Evaluating the correspondence between training and reference workloads in SPEC CPU2006

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

Abstract

Profile feedback (sometimes called Feedback Directed Optimisation FDO) is a useful technique for providing the compiler with additional information about runtime program flow. The compiler is able to use this information to make optimisation decisions that improve the way the code is laid out in memory or determine which routines are inlined, and hence improve the performance of the application.

The use of profile feedback requires the code to be compiled twice. The first time the compiler generates an instrumented version of the application. This instrumented version is then run on one or more 'representative' training workloads to gather profile data. This profile data contains information such as how many times each routine is executed and how frequently each branch is taken. The second pass through the compiler uses this information to make more enlighted optimisation decisions.

The quality of the training data impacts the ability of the compiler to do the best job that it can. This paper discusses a method of assessing the similarity of the training workload to the reference workload, and applies this methodology to evaluate the training workloads in the SPEC CPU2006 benchmark suite.

1 Introduction

When an application is built, the compiler has to make the best guess that it can as to what the most frequently occurring path through the code will be. This guess will impact how branch statements are laid out, which routines are worth inlining, and even low level details such as which variables are held in registers and which are spilled to memory. Given the complexity of decisions to be made, the compiler is unlikely to guess correctly in all cases. The cost of making the wrong decision is often a missed opportunity to make the application run faster, the sum of these missed opportunities can be a significant loss of runtime performance.

One way to help the compiler make the correct decision is to use profile feedback. The idea here is to build an instrumented version of the application, run this instrumented version on a training workload, then use the data that is collected to build a final version of the application. The instrumentation typically gathers data on which branches are usually taken and which routines are frequently called. Using this information, the compiler can make very good decisions about how to lay out the basic blocks, and which routines should be inlined. However, the quality of the decisions is determined by the correspondence between the training workload and the actual ('reference') workload which is the real use for the application.

There are many different attributes that could be collected under profile feedback. For example, an application could be profiled to determine where the cache misses occur. However, many of these are very dependent on the actual implementation of the hardware. Cache miss behaviour, for example, is largely determined by the attributes of the cache hierarchy. Even the runtime of the application when built with different training workloads does not guarantee that the workload that produces the fastest final runtime on one platform is the best training workload for all platforms. The metrics proposed in this paper focus on program flow which is approximately architecture neutral. Even program flow is not perfectly platform independent since it can be altered by architecture dependent factors such as floating point rounding, predicated execution, and library implementation.

The recently released SPEC CPU2006 benchmark suite [1] includes training workloads for profile feedback. A change from the older CPU2000 suite is that profile feedback can only be used under peak runs of the suite. As part of the development of the SPEC CPU2006 benchmark suite, the training workloads were evaluated using the methodology proposed in [2]. The objective was not necessarily to provide perfect training data in all cases, but to ensure that

the training data was reasonable. There is a relatively compelling argument that in developing a benchmark suite, the objective is not to exclude all traces of real world imperfections.

2 Measuring the quality of the training workload

This section describes the methodology used in evaluating whether or not the training workload is representative of the reference workload. The methodology used is covered in detail in [2], and the implementation is covered in more detail in [3].

The binaries for all of the benchmarks were instrumented using BIT [4]. This instrumentation gathered counts for the number of times each branch instruction was taken and untaken, and counts for the number of times each 'basic block' (a basic block of code is a small chunk of assembly language code with one entry point and one, or more, exit points) of code was executed. The count data was collected for both branches and basic blocks for all the reference and training workloads. For benchmarks (such as 400.perlbench) where there are multiple training and reference workloads, the counts were aggregated over all the training and all the reference runs.

There are two metrics that are relatively simple to calculate, and also have an intuitive meaning:

- Coverage. The coverage reflects the number of basic blocks in the code that are used by the reference workload that are also used by the training workload.
- Branch correspondence. The correspondence indicates the proportion of branch instructions where both the training and reference workloads behave the same way. For example, the branch is usually taken by both the training and reference workloads.

The coverage is important because the compiler gains no useful information from a block of code that is not exercised by the training workload. In fact the compiler gets the negative information that the block of code is not executed, and therefore may not be worth optimising at all.

