Evaluating whether the training data provided for profile feedback is a realistic control flow for the real workload

Darryl Gove & Lawrence Spracklen
Scalable Systems Group
Sun Microsystems Inc., Sunnyvale, CA
{darryl.gove,lawrence.spracklen}@sun.com

Abstract

Profile feedback Optimization (PFO) techniques are used to improve the performance of compiler generated code. Under PFO, the code is compiled twice. The first compilation produces an instrumented version of the code. This instrumented version is then run with a 'representative' workload to generate a profile. This profile is used as an input into the second compilation of the code. One of the problems associated with using profile feedback is determining whether the workload used for training is appropriate.

In this paper we develop metrics to define whether or not a workload is representative. Using these metrics, we evaluate whether the training workloads provided for the SPEC CPU2000 benchmark suite are appropriate and illustrate that, for several of the benchmarks, there are significant problems.

This analysis was also used as part of the process of defining the workloads used in the forthcoming SPEC CPU2006 suite.

1 Introduction

When a program is compiled, the compiler usually has to make best guesses at whether a particular branch in the program is taken or untaken. Based on these guesses, the compiler makes a variety of decisions about how to best lay out the code, and whether particular routines should be inlined [1].

If the compiler guesses correctly, there is the potential for large performance gains. On the other hand, guessing wrongly may lead to missed opportunities for performance improvements.

To assist the compiler in making these decisions, there is the profile feedback mechanism [2, 3]. The method is to build an instrumented version of the binary. This binary is run on one or more representative workloads, in order to collect data on the frequency with which branches are taken and routines are called. The compiler uses this information to guide a second compilation, resulting in better decisions for code layout and inlining.

Despite these potential improvements in performance, it can be difficult to persuade developers to use profile feedback, and there are two common concerns:

1. **Build time**: using profile feedback requires two compiles of the application and one (or more) runs of representative workloads. This may mean that the entire build time for the application is more than doubled.

2. **Representative workloads**: the developer may be unsure of what constitutes a representative workload for their application. They may also be concerned whether the training data that they are using is representative of all workloads that the application will be run on.

While it is beyond the scope of this paper to discuss solutions to the build time problem, this paper will present information on what is a representative workload, and ways of visualizing this data so that it can be readily determined whether the training workload is representative or not.

2 Method

The SPEC CPU2000 [4] benchmark suite was used for this analysis. The benchmarks were compiled using the Sun Studio 11 compiler and were run on an UltraSPARC III processor. The suite has workloads of three different sizes for each benchmark in the suite. The *test* workload for each benchmark generally runs for less than a second, its purpose being to test that the benchmark compiled. The *training* workload is longer running, and this workload is used when compiling the benchmark with profile feedback. The final workload is the *reference* workload which is used for the timed run to generate the final scores.

CPU2000_int Benchmark	Correspondence between train and reference	CPU2000_fp Benchmark	Correspondence between train and reference
164.gzip	100%	168.wupwise	100%
175.vpr	100%	171.swim	100%
176.gcc	98%	172.mgrid	98%
181.mcf	100%	173.applu	100%
186.crafty	96%	177.mesa	96%
197.parser	99%	178.galgel	83%
252.eon	100%	179.art	100%
253.perlbmk	95%	183.equake	100%
254.gap	95%	187.facerec	100%
255.vortex	100%	188.ammp	100%
256.bzip2	96%	189.lucas	89%
300.twolf	100%	191.fma3d	100%
		200.sixtrack	100%
		301.apsi	72%

Table 1. Correspondence Values between train and reference workloads

To gather data on the correspondence between the branch behaviors of the various workloads, the codes were compiled without profile feedback, and with low optimization and then instrumented to collect data on both basic block frequency and the frequency with which each branch statement was taken or untaken. The instrumentation phase involved disassembling the applications, adding instrumentation code to the basic blocks, and then reassembling the applications. This introduces significant overhead into the run time, but does not alter the code paths.

The instrumented versions of the benchmarks were separately run on the training and reference workloads, leading to two sets of branch and basic block data for each benchmark.

In some cases a single 'workload' may comprise several runs of the application on different datasets. For these benchmarks, the behavior over all the datasets was collected.

