On the Predictability of Program Behavior Using Different Input Data Sets

Wei Chung Hsu, Howard Chen, Pen Chung Yew
Department of Computer Science
University of Minnesota

Dong-Yuan Chen
Microprocessor Research Labs
Intel

Abstract

Smaller input data sets such as the test and the train input sets are commonly used in simulation to estimate the impact of architecture/micro-architecture features on the performance of SPEC benchmarks. They are also used for profile feedback compiler optimizations.

In this paper, we examine the reliability of reduced input sets for performance simulation and profile feedback optimizations. We study the high level metrics such as IPC and procedure level profiles as well as lower level measurements such as execution paths exercised by various input sets on the SPEC2000int benchmark.

Our study indicates that the test input sets are not suitable to be used for simulation because they do not have an execution profile similar to the reference input runs. The train data set is better than the test data sets at maintaining similar profiles to the reference input set. However, the observed execution paths leading to cache misses are very different between using the smaller input sets and the reference input sets. For current profile based optimizations, the differences in quality of profiles may not have a significant impact on performance, as tested on the Itanium processor with Intel compiler. However, we believe the impact of profile quality will be greater for more aggressive profile guided optimizations, such as cache prefetching.

1 Introduction

The SPEC benchmark suite [Henn2000] is a collection of CPU-intensive application programs. It has been widely used in the research community to evaluate architecture and micro-architecture designs and compiler optimizations. From SPEC89 to SPEC2000, the number of benchmarks and the average execution time of each benchmark program have continuously been increased. On average, each SPEC92int program executes about 1.3 billion instructions [Yung96] while this number increased to 64 billion for SPEC95int programs. In SPEC2000int, the average number of dynamic instructions executed reached a few hundred billion instructions. With significantly increasing execution times, and with more complex architecture/micro-architecture features to simulate, it is becoming increasingly difficult to simulate the complete SPEC benchmark suite. As a result, a common practice in the research community is to apply techniques to a small snapshot of the execution trace, for example, the first 100 to 500 million instructions of the trace. Another common practice is to use smaller "test" or "train" input data sets to reduce simulation time[1]. In addition to the *reference* input sets, which give the complete run of each program, SPEC also provides the *test* data sets which give a quick test of the benchmark, and the *train* data sets which allow the compiler to generate training profiles used for PBO (Profile-Based Optimization).

With the execution of a few hundred billion instructions in each program, the first 100 million instructions contribute about 0.1% of the total runtime, and are likely to perform initializations instead of accurately representing typical program behavior. To avoid capturing non-representative initialization behavior, some researchers wait until the initializations are complete begin detailed simulation. However, this approach does not guarantee a representative snapshot of the program's behavior, because some programs exhibit different execution phases, exercising completely different code and data behavior when it shifts from one phase to another. To accurately represent the execution of a benchmark program with multiple phases, at least one trace snapshot needs to be captured for each phase.

Using reduced data sets may be more attractive than studying a snapshot of the reference set,

[1] A survey of recent research publications shows that more than 60% of studies used reduced data sets.

because the smaller input sets may represent execution behavior similar to reference input sets. Since the test data sets and training data sets have shorter overall execution times than reference input, a large amount of research has been conducted using these smaller data sets in their simulations to conduct faster performance evaluations. However, since the test and the train data sets were not originally designed to serve as reduced data sets for the reference input, they may exercise different execution paths in the programs than the reference input sets. If this is indeed the case, the performance evaluation conducted based on such input sets could be misleading. For example, if the complete run with reference input would cause significant I-cache misses and D-cache misses, but the run with test input incurs no cache misses, the evaluation results based on the test runs would be very misleading.

In 1992, Fisher and Freudenberge [Fish92] reported that branch instructions could be predicted statically by using previous runs of a program. This provides evidence to support Profile Based Optimizations (PBO). Starting in SPEC92, training input sets have been provided by SPEC for compilers to generate execution profiles and perform profile directed optimization. The success of using small data sets to predict branch directions for future runs suggests that test or training input sets are capable of predicting the program behavior for the reference runs. However, some recent studies [Cohn98] on post-link time optimizations report that an application may exercise different code when different users use the application. This observation is particularly common for general-purpose applications that are rich in features. Profiled based optimization has also advanced beyond static branch prediction. For example, some commercial compilers [Ayer98] have been using profiles to determine which procedures to optimize, which execution paths get a high priority on resource allocation [Holl96], and which region to allocate more optimization time. Furthermore, recent research suggests using path profiling for trace cache allocation [Rami99], using value profiling for value prediction optimization [Cald99], and using cache profiling for data layout optimization [Cald98]. It is therefore important to understand to what extent we may use one input data set to predict the program behavior of future runs.

