986.
- [2] 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.
- [3] 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.