3 Analysis using branch taken probabilities

For each branch statement in an executable, data was collected on both the number of times that branch statement was encountered, and the number of times the branch was taken.

This data was collected for the train and reference workloads, and the results compared. The objective of this was to determine whether the train workload is 'representative' of the reference workload. The following definition of representative is proposed:

> *A representative training workload is one in which the behavior of each individual branch statement is similar to its behavior in the reference workload.*

The above definition is still imprecise, because it relies on the word 'similar'. In fact, the definition is good, but there are several ways of interpreting it.

- **Static or dynamic**: a critical difference is the number of static branches compared the number of dynamic branches. A static branch is one that appears in the disassembly of the application, a dynamic branch is a branch statement that gets encountered at runtime (accordingly, a single static branch can be encountered multiple times at runtime, and hence contribute many dynamic branches). For the purposes of this paper, the dynamic behavior of the branches is considered more important. The reason for this is that performance typically comes from improvements to the layout of the code that are derived from knowing the branch probabilities. The potential to improve performance depends on how often that code is used during the reference run of the benchmark. Hence improvements to the layout of code that is only encountered a few times during the run will have little opportunity to impact the runtime.
- **Probabilities**: although branches have probabilities of being taken or untaken, this level of detail can be reduced to whether the branch is usually taken or usually untaken. Put another way, a branch whose probability of being taken is greater than a half, is a usually taken branch. This is a useful simplification, since there is little practical difference between a branch that is taken 80% of the time, compared to a branch that is taken 90% of the time - both probabilities imply the same most likely code path. Using this simplification, one

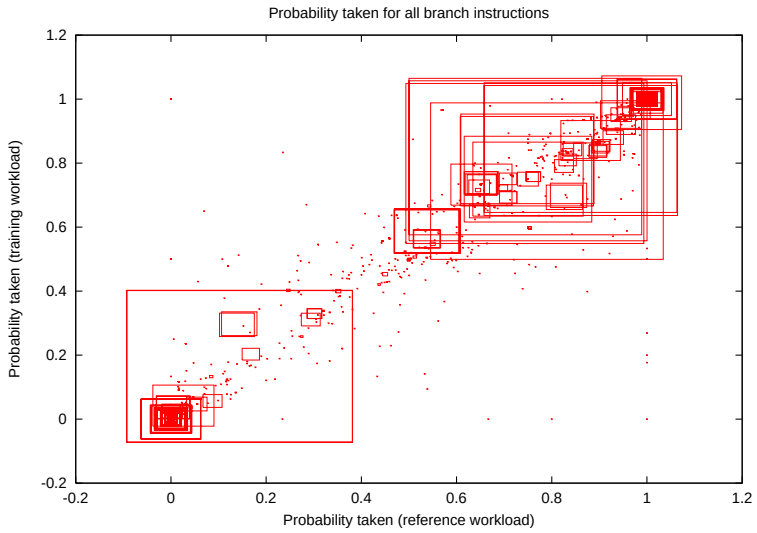

Figure 1. 300.twolf branch probabilities

measure of 'representativeness' is whether the branch is usually taken or usually untaken in both the training and reference workloads.

Taking these observations into account, a more precise definition of representative can be proposed:

> *A representative training workload is one for which each static branch is either usually taken by both the training and reference workloads, or usually untaken by both of them.*

Using this definition, it is possible to determine a Correspondence Value (CV), as illustrated in Equation 1. In Equation 1, F_b is the number of times that a branch b is executed in the reference workload, TT_b has the value one if branch b is usually taken in the training workload (or zero otherwise) and RT_b has the value 1 if branch b is usually taken in the reference workload (or zero otherwise).

$$CV = \frac{\sum_b (F_b * (TT_b == RT_b))}{\sum_b F_b} \qquad (1)$$

In this formula, the Correspondence Value ranges from zero, indicating that there is no agreement on branches between the training and reference workloads, to one where all the branches in both the training and reference workloads agree.

4 Results using the Correspondence Value

The Correspondence Value can be calculated for the workload pair of train and reference. Table 1 shows the results for the benchmarks in SPEC CPU2000.

From the results, it is apparent that most benchmarks have good agreement between the behavior of the branches in the training and reference workloads. However, three of the benchmarks show poor agreement: 178.galgel, 189.lucas, 301.apsi.