The coverage is calculated using the following procedure. The number of times each basic block is executed by the reference workload is summed up for those basic blocks which are also executed by the training workload. This is expressed as a proportion of the sum of the execution counts for all the basic blocks executed by the reference workload. The coverage of a program can be expressed using equation 1. Let $XCref_i$ denote the number of times that basic block i is executed during the reference workload. Similarly $XCtrain_i$ denotes the number of times that the basic block is executed during the training workload.

$$coverage = \frac{\sum_{bb} \begin{cases} XCref_{bb} & XCtrain_{bb} > 0 \\ 0 & \text{otherwise} \end{cases}}{\sum_{bb} XCref_{bb}} \quad (1)$$

The formula for coverage, expressed as a percentage, is a value between zero and 100%. If all the basic blocks that are used by the reference workload are also used by the training workload, then the value for coverage will be 100%. If none of the blocks used by the reference workload are executed by the training workload, the value for coverage will be zero. The weighting of each basic block by the execution count in the reference workload means that a value for coverage of nearly 100% can be obtained even when the training workload does not execute *all* of the basic blocks used by the reference workload.

The correspondence value is calculated by summing the execution count for every branch under the reference workload for all the branches where the branch is either usually taken for both the training and reference workloads, or usually untaken for both the training and reference workloads. This is then reported as a proportion of the total execution count for all branches in the reference workload.

The branch correspondence value can be expressed as equation 2. Let $ECref_b$ represent the number of times that branch statement b is encountered over the run of the application. Let $Tref_b$ be one if the branch is usually taken during the reference run, and zero otherwise. Let $Ttrain_b$ be one if the branch is usually taken during the training run, and zero otherwise. Let CV denote the correspondence value for this particular application.

$$CV = \frac{\sum_{b} \begin{cases} ECref_b & Tref_b = Ttrain_b \\ 0 & \text{otherwise} \end{cases}}{\sum_{b} ECref_b} \quad (2)$$

The correspondence value, expressed as a percentage, is a value between zero and 100%. A value of zero means that all the branches that are usually taken in the reference workload are usually untaken in the training workload (and visa versa). A value of 100% indicates that all the branches that are usually taken in the reference workload are also usually taken by the training workload (and visa versa). The correspondence value is weighted by the number of times that the branch is encountered during the reference run, consequently a value of nearly 100% can be obtained so long as the behaviour of each frequently encountered branch agrees in the training and reference workloads.

3 Graphing coverage and correspondence

The values obtained for coverage and correspondence are helpful in determining if there is an issue, but they do not provide insight into what the issue is. This section discusses how to present both coverage and correspondence in a graphical format. This format makes it easier to identify whether the poor result is due to a few significant points of difference, a multitude of smaller differences, or a borderline change in the behaviour (for example where a branch is taken about 50% of the time).

For both coverage and correspondence values, the graphs are x-y plots, the size of the marker is proportional to the frequency of execution of the basic block (or branch) as a proportion of the execution frequency for the most frequently executed block (or branch). The most frequently executed blocks will have the largest markers. The marker size for the infrequently executed blocks has been artificially increased to render them visible.

To show the coverage information, the basic blocks are sorted in increasing order of execution frequency. These are then plotted on a scale that runs from zero to one; the most frequently executed basic block being assigned the value 1, the least frequently the value 0. The basic blocks in the reference workload are plotted along the x-axis and the basic blocks for the training workload on the y-axis. A result of this is that blocks which are frequently executed in both the training and reference workloads appear at the top right of the graph. It is these frequently executed basic blocks that give the graph a shape reminiscent of a 'popsicle'. Frequently executed basic blocks which are not covered by the training workload will be shown by large markers plotted on the x-axis. Hence deviation from the 'popsicle' shape indicates a mismatch between the training and reference workloads.

Branch correspondence can be plotted using the probability of the branch being taken by the reference dataset as the value for the x-axis, and the probability of being taken in the training workload on the y-axis. Branches that get plotted in the upper right quadrant are those that are usually taken in both the training and reference workloads - these branches are well trained. Similarly branches that appear in the lower left quadrant are ones that are usually untaken by both the training and reference workloads. The branches that appear in either the upper left or lower right quadrant are those that are mistrained.

4 Results

Table 1 shows the results of applying these formula to the Integer benchmarks in SPEC CPU2006. 483.xalancbmk and 464.h264ref both score low on coverage of the critical parts of the code. 483.xalancbmk also scores low correspondence for the branch behaviour. 483.xalancbmk is an XML parser, so it might be expected that the branches behave differently under different workloads, however it is unexpected that the training workload does not cover all the code that is executed by the reference workload. Similarly the training workload for 464.h264ref does not cover all the code that is critical for the reference workload. The coverage graph for 464.h264ref is shown in Figure 1. From inspecting the graph it is apparent that the low value for coverage comes from a number of smaller routines which are not executed during the training workload.