In this paper, we evaluate how reliably small input sets can be used in place of more time-consuming reference input sets. For some benchmark programs, small input sets exhibit the same execution behavior as the reference inputs, and the research community can comfortably use them to reduce simulation time. However, some programs do not have train or test input sets that are representative of their reference input set. We first examine the similarity of program behavior using high-level information such as execution profiles and IPC numbers. We then go into low-level analysis to investigate the frequent execution paths covered by each input data sets. Since the small and "light" input data sets generally do not stress the data cache as much as the reference input data set does, we also investigate whether different "heavy" input sets stress the data cache in a similar way. In other words, we would like to know how accurately and reliably we can use one input sets to predict the data cache behavior of a different set.

This study has two goals. One goal is to provide the research community some guidelines on using smaller input sets in reducing simulation time for SPEC benchmarks without giving misleading performance results. The second goal is to examine program behavior under different input sets. The key question is whether the smaller data set exercises the same execution paths and exhibits the same behavior as the reference input sets do? If not, we may not use the simulation results from smaller input sets to indicate the performance impact of the Spec2000 benchmark. Also, we evaluate the performance impact of using different input sets on the Itanium processor using the Intel compiler.

The remainder of this paper is organized as follows. In Section 2, we look at the high-level measures of execution profiles of Spec2000 programs using different input sets. In Section 3, we describe how to use the branch trace buffer feature in the Itanium processor to look into frequently executed paths exercised by different input data sets. Section 4 compares the frequent execution paths sampled by running different

Table 1. Execution time distribution of 181.mcf with different input sets

Procedure Name	Ref Input	Train Input	Test Input
price_out_impl	50.29%	31.06%	3.49%
refresh_potential	37.54%	39.24%	8.72%
primal_bea_mpp	8.47%	19.14%	54.65%
replace_weaker_arc	1.09%	1.99%	0.00%
sort_basket	0.76%	2.57%	18.02%

Table 2. Procedure coverage from one input set to the other

Gprof	Train vs Ref			Test vs Ref			Test vs Train		
	50%	80%	90%	50%	80%	90%	50%	80%	90%
Gzip	71.00	87.18	93.30	71.00	85.83	91.95	51.19	72.97	86.00
Vpr	29.65	73.61	83.58	55.19	78.20	86.06	59.75	84.55	92.31
GCC	64.45	84.86	91.04	66.15	85.24	90.82	55.89	76.91	86.99
MCF	37.64	88.06	96.56	8.49	9.26	97.74	19.30	21.89	92.79
Crafty	42.20	67.88	78.38	41.58	67.36	77.77	48.72	78.22	88.33
Parser	47.56	80.41	88.76	28.05	62.11	73.45	36.40	66.70	76.03
Eon	41.57	45.28	47.25	19.99	41.57	41.57	20.51	56.89	64.30
Perl	25.37	29.71	33.10	0.00	1.28	1.28	0.00	7.27	7.27
Gap	48.63	85.02	95.39	44.38	58.54	65.84	43.13	63.65	70.80
Vortex	33.86	53.56	68.87	37.85	65.80	71.58	48.50	73.50	87.99
Bzip2	5.57	49.29	59.03	27.27	53.45	67.93	24.56	40.79	93.34
Twolf	46.43	79.53	90.49	44.40	74.25	83.85	49.29	74.70	84.66

input data sets. Section 5 compares the execution paths for frequent data cache misses since many Spec2000int programs exhibit a high data cache miss rate. We evaluate the impact of different profiles on PBO performance in section 6, and the summary and conclusion are given in Section 7.