While it is useful to have a set of absolute numbers that give an indication of the degree of agreement between the workloads, it is rather hard to interpret these results. To make the results easier to understand, it is possible to present them graphically; the x and y axes on the charts are the probability that a given branch is taken, and the size of the mark made on the chart indicates the number of times that branch was executed in the reference workload.

Figures 1 through 4 show results from four of the 26 benchmarks in SPEC CPU2000; 178.galgel, 301.apsi, 186.crafty, and 300.twolf. The x-axis in each graph shows the probability of a branch being taken in the reference workload. The y-axis shows the probability of the branch being taken in the training workload. Consequently branches that are in the upper right or lower left quadrants are where the training and reference workloads agree, branches that appear in the other two quadrants are where the behavior in the training workload is not representative of the behavior in the reference workload. The size of the

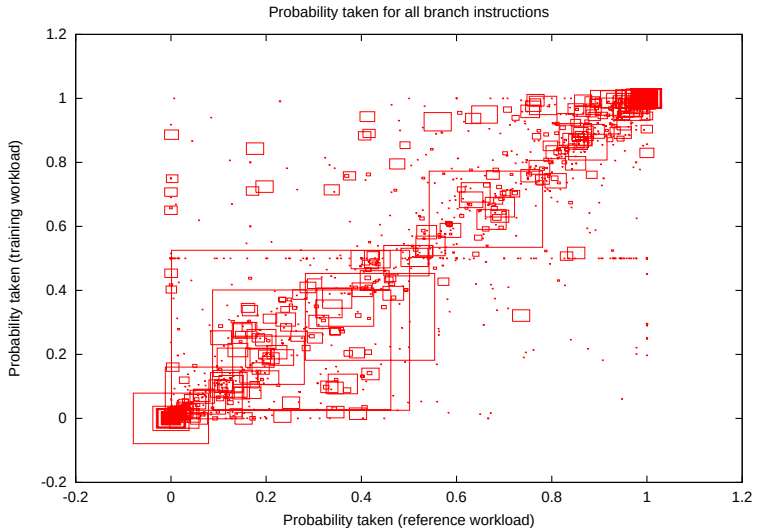

Figure 2. 186.crafty branch probabilities

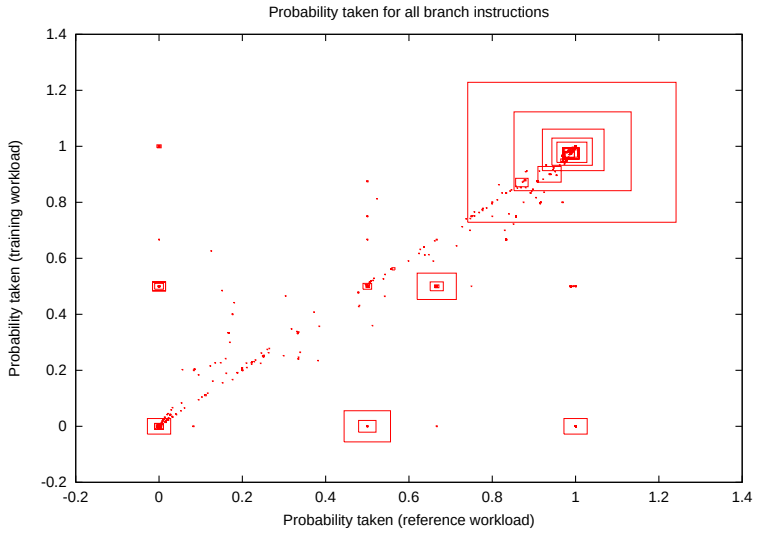

Figure 3. 178.galgel branch probabilities

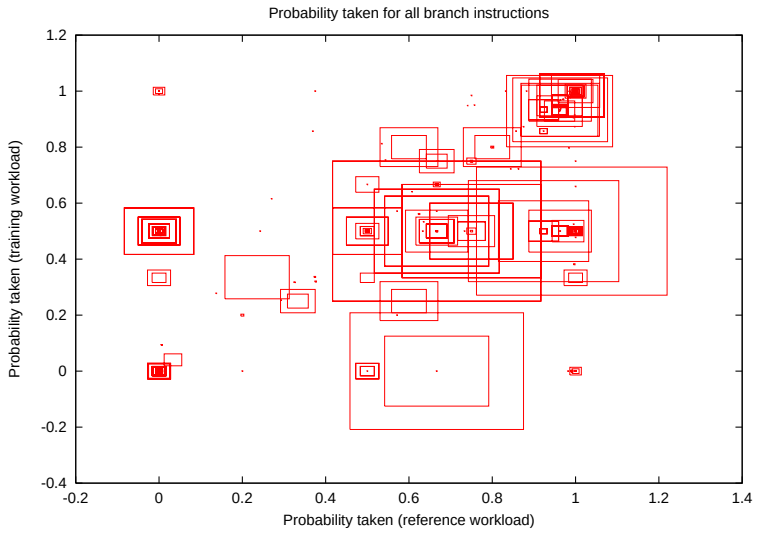

Figure 4. 301.apsi branch probabilities

mark used to plot each data point is determined by the number of encounters for that branch in the reference workload as a proportion of the maximum number of encounters for any single branch . A frequently encountered branch will have a large square, a branch that is rarely encountered will be denoted by a small dot. Branches that would not appear on the chart using this metric have had their point size increased so that they are visible.

Focusing on each workload individually, it is possible to observe:

- 300.twolf (Figure 1) shows very good correspondence between the training and reference workloads. The branches are all plotted in either the top right (indicating that they are usually taken for both the training and reference workloads), or in the bottom left (indicating that they are usually untaken in both training and reference workloads) quadrants. Furthermore the majority of the branches that are frequently encountered in the reference workload appear on the diagonal, which indicates that the probability of them being taken is similar for both workloads.