Integer benchmarks	Coverage	Correspondance value
400.perlbench	99%	98%
401.bzip2	100%	100%
403.gcc	97%	96%
429.mcf	100%	99%
445.gobmk	100%	99%
456.hmmer	99%	100%
458.sjeng	100%	97%
462.libquantum	100%	100%
464.h264ref	92%	97%
471.omnetpp	100%	95%
473.astar	100%	100%
483.xalancbmk	88%	91%

Table 1. Results for Integer benchmarks in CPU2006

The chess benchmark, 458.sjeng, also has poor branch correspondence, this is shown in Figure 2. As might be expected for a game playing benchmark, this is the result of many unpredictable branches rather than one or two mispredicted branches.

Table 2 shows the results of applying these formula to the Floating Point benchmarks in SPEC CPU2006. The benchmark 416.gamess has poor coverage, this is shown in Figure 3. It is apparent from the graph that, for this benchmark, several key routines are not exercised by the training workload. The branch correspondence for this benchmark is also low. This is shown in Figure 4. Most of the branches are in either the upper right or lower left quadrants, but there are two significant branches that appear in the other quadrants, and the most significant branch is taken about 50% of the time in both training and reference workloads.

From the graph of the coverage for 481.wrf, shown in Figure 5, it is apparent that the reason for its low coverage score is that several significant routines are not executed by the training workload.