2 Profile Comparisons

2.1 Execution Profile Comparison

We first examine the high-level performance characteristics of each benchmark program. This includes the gprof [Grah82] profiling and IPC information. In this study, we compile SPEC2000int benchmarks for a Pentium-III processor running on the Linux at O3 optimization level. Table 1 shows the execution time distribution from gprof of program 181.mcf. With the reference input set, the mcf benchmark spends 50% of time in procedure *price_out_impl*, 37.5% of time on procedure *refresh_potential*. When the train input is used, they are also the top two procedures in the profile. However, procedure *refresh_potential* now becomes the number one routine, while procedure *price_out_impl* reduces its execution time contribution from 50% to 31%. When the test input is used, the profile becomes very different. Now the top two procedures, *price_out_impl* and *refresh_potential* are insignificant, while procedure *primal_bea_mpp* and *sort_basket* became the top ones.

When a reduced input data set is used, we would like to know whether it covers the important part of the program for the reference runs. In Table 2, we try to correlate procedure profiles among different input data sets. For example, in the first column, we compare profiles of train input to the reference input. In the column labeled as 50%, we take the top procedures accounting for 50% of runtime cumulatively from the train input run, and give the percent of runtime these procedures cover in the reference input run. As shown in Table 2, Test input sets do not cover procedures very well for the reference run of Mcf, Eon, Perl, Gap, and Bzip. The procedures accounting for 80% of execution time of test input runs cover only 9.26%, 41.57%, 1.28%, 58.54%, and 53.45% of the reference run, respectively.

In general, train input runs have good procedure coverages. For Gzip, Vpr, Gcc, Mcf, Parser, Gap, and Twolf, the procedures accounting for 80% of execution time of train input runs cover similar execution percentages for reference input runs; in Perl, Eon, and Bzip the coverage is less than 50%. The compiler must be careful when training profiles are used to determine which procedures to optimize for these three programs. For example, Perl spends about 20% of time on procedure *regmatch*, but this procedure does not even show up in the gprof result for the training run. Therefore, using the training profile, the compiler may decide not to optimize the *regmatch* procedure.

2.2 IPC comparison

In this section, we measure the IPC (Instruction Per Cycle) for each benchmark program using all three different input data sets. Several SPEC2000int programs, such as gzip, vpr, mcf, bzip and twolf, spend 90% of execution time on a very small number of procedures (less than 10), so the relative procedure coverage reported in the previous section is very high. However, it is not clear whether the execution behavior inside each procedure is similar under different input data sets. In this section, we examine the IPC numbers and in the next two sections, we sample the execution paths for a more detailed comparison.

We use hardware performance counters to report IPC numbers. The same set of benchmarks is compiled for Itanium using a beta version of the Electron compiler from Intel at O2 optimization level. For programs that have multiple reference input files, we average the IPC for each individual runs.

Figure 1 shows that the IPC numbers of vpr, mcf, perlmark, and twolf change significantly from one input set to the other. Consider vpr, for example, the IPC for the test run is about two times the IPC of the reference input. Mcf has an IPC number using test run more than three times the IPC using the reference input. In these benchmarks, different input sets exposed very different performance characteristics. For programs that have similar IPC numbers, such as gcc, gap, and gzip, it is not guaranteed that different input sets for such programs exercise the same execution paths and exhibit the same cache behavior. We will examine the sampled execution paths of each program to verify their behavior in following sections.

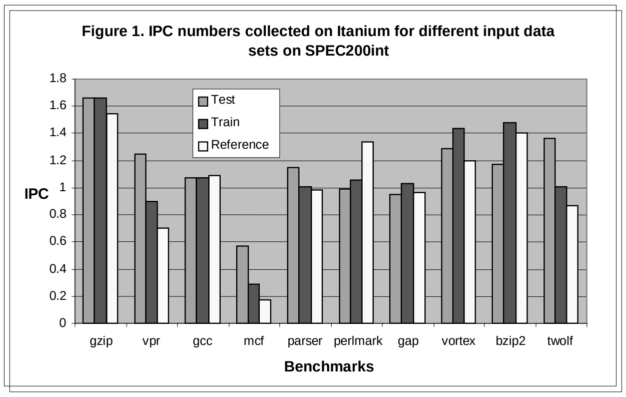


Figure 1. IPC numbers collected on Itanium for different input data sets on SPEC200int