- 186.crafty (Figure 2) has many branches, the majority of the branches are plotted in either the top right or bottom left quadrants (indicating a good agreement between training and reference workloads). The branches are spread around the diagonal indicating that the probability of being taken in the training and reference workloads are not exactly the same, but this is almost certainly a characteristic of the application rather than a weakness in the quality of the training workload. There are also a small number of branches that appear in the top left and bottom right quadrants, indicating a disagreement between the training and reference workloads, again this is probably to be expected.

- 178.galgel (Figure 3) has a few slightly important branches that are plotted in the bottom right quadrant. These represent branches which are usually taken in the reference workload, but untaken in the training workload. The most frequently executed branches are in the upper right quadrant, indicating that they are usually taken for both the training and reference workload. This is to be expected of a floating point benchmark which has a number of loops with high trip count.

- 301.apsi (Figure 4), has a number of branches which are poorly predicted by the training workload. There are several significant branches that are plotted in the lower right quadrant, indicating that they are usually taken by the reference workload, but usually untaken by the training workload. There are also a number of branches, which are frequently executed by the reference workload, that are usually taken by the reference workload, but are only taken half the time by the training workload.

CPU2000_int Benchmark	Coverage of reference by train	CPU2000_fp Benchmark	Coverage of reference by train
164.gzip	100%	168.wupwise	100%
175.vpr	100%	171.swim	100%
176.gcc	100%	172.mgrid	100%
181.mcf	100%	173.applu	100%
186.crafty	100%	177.mesa	98%
197.parser	100%	178.galgel	85%
252.eon	100%	179.art	100%
253.perlbmk	100%	183.equake	100%
254.gap	99%	187.facerec	100%
255.vortex	100%	188.ammp	100%
256.bzip2	100%	189.lucas	81%
300.twolf	100%	191.fma3d	100%
		200.sixtrack	100%
		301.apsi	37%

Table 2. Coverage between train and reference workloads

5 Analysis using basic blocks

An alternative approach is to use the execution counts for basic blocks (a basic block is a small segment of code which has only one entry point and all the instructions in the block are executed the same number of times). An advantage of this approach is for the situation where a single branch has multiple possible targets (the exact target determined at runtime), in this situation it is not possible to assign a branch probability, but it is possible to look at the frequency of execution of all the target basic blocks.