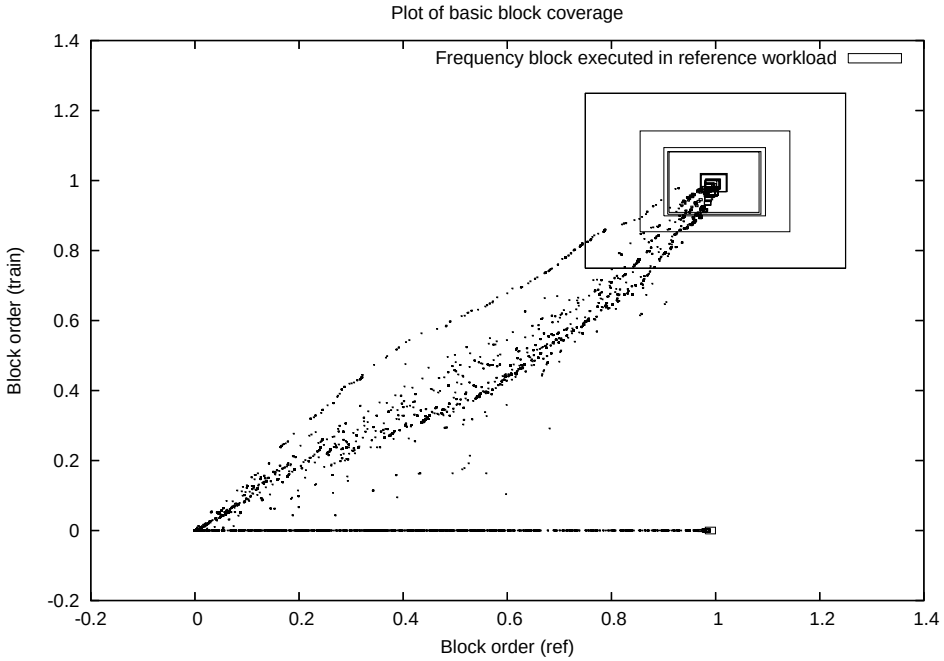

Figure 1. Basic block coverage for 464.h264ref

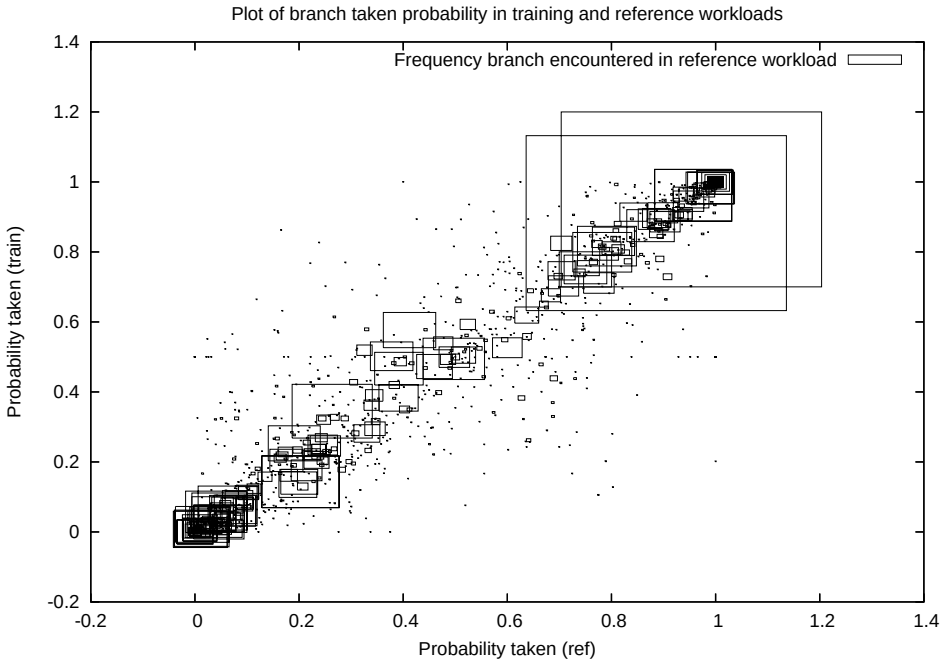

Figure 2. Branch correspondence data for 458.sjeng

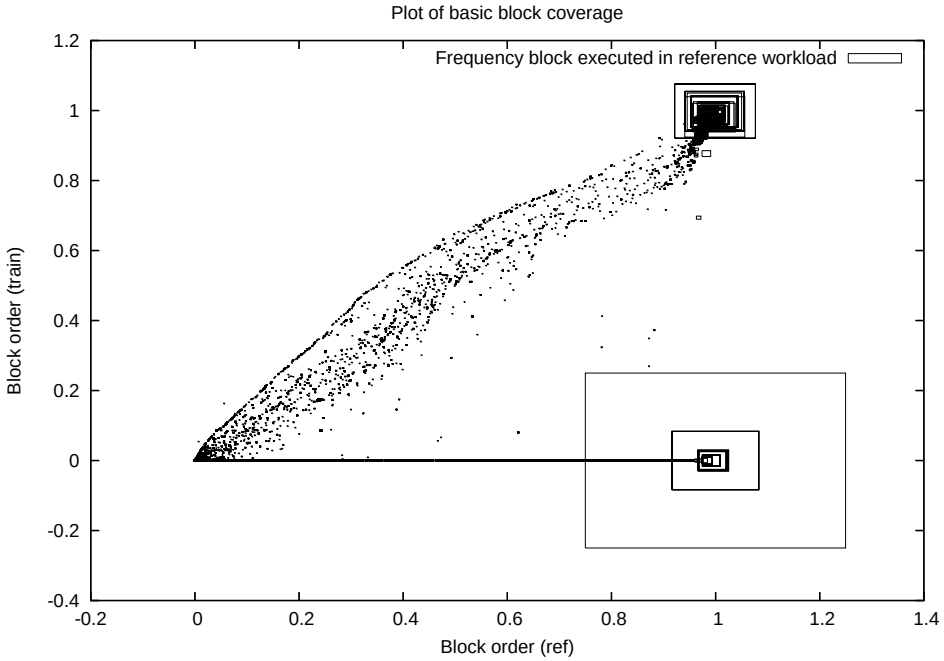

Figure 3. Basic block coverage for 416.gamess

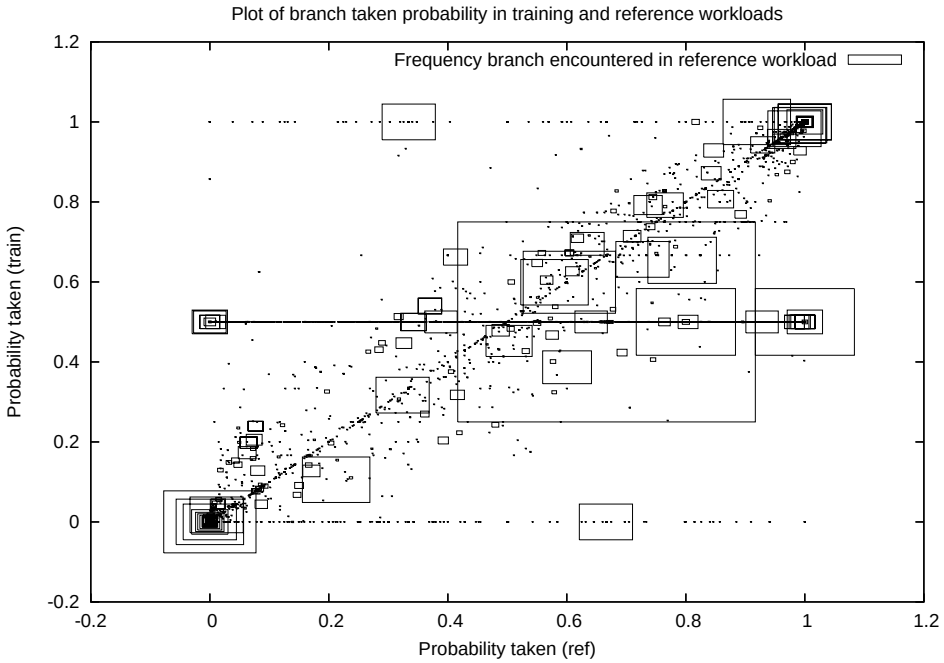

Figure 4. Branch correspondence data for 416.gamess

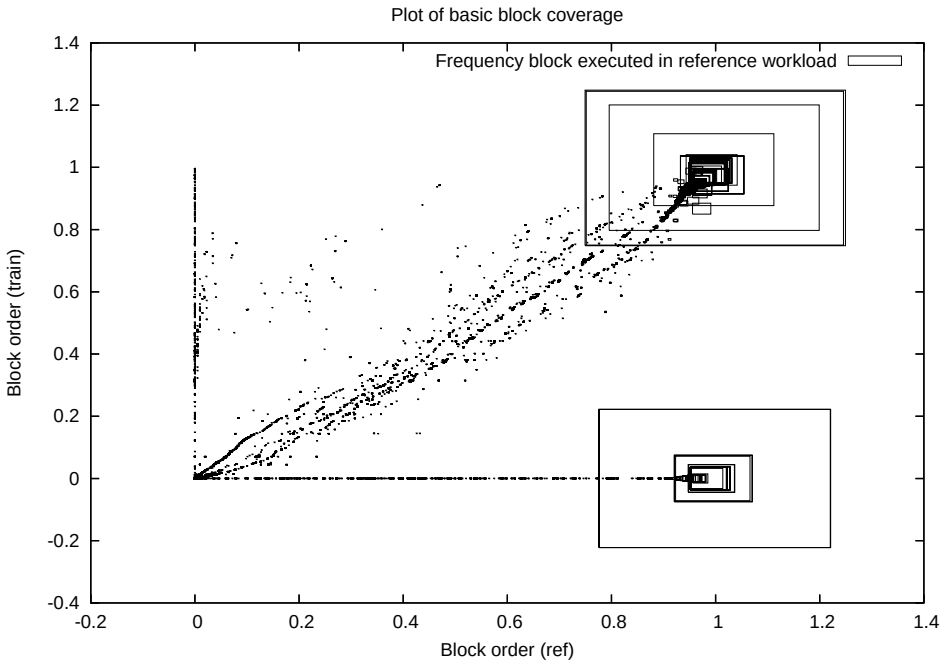

Figure 5. Basic block coverage for 481.wrf

5 Examining 481.wrf in more detail

The benchmark 481.wrf scores very poorly on both coverage and correspondence metrics. Given the low coverage score, it is not surprising that the correspondence is similarly low, since the training workload cannot correctly train those branches that it does not encounter. It is possible to generate profiles showing execution counts for each assembly language instruction for both the training and reference workloads. These instruction level counts can be aggregated to both instructions executed for each line of source, and instructions executed for each routine.

It would be possible for the reference and training instruction count profiles for the benchmark to look very similar when aggregated at the function level if, as an example, the difference in basic block coverage was due to an if condition being true for the training run and false for the reference run.

The instruction count profile of 481.wrf (number of instructions executed in each routine as a percentage of the total instructions) is shown for the top 17 routines in Table 3. Of the first 17 routines two routines are not present in the profile from the training workload. These routines are *clphyld_* and *sint_*. Examining the call stacks for these two routines shows that the routine *sint_* is called from the routine *med_nest_initial_* which is also not called in the training workload, and *clphyld_* is called by *microphysics_driver*. However, *microphysics_driver* is called by the training workload. Examining the program flow in the *microphysics_driver* routine it is apparent that different 'schemes' are used by the training and reference workloads. In the training workload the scheme *WSM3* is used, whereas the reference workload uses the *Lin* scheme.