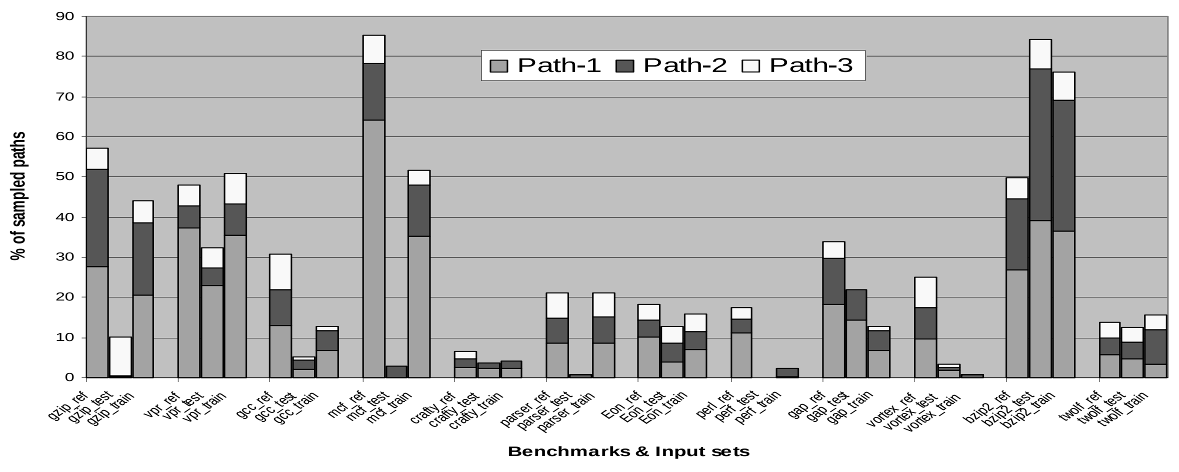
3 Using Branch Trace Buffer to examine frequently executed paths

The Itanium processor defines and supports a rich set of performance monitoring features that can be used to characterize workload and to profile application execution [Itan00a, Itan00b]. Itanium has four performance-counter registers that can be programmed to measure stall cycles in eight different categories as well as to count occurrences of over a hundred events. Furthermore, Itanium supports Event Address Registers (EARs) for both Data and Instruction events as well as an eight-entry Branch Trace Buffer (BTB). The Instruction EAR can capture the addresses of instructions that trigger I-cache or ITLB misses. The Data EAR can capture the addresses of load instructions that cause D-cache or DTLB misses together with the target addresses of these loads. The Branch Trace Buffer is an 8-entry circular buffer that can capture information on the most recent branch instructions and their outcomes. These performance-monitoring features enable us to gain a detailed understanding of the dynamic execution behavior of the running application. Since each retired branch instruction that is recorded in the Branch Trace buffer may take up to two entries, one entry for the address of the branch instruction and another entry for the address of the branch target, we can program the Branch Trace Buffer to capture the most recent four taken branch instructions for each sample.

A profiling tool that utilizes the performance monitoring features, called Itanium Profiling Tool (or IPT), has been developed on Itanium running 64-bit Linux in the MRL of Intel. IPT required a customized performance monitor device driver (PMU driver) that runs as part of the Linux. IPT supports various modes of profiling on a running application, including the measurement of stall accounting, the counting of all performance events supported by the Itanium processors, and the collection of samples on various events. The IPT program interacts with the PMU driver to configure the performance monitoring registers and to receive profiling or sampling data from the PMU driver and store them in a profiling file.

To sample execution paths for this study, we used IPT to collect branch trace information for SPEC2000 integer benchmarks. We configure the Branch Trace Buffer to capture only the taken branches regardless of their branch prediction outcomes. While running the integer benchmarks with reference input, one branch trace sample was taken every one million cycles and every ten thousand D1 cache misses. For test and training input, one branch trace sample was taken for

Figure 2. Comparison of frequent execution paths using different input sets



every ten thousands clock cycles and every one hundred D1 misses. This is because we try to maintain roughly the same number of samples between reference, test, and training runs. The sampling rates used is faster than the sampling rate usually used by gprof in all cases. Although a faster sample rate will obtain more unique execution paths, the most frequently sampled execution paths of each program remain the same as with the slower sampling rate. Since this study focus on the most frequently executed paths, we do not collect data on various sampling rates. Each branch trace sample was captured by the IPT program and stored to disk for offline processing. Offline, all branch trace samples were sorted to count the number of times each branch trace path was executed.