In the case of branches, it is possible to determine the number of times that a branch is taken out of the total number of times that it could have been taken, and hence derive a single valued probability of being taken for that branch. Unfortunately, basic block counts lack a denominator that could be used to convert them into a similar probability. Furthermore, basic block counts will scale with runtime, so the longer the run of the application, the more times each basic block is executed. It is therefore necessary to turn the basic block counts into something which has no dependency on the length of the run.

The simplest way of converting basic block counts into a single value is to use them to derive coverage data for the pair of workloads. This leads to the following description of a representative workload:

> *A representative training workload will exercise all the frequency executed basic blocks of the reference workload.*

This can be represented as follows: let Ct_i be a count of the number of times that the basic block was executed in the training workload. Let Cr_i be the number of times that the basic block was executed in the reference workload. The coverage can be calculated as illustrated in Equation 2.

$$coverage = \frac{\sum_i Cr_i * (Ct_i > 0)}{\sum_i Cr_i} \tag{2}$$

The value for coverage runs from zero, indicating that none of the frequently executed basic blocks in the reference workload are executed by the training workload, to one indicating that the training workload executes all the basic blocks that the reference workload uses.

Other, more elaborate, formulae for handling basic block counts were considered. However the more complex formulae produced results which were more subject to interpretation than this simple measure of code coverage. The other consideration is that a training workload which fails to even cover the critical sections of the real workload is obviously unable to provide correct training data.

6 Results using basic block coverage

The results for basic block coverage are illustrated in Table 2. The data shows that the training workloads for most of the benchmarks cover all the important basic blocks for the reference workloads. Unfortunately, three benchmarks have training workloads that are inadequate: 178.galgel, 189.lucas, and 301.apsi. With 37% coverage, 301.apsi has very low coverage of the reference workload by the training workload.

Again, it is easier to interpret this data when presented in a graphical format. In the graphs depicting coverage, the basic blocks are sorted into order of increasing execution count. The x-axis shows the basic block ordering for the