In this situation it is apparent that the training workload needs to be improved to cover more of the code used by the reference workload. One approach would be to remove the existing training workload, and replace it with a new variant which does exercise the appropriate routines. However, the current training workload is, presumably, representative of some other real workload. Consequently a better approach would be to add another training workload to exercise those routines used in the reference workload. The advantage of doing this would be that the training workloads would train for a wider-range of uses of the application, rather than just the single use that happens to have been picked for the reference workload.

6 Related Work

In [2], the analysis technique discussed in this paper is applied to SPEC CPU2000. Our paper builds on this research by applying the same techniques to the recently released SPEC CPU2006 suite.

In addition, to the techniques discussed in [2], there has been significant research focussed on feedback profile optimizations and workloads. This work can be viewed as three broad categories: firstly, the effectiveness of the training profile is investigated by examining the resulting per-

Floating point benchmarks	Coverage	Correspondance value
410.bwaves	100%	100%
416.gamess	73%	86%
433.milc	100%	100%
434.zeusmp	100%	100%
435.gromacs	100%	100%
436.cactusADM	100%	100%
437.leslie3d	98%	95%
444.namd	100%	100%
447.dealII	100%	100%
450.soplex	100%	99%
453.povray	100%	100%
454.calculix	89%	99%
459.GemsFDTD	100%	88%
465.tonto	100%	100%
470.lbm	100%	100%
481.wrf	67%	93%
482.sphinx3	100%	98%

Table 2. Results for the Floating Point benchmarks in CPU2006

Routine name	%total instruction count ref	%total instruction count train
advect_scalar_	26.64%	14.21%
wsm32d_	None	13.91%
advance_uv_	6.96%	6.49%
rtrn_	1.34%	5.95%
advance_w_	6.18%	5.73%
advance_mu_t_	4.34%	4.04%
advect_w_	4.23%	3.95%
advect_v_	4.04%	3.76%
advect_u_	4.03%	3.75%
calc_cq_	3.28%	2.32%
ysu2d_	2.94%	3.14%
clphy1d_	2.73%	None
rk_update_scalar_	2.71%	1.31%
curvature_	2.24%	2.15%
horizontal_pressure_gradient_	2.23%	2.13%
sint_	2.18%	None
calc_p_rho_	2.18%	2.08%

Table 3. Instruction count profile for 481.wrf

formance benefits [5] and the problems that can be caused when the functions are used in different ways by different workloads [6]. Secondly, research has been conducted into calculating the potential benefits of profile feedback for different applications [7]. Finally, there has been workload characterization, primarily to reduce the simulation space for processor design [8, 9, 10].

7 Conclusions

The methodology proposed in this paper enables the developer to make informed decisions about the quality of the training workload, and to improve the training workload as necessary. Unfortunately, there is the concern that many users will use imperfect training data when building their application. Hence the inclusion in the CPU2006 suite codes where the training data is less than perfect; some training workloads do not cover all of the code executed by the reference workload, and the branch behaviour for some training workloads is noticeably different from that of the reference workload.

Possibly as a result of the attention paid to selecting appropriate training workloads, there appear to be no benchmarks where the training workload is a very poor match for the reference workload. This can be contrasted with the situation for CPU2000 in which the benchmark 301.apsi has a very poor training workload that both misses critical basic blocks (it has a coverage of only 37%), and also has very different branch behaviour (the branch correspondence value for the benchmark is 72%) [2].

In contrast, the benchmarks with the worst coverage values in CPU2006 are 481.wrf with 67%, and 416.gamess with 73%. As for branch correspondence values, 416.gamess also scores lowest with 86%.

A further observation is that relatively few of the workloads in CPU2006 needed to have their training data adjusted as the result of this analysis. One cautious conclusion that may be drawn from this is that for most of the applications in CPU2000 and CPU2006 the behaviour of the branches and the coverage of the code is largely the same regardless of the workload run. In essence, branch behaviour for the most frequently executed branches in an application, is largely independent of the workload that the application is running.

References

[1] "SPEC CPU2006 Benchmark Suite," *http://www.spec.org/cpu2006/*.

[2] D. Gove and L. Spracklen, "Evaluating Whether the Training Data Provided for Profile Feedback is a Realistic Control Flow for the Real Workload," in *Proc. SPEC Benchmark Workshop* *http://www.spec.org/workshops/2006/*.

[3] D. Gove, "Selecting Representative Training Workloads for Profile Feedback Through Coverage and Branch Analysis," *http://developers.sun.com/sunstudio/articles/coverage.html*.

[4] "Cool Tools - Binary Improvement Tool (BIT)," *http://cooltools.sunsource.net/bit/*.

[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] 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.

[9] 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.

[10] 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.