4 Execution Path Analysis for Different Input Sets

As we stated earlier, even if a program has similar procedure coverage and IPC numbers for different input data sets, the execution paths exercised by the different input sets may be different. If different execution paths are exercised under different input sets, using one input set may not reliably predict the performance of other runs. For the same reason, aggressive PBO based on one input set may not be effective for other runs.

We use the IPT tool described in Section 3 to study the frequent execution paths for each program under different input sets. We first select the top three frequently executed paths from the reference input runs. For each path, we report its percentage in the total sampled paths. For example, as shown in Figure 2, the number 1 path of Gzip accounts for 27.69% of total sampled paths. We also report their respective percentage for test and train input runs. The number 1 path selected from the reference run of Gzip accounts for only 0.26% from all the sampled paths for the test run, and 20.71% for the train input.

Figure 2 shows that the top three paths of Gcc using reference input account for about 30% of execution time. This seems to contradict with the common sense that Gcc tends to have a very flat profile. The hot execution paths come from the memcpy and the memset library routines. These two routines also account for 30% of execution time on both Pentium-III based and Sun Ultra SparcIIe based systems, using gprof with reference input.

In Figure 2, we can see that some important execution paths for the reference runs are insignificant for the train or the test runs. From the high-level comparisons in Section 2, we may believe Crafty, Gcc and Gap can reliably take advantage of reduced input data sets. However, Figure 2 shows that there are substantial variations on the relative importance of the frequently executed paths for these three programs.

Readers may wonder how important the relative order and their respective weight for those frequently executed paths are. For example, if for training input, path-1 accounts for 40%, and path-2 accounts for 10% of the execution time, and if the distribution becomes path-1, 10% and path-2, 40% for the reference input, would it make a big difference in PBO optimization? The answer depends on how the profile is actually used in optimization. If the optimizer uses the profiles to select all important paths, the relative order may not matter much. Eventually, both path-1 and path-2 will be selected and optimized. However, if PBO takes weight into account, it may decide to optimize only the number 1 path (due to compile time consideration), the outcome could be very different. Furthermore, when using small input sets for performance projection in simulations, the relative weight of different execution paths do make a difference.

The compiler may choose to take the top 80% of execution paths of the profile as optimization candidates. We are interested to know how much execution time these selected candidates may cover the run time of the full reference input run. Table 3 shows the possible coverage. For example, if we take the top 90% (accumulative) of execution paths from the profile collected with the Test input on Gzip, these paths may cover only 28.5% of the run time for the reference input run. In Table 3, we can see more than half of the benchmarks have very poor coverage if Test profile is used. In general, profiles using train input sets have better coverage than the test input sets.

Table 3. Coverage of execution paths using one input run to predict the other

CPU	Train Vs Ref			Test Vs Ref			Test Vs Train		
	50%	80%	90%	50%	80%	90%	50%	80%	90%
Gzip	59.55	76.82	89.32	16.21	22.77	28.05	28.11	39.28	44.08
Vpr	48.01	64.89	69.89	47.01	80.80	92.06	65.70	91.10	96.09
GCC	69.69	85.99	91.32	76.65	88.70	91.59	64.46	84.08	88.77
MCF	66.16	91.72	96.59	2.83	18.79	18.86	9.11	26.18	26.69
Crafty	34.17	51.70	59.33	26.45	38.55	44.38	36.74	54.85	60.03
Parser	50.93	81.19	90.32	27.79	76.62	89.39	26.20	77.57	89.53
Eon	47.82	76.93	88.82	48.84	76.76	84.97	51.83	81.15	89.61
Perl	47.25	68.32	76.14	-	-	-	-	-	-
Gap	34.96	67.09	75.70	34.56	42.78	52.76	39.93	73.32	82.84
Vortex	41.40	80.61	89.03	63.21	84.17	91.66	44.99	75.00	87.03
Bzip2	26.99	52.24	72.07	26.99	44.59	56.58	36.41	69.16	82.02
Twolf	45.19	79.34	90.44	42.07	71.71	79.25	45.03	71.58	77.64

5 Comparison of Frequent Execution Paths for Data Cache misses

5.1 Path coverage analysis

Figure 3 compares frequent paths leading to data cache misses. For programs without many data cache misses, it is not important to study such paths. However, since many SPEC2000int programs have a high D1-cache miss rate running on Itanium, it is important to understand whether such paths can be predicted using profiles generated from smaller input data sets. Figure 3 does not contain all the programs-- some programs with insufficient data cache miss samples were not included.