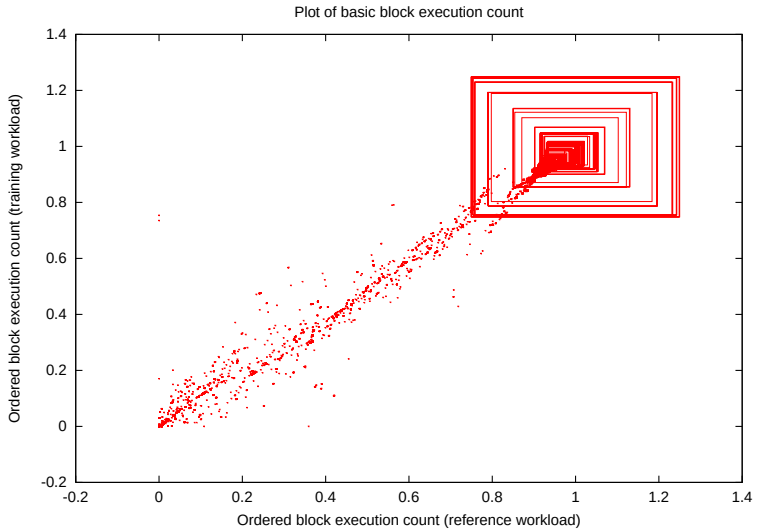

Figure 5. 300.twolf code coverage agreement

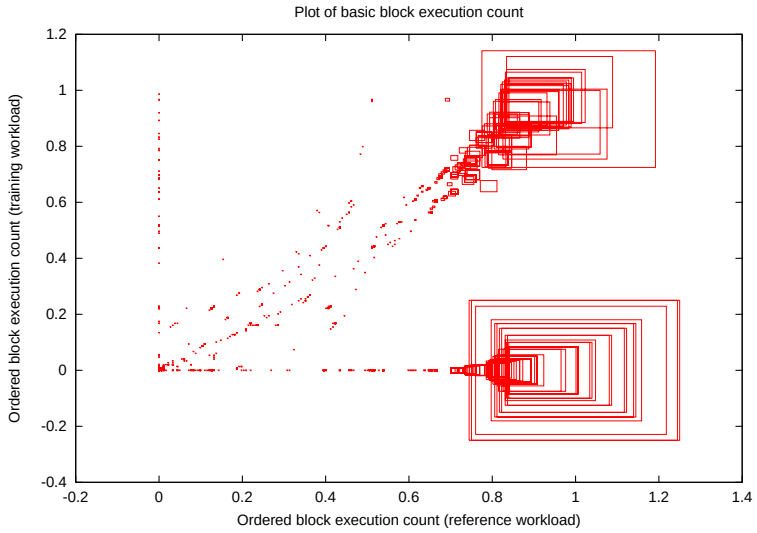

Figure 6. 301.apsi code coverage agreement

reference workload, the y-axis shows the basic block ordering for the training workload. A basic block which has a low execution count for both the training and reference workloads will appear at the origin; a basic block which has a high execution count for both the training and reference workloads will appear at the top right of the chart. The size of the mark used for each data point corresponds to the execution count of the basic block in the reference workload scaled against the execution count for the basic block with the highest execution count. Blocks that would not appear on the chart using this metric have had their point sized increased so as to be visible.

It is worth observing that the size of the mark is correlated with the order of the block on the x-axis, but the size of the mark conveys additional information. Two basic blocks may have adjacent ordering, but have very different execution counts.

In these charts, the ideal shape would be a line that from bottom left to top right that indicates that the basic blocks in the training dataset are ordered in the same way as the basic blocks in the reference dataset. A shape which indicates poor agreement, between the training and reference workloads, would manifest as important basic blocks appearing significantly off the diagonal.

The charts convey slightly more information than the calculations of coverage. The coverage calculation is just evaluating the important basic blocks in the reference workload that are not executed in the training workload. These basic blocks will appear along the y=0 axis in the chart. The charts display more information than this, since they will also highlight blocks which are rarely executed in the training workload but frequently executed in the reference workload.

Figures 5 and 6 show the basic block count correspondence for the benchmarks 301.apsi and 300.twolf and illustrate:

- 300.twolf (Figure 5) has the shape of a 'lolly-pop' all the basic blocks that have high execution count in the reference workload are also the most frequently executed blocks in the training workload.
- 301.apsi (Figure 6) has a problem where the frequently executed basic blocks in the reference workload are not executed in the training workload.

7 Discussion of coverage data

It would seem reasonable to expect that the basic block count data could be used in more advanced ways than to just calculate coverage. In fact, it would seem that coverage would be such a simple measure that it would not help. However, coverage actually turns out to be a useful measure for the following two reasons:

1. It is easy to calculate, and there is no ambiguity or subjectiveness in the results.
2. It is a low bar for representativeness of the code, training workloads which do not cover the important sections of the application cannot train those sections.

Several alternatives were considered, such as a correlations based on the order of the basic blocks or a comparison of the profiles of the two workloads. However these had a number of issues:

- The formulae made some assumptions about the distribution of the data, and these assumptions may not have been valid for all situations.
- The results of the formulae were often skewed either by the number of unimportant blocks, or by a slight difference in ordering for the important blocks. This rendered the formulae either very insensitive, or very sensitive.
- If the training workload over-trained an area of the code, this changed the profile making it dissimilar to the reference workload, even if the critical sections of the workload matched.
- The calculations produced numbers which were statistics with no connection to a characteristic of the actual code.

8 Related Work

There is a significant body of research focussed on feedback profile optimizations and workloads. In [5], the effectiveness of the training profile is investigated by examining the performance benefits observed by leveraging a specific profile. In [6], the author discuses the problems associated with generating profile data for dynamic libraries, which are used by numerous applications in varying ways. In [7], the authors introduce a metric, instructions per mispredicted branch, for determining the importance of profile feedback optimizations for a specific application. They use this to evaluate whether training data will result in a significant performance gain, which is a complementary approach to that taken in this paper, where we reduce the problem space by focusing on just the branch and basic block behaviors.

Much of the recent research that is relevant to our discussion of 'representative' workloads for feedback profile optimizations has been undertaken, not in the context of FPO analysis, but rather to reduce the simulation space for processor design. Rigorous, data-driven, processor design, requires simulation time of target workloads on cycle-accurate simulators. These simulators are slow and, as a result, it is infeasible to simulate the entirety of every target

application. In [8], the authors investigate the validity of using smaller datasets to decrease the simulation of the target applications. Looking at SPECint2000, the authors illustrate that the use of the short-duration test or training workloads impacts procedure coverage, IPC, cache miss rates and impacts the frequent execution paths. In [9], the authors investigate the use of a single application to represent the behavior of a wide class of target applications. Again, the authors wrestle with the definition of a representative workload. In [10], the authors undertake a clustering analysis and measure the similarity of the benchmarks in four generations of the SPEC CPU suite.