Figure 3 shows variations of such execution paths are far greater than the variations in Figure 2. For example, the path that accounts for the highest data cache misses in Vortex (responsible for 61.85% of D1-cache misses) does not even show up in the train input run (it covers 0.0% of sampled execution paths for data cache misses). Figure 3 shows that the test input is almost useless in predicting frequent D-cache miss paths except for Crafty. The train input can be used to predict data cache miss paths for Vpr and Parser. It may also capture frequent data cache miss paths for Gcc, Mcf, with substantially different weights on the paths. Train inputs predict data cache miss paths poorly for Gap, Vortex and Bzip2.

Figure 3. Comparison of frequent execution paths leading to data cache misses using different input sets

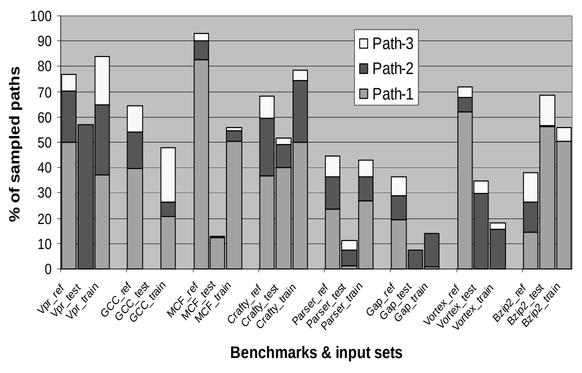


Table 4 shows the similarity of the top branch paths leading up to D-cache misses for the different input sets. The execution path coverage in Table 4 is in general lower than the coverage in Table 3. In particular, if the threshold is 50%, the prediction for reference run can be very poor. Table 4 shows test data sets can capture many frequent execution paths leading to data cache misses for Parser. However, from Figure 3, we have observed that the top three paths for data cache misses in Parser do not stand out during test data set runs. This shows a difference of using test data sets for PBO and for reducing simulation time. If the PBO compiler takes 90% of observed paths from one run to optimize for the other run, the relative weights of each path become less important. As long as the frequent executed paths are optimized, PBO has achieved its goal. However, the relative weights and order of such paths are important when simulation time is considered.

Table 4. Coverage of execution paths leading to data cache misses using one input run to predict the other

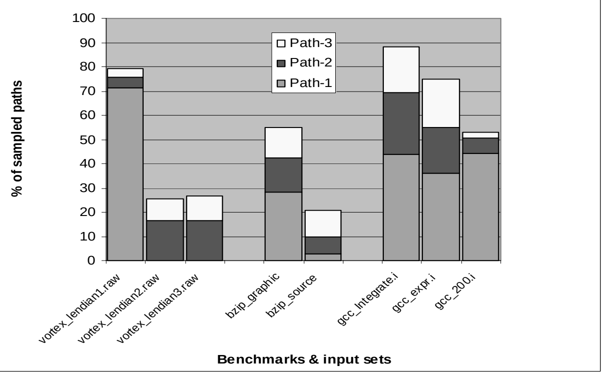
D1	Train Vs Ref			Test Vs Ref			Test Vs Train		
	50%	80%	90%	50%	80%	90%	50%	80%	90%
Vpr	37.02	76.93	78.17	20.23	20.23	21.47	27.85	27.85	35.92
GCC	15.06	73.92	92.12	0.00	0.00	0.00	0.02	0.06	0.07
MCF	82.70	85.33	93.51	2.18	85.44	86.07	25.83	84.12	88.29
Crafty	36.72	66.81	79.64	36.72	59.29	68.14	40.09	74.36	83.36
Parser	54.07	82.56	91.85	60.83	81.10	88.29	58.38	81.37	88.37
Gap	17.96	55.21	70.06	16.35	20.80	43.53	38.01	71.69	81.24
Vortex	16.95	25.37	28.61	13.84	24.18	30.88	35.07	71.84	83.78
Bzip2	14.44	31.86	51.97	14.44	29.97	35.84	50.21	65.77	73.25