Our paper builds on this research, not only by investigating the entire SPEC CPU benchmark suite, but also by introducing quantitative metrics that can be used to define the degree to which a workload is 'representative'.

9 Conclusions

This paper evaluates the agreement between training and reference workloads in the context of code paths. It does not attempt to determine whether the cache access behavior is similar for the two workloads, or many other ways in which two workloads could be evaluated for 'representativeness'. Code path optimizations are the most common use of profile feedback information, so it is most important to ensure that the training workloads are at least meeting this criteria.

Two approaches are presented in this paper for evaluating whether a given training workload is 'representative' of a particular reference workload. These are to evaluate whether the branches are exercised in similar ways by the two workloads, and also to evaluate whether the training workload covers the same code as the reference workload.

The code coverage can be seen as the lowest bar for a workload to be declared representative. If a training workload fails to cover critical sections of the reference workload, then it has failed in its required purpose. To put this another way, there is no point in running a training workload that imparts no information that can be used in improving the important parts of the code.

The evaluation of branch behaviors is a more detailed level of analysis. It may be, as in the benchmark 186.crafty, that the behavior of the branches is somewhat unpredictable. In which case, searching for a more representative training workload may be a futile exercise.

The benchmark 301.apsi demonstrates that the two approaches complement rather than contradict each other. Using both methods, this benchmark comes out showing serious problems in the ability of the training workload to adequately represent the reference workload. The coverage data clearly shows that the problem lies with large sections of the critical code for the reference workload not being executed by the training workload.

Using the techniques presented in this paper it is possible to evaluate the representativeness of the workload, and come to a conclusion as to whether the workload used in the profile feedback build is representative of the various workloads that will be run on the application. Evaluating both the coverage of the training workload, and also the behavior of the individual branches is a necessary step in evaluating whether the training workload is 'representative'. The findings from this evaluation can be used to determine whether to change the training workload, or whether to add additional training workloads to improve coverage.

10 Acknowledgments

Joel Williamson and Mario Wolczko for comments on earlier drafts of this paper. Chet Wood for providing the tools infrastructure. John Henning for instigating this analysis.

References

[1] D. Wall, "Predicting program behavior using real or estimated profiles," in *Proc. Conf. on Programming language design and implementation*, 1991, pp. 59–70.

[2] R. Cohn and P. Lowney, "Feedback directed optimization in Compaq's compilation tools for Alpha," in *Proc. Workshop on Feedback Directed Optimization*, 1999.

[3] E. Albert, "A Transparent Method for Correlating Profiles with Source Programs," in *Proc. Workshop on Feedback-Directed Optimization*, 1999.

[4] J. L. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millennium," *Computer*, vol. 33, no. 7, pp. 28–35, 2000.

[5] G. Langdale and T. Gross, "Evaluating the Relationship Between the Usefulness and Accuracy of Profiles," in *Proc. Workshop on Duplicating, Deconstructing, and Debunking*, 2003.

[6] S. McFarling, "Reality-based optimization," in *Proc. Intl. Symp. on Code generation and optimization*, 2003, pp. 59–68.

[7] J. A. Fisher and S. M. Freudenberger, "Predicting conditional branch directions from previous runs of a program," in *Proc. Intl. Conf. on Architectural support for programming languages and operating systems*, 1992, pp. 85–95.

[8] W. Hsu, H. Chen, P. Yew, and D. Chen, "On the Predictability of Program Behavior Using Different Input Data Sets," in *Proc. Workshop of Interaction between Compilers and Computer Architectures*, 2002.

[9] L. Eeckhout, H. Vandierendonck, and K. De Bosschere, "Workload Design: Selecting Representative Program-Input Pairs," in *Proc. Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2002, pp. 83–94.

[10] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John, "Measuring Program Similarity: Experiments with SPEC CPU Benchmark Suites," in *Proc. Intl. Symp. Performance Analysis of Systems and Software*, 2005.