5.2 Small Vs large input data sets

From Table 4, we might conclude that for data cache profiling, using small input data sets may misrepresent the projected performance. It seems like the compiler should avoid using small input data sets to collect data cache miss profiles because reduced memory accesses are less likely to generate frequent data cache misses. However, the remaining questions are a) is it practical to use large input data sets to collect profiles for PBO, and b) Even if a large input data set is used for profiling, can it reliably identify execution paths to the data cache misses for the other input set. For question (a), we shall leave it to software vendors to decide how much profiling overhead they can tolerate. For question (b), we looked at the predictability of using one reference input to predict for future runs with different input sets. Note that in this case both input data sets are from reference sets, not from the small data sets.

For those programs that incur frequent data cache misses on the Itanium and have multiple reference input files, we compare their most frequent paths leading to data cache misses in Figure 4. It shows that even if the full reference input is used to collect data cache miss path profiles, the variation is still large from one input to another. The number one execution path to data cache misses in Vortex account for 71.5% of all sampled paths when the input lendian1.raw is used. However, this path does not appear when input lendian2.raw and lendian3.raw are used. On the other hand, a path that accounts for 16% of the sampled paths for input lendian2.raw and lendian3.raw, contributes only 4% when lendian1.raw is used. Similar results can be found for Bzip2 and Gcc. Using different input sets, no matter whether they are reduced size or regular size, may not reliably predict the paths leading to data cache misses for Spec2000int benchmarks.

Figure 4. Frequent execution paths to data cache misses comparison using different reference input files.



6 Performance Impact of Different Profiles on Profile Based Optimization

Although our study indicates one input set does not always accurately predict the program behavior for another, it is not necessarily a problem in PBO because a) the compiler may select a set of inputs with different behaviors to generate profiles; b) some PBO transformations are less aggressive so that they depend less on the execution path or memory access behavior. In this section, we evaluate the performance impact of

PBO based on profiles generated from the test, the train and the reference input sets. The experiment was conducted on the Itanium processor, where PBO is regarded as very important.

We compiled Spec2000int programs, using the Intel C/C++ compiler, on the RedHat 7.1 Linux. We used the performance of programs compiled at O2 as the base. We then compiled our benchmarks using IPO (Inter-Procedural Optimization) and PGO (Profile Guided Optimization, in Intel's term). Note that PGO is the same as PBO, so we call it PBO here. When compiled with IPO/PBO, we use profiles collected from test, train and reference input sets. The performance relative to the base performance is reported in Table 5. All performance reported in Table 5 are relative to the base performance. As shown in Table 5, Gzip, Gap, Vortex, Bzip and Twolf can benefit from train profiles. This is no surprise; because Table 3 shows that train profiles cover the runtime of reference input better than test profiles on the aforementioned programs. Table 3 also shows that using profiles collected from reference inputs in PBO does not increase performance much. One thing worth noting is that the test profile of Mcf does not represent reference input at all. However, there is no performance difference for Mcf when more accurate profiles are used. This is because the performance of Mcf is dominated by several link-list chasing loops that have intensive data cache misses. Several existing effective PBO transformations would not improve those loops. However, if cache profile guided prefetching is implemented, using train profiles may expose such optimization opportunities.

Table 5 shows PBO can benefit from better profiles. The performance gain from using better profiles is not very significant, except for Gzip, which could gain 10% of performance if the train profile is used instead of the test profile. The performance impact of different profiles is not as significant as we expected. This is because Vpr, Mcf, Parser, Gap, Vortex and Bzip suffer significantly from frequent data cache misses on Itanium. When the performance is dominated by data cache misses, non-cache related PBO would not change performance much. When cache profile guided optimizations are adopted by compilers, different profiles may have a higher impact on performance.

Table 5. Performance of PBO on Itanium using different profiles

Program	*IPO*	*IPO+PBO (test)*	*IPO+PBO (train)*	*IPO+PBO (reference)*
164.gzip	1.07	1.19	1.31	1.29
175.vpr	1.15	1.19	1.19	1.19
181.mcf	1.03	1.04	1.03	1.03
186.crafty	1.25	1.29	1.3	1.32
197.parser	1.08	1.11	1.11	1.11
254.gap	1.08	1.2	1.25	1.28
255.vortex	1.1	1.3	1.36	1.35
256.bzip	1.18	1.15	1.17	1.18
300.twolf	1.05	1.1	1.14	1.15
Average	1.11	1.17	1.21	1.21

7 Summary and Conclusion

It has been a common practice to use smaller input sets to estimate the performance of a benchmark or to generate profiles for PBO. In this paper, we look at how reliable this approach is. We have studied the high level metrics such as IPC and procedure level profiles and the low level measurement such as execution paths exercised by various input sets on SPEC2000int programs.

Our study indicates that the test input sets are not suitable to be used for simulation because they do not have an execution profile similar to the reference input runs. The train input is far better than the test data sets at maintaining similar profiles. However, there are significant differences between train profiles and reference profiles for Perl, Eon, Bzip2, and Vortex. We recommend cautiousness in using train input to project simulation performance for Vpr, Mcf, Gap, Gcc and Perl. We have observed significant variations in respective weights of those frequently executed paths using different input sets. Such relative weights could be critical when aggressive PBO is used. Profiles from train input could be reliable when predicting branch directions for other runs, but they could be misleading if the relative weights are used to guide optimizations.

A common practice has been adopted in PBO is to merge profiles from several different training input sets. However, most SPEC2000int programs have only one training input (only Perl and Eon have more than one training input files). In general, identifying representative small input sets for an application is not easy, even ISVs (Independent Software Vendors) have difficulties identifying representative sets. We have evaluated the impact of different profiles on PBO performance using the Itanium processor. While more accurate profiles lead to higher performance, the overall performance impact has not been shown to be very significant. Our study shows that smaller data sets do not predict frequent data cache miss paths in the reference input runs. We have also shown that data cache miss paths may not be predicted using a different reference input set. Since the profiled execution paths using small data sets often carry weights significantly different from paths in full runs, and since data cache miss paths are difficult to predict using different input sets, it would be a challenge to use profiles from small inputs to guide cache prefetching related optimizations.

8 References

[Ayer98] Andrew Ayers, Stuart deJong, John Peyton and Richard Schooler; "Scalable Cross-module Optimization", Proceedings of the ACM SIGPLAN '98 conference on Programming language design and implementation, 1998.

[Burg97] D. Burger, and T. Austin, "The SimpleScalar Tool Set, Version 2.0" Technical report 1342, Computer Sciences Department, University of Wisconsin Madison, June 1997.

[Cald99] B. Calder, P. Feller, and A. Eustace, "Value Profiling and Optimization", Jounal of Instruction Level Parallelism, March, 1999

[Cald98] B. Calder, C. Krintz, S. John, and T. Austin, "Cache-conscious data placement", In Proceedings of the the 8th nternational Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII) Oct. 1998.

[Cohn98] Robert S. Cohn, David W. Goodwin, P. Geoffrey Lowney, "Optimizing Alpha Executables on Windows NT with Spike", Digital Technical Journal, Vo l 9 No 4, June, 1998

[Fish92] J. A. Fisher, and S. Freudenberger, "Predicting Conditional Branch Directions From Previous Runs of a Program," Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, Oct., 1992

[Grah82] Graham, S. L., P. B. Kessler, M.K. McKusick, "gprof: A Call Graph Execution Profiler", Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, 1982.

[Henn2000] Henning, John L., "SPEC CPU2000: Measuring CPU Performance in the New Millennium," IEEE Computer, Vol. 33, No. 7, July 2000.

[Holl96] Anne M. Holler "Optimization for a Superscalar Out-of-Order Machine", Proceedings of the 29th annual IEEE/ACM international symposium on Microarchitecture, 1996.

[Itan00a] Intel Itanium Architecture Software Developer's Manual Vol. 4 rev.1.1: Itanium Processor Programmer's Guide. Intel Corp. July 2000.

[Itan00b] Intel Itanium Architecture Software Developer's Manual: Specification Update. Intel Corp. August 2001.

[Rami99] Alex Ramirez, Josep-L. Larriba-Pey, Carlos Navarro, Josep Torrellas, and Mateo Valero, "Software Trace Cache", 1999 ACM International Conference on Supercomputing (ICS), June 1999.

[Yung96] Robert Yung, "Design Decisions Influencing the UltraSPARC Instruction Fetch Architecture," Proceedings of the 29th annual IEEE/ACM international symposium on Microarchitecture, 1996