Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 04-033

Using Large Input Sets with Hardware Performance Monitoring for Profile Based Compiler Optimizations

Sagar Dalvi, Wei-chung Hsu, and Pen-chung Yew

September 17, 2004

Using Large Input Sets with Hardware Performance Monitoring for Profile Based Compiler Optimizations

Sagar Dalvi, Wei-Chung Hsu, Pen-Chung Yew
(sagar,hsu,yew@cs.umn.edu)

Abstract

Traditional Profile Guided Optimization (PGO) uses program instrumentation with one or more small training input data sets to generate edge or value profiles to guide compiler optimizations. This approach has been effective in predicting branch directions for many applications. However, for optimizations that are more dependent on the performance characteristics and the accuracy of the profiles, it is not clear whether profiles generated with small input data sets can reliably predict the program behavior under different input sets.

We studied the frequent execution paths, data cache miss paths, IPC and the stall cycles breakdown of the test, train and different reference input sets of the SPEC2000Int benchmarks. Our studies indicate that small input sets are less effective in predicting the program behavior for larger input data sets. We propose to use Hardware Performance Monitor (HPM) sampling based profiles to guide optimizations, because it can work with larger input sets and gather information on important performance events.

As a proof of concept, we have implemented one type of HPM sampling based PGO. We use the dynamic call path sampled by HPM to automatically guide procedure inlining in the ORC Compiler. Our results show that this approach has much lower profiling overhead, and offers significant performance gains.

1 Introduction

Profile Based Optimization (PBO) has been widely adopted for code layout transformations to increase code locality [Pett90, McFa03, and Rami99]. Modern processors have also relied on PBO to more effectively exploit instruction level parallelism [Itan03]. Many recent research papers have focused on using PBO in optimizations to improve power consumption and memory performance [Kris02, Chil02]. Traditional PBO is mostly instrumentation based. The program is

first instrumented with a compiler to collect edge, value, or memory reference profiles. Instrumentation based profiling has a relatively high profiling overhead, requires one extra compilation pass, and is intrusive in the sense that the performance characteristics of the program may change.

The large overhead of profile generation restricts it to using small input sets like the train data set provided by the SPEC [Henn2000] benchmark suite. Substantial effort has been devoted by some benchmark sponsors to ensure the train input set of the SPEC benchmarks can reasonably represent the longer reference (ref) input set. However, in practice, it is usually difficult for a software developer to come up with reduced input sets that can accurately represent the diversified use of the application by different customers.

Much research effort has been focused on addressing the problem of coming up with representative profiles for the ref input set and analyzing its validity for subsequent runs. [Wall00] shows that profiles from real runs can predict program behavior better than estimated profiles, but not as well as a perfect profile from the same run being measured. Some studies [Cohn97] on post-link time optimizations report that an application exercises different code when different users use the application, particularly for feature rich applications. User site PBO [Cohn97] accounts for the varying usage patterns by collecting multiple training inputs at user site for PBO. Fisher and Freundenberge [Fish92] report that branch instructions can be predicted statically by using previous runs of a program. This provides evidence to support PBO. The Reality Based Optimization (RBO) [McFa03] approach uses profiles from a large number of larger input sets to improve code layout; since smaller input sets are not suitable for driving PBO in real-world applications. RBO found that most of the common behavior can be captured using few profiles and any incremental addition would require larger number of profiles. Both the PBO and RBO approaches have advantages and disadvantages vis-à-vis the accuracy and profile generation time.

Instrumentation based profiling may alter the performance characteristics of the program being profiled. For example, if the performance bottleneck of the target application is I-cache misses, the instrumented binary may further increase I-cache misses and making the identification of hot spots of frequent I-cache miss more difficult. In addition to the profiling overhead and intrusiveness, instrumentation based profiling makes the PBO process less transparent. The extra compilation to instrument the code and the size of the inflated binary often discourage software developers from including PBO optimization in their binary building process.

Conte [Cont96] demonstrates the use of sampling to effectively capture branch behavior with low overhead. Similarly, sampling the HPM allows us to implement a profiling approach that can capture profile for larger input sets with very low overhead. If necessary, HPM sampling based profiling can also be used with multiple small input sets to achieve a higher coverage with low profiling overhead. Further, HPM sampling based profiling can capture important performance events, such as the cache or TLB misses. By configuring the HPM to monitor taken branches, it is possible to identify a trace of the execution path executed for each sample. [Ammo97] shows the advantage of flow sensitive path profiling. [Haze03] applies context information to make better inlining decisions. Although current compilers may not use this kind of performance information in optimizations, many recent research compilers have started exploiting performance critical events guided optimizations [Coll01]. For example, data speculation check failures reported by HPM on Itanium processors could be used to guide more effective data speculation transformations in the compiler.

We believe HPM based sampling is more suited to generate profiles using multiple and larger input sets. Profiles generated from multiple and larger input sets can more accurately predict the program behavior under diversified usage. This profiling approach collects various important performance events and characteristics not accessible to instrumentation based profiling. It also avoids the extra compilation step of generating an instrumented binary, making the profiling process more transparent. As a proof of concept, we experimented with HPM profile

based procedure inlining [Chan92]. This approach shows an increased performance at a lower profiling overhead. The rest of this paper is organized as follows. Section 2 explains HPM based sampling along with the experiment setup. Section 3 shows that larger input sets capture program behavior more accurately than smaller input sets. Section 4 explores the feasibility of using one ref input to predict the behavior of other ref inputs. Section 5 explains the HPM feedback mechanism to guide the procedure inlining optimization. Section 6 concludes the discussion.

2 Experiment Setup

The SPEC2000 INT benchmark suite running on a 900Mhz Itanium 2 system (IA-64 ISA) is used in this study. We consider those benchmarks, which have multiple reference inputs in order to show the differences between the different reference inputs themselves. The benchmarks were compiled using Intel's ECC compiler [ECC02] and Open Research Community's ORC compiler [ORC02]. IA-64 systems provide support for performance monitoring through a set of dedicated programmable registers. The Pfmon [Pfmo02] user tool harnesses the power of these registers by providing a configurable and easy to use interface to the IA-64's performance monitoring features.

At the heart of the IA-64's performance monitoring support is the eight-entry rollover Branch Trace Buffer (BTB). As the name suggests, it stores the last eight branches encountered before a performance event (the event being monitored). The BTB can be configured to store sets of source-target pairs and can be further restricted to monitor only taken/ not-taken branches or a particular type of branch (direct, indirect etc). By sampling the BTB periodically we can get a reasonably accurate view of the runtime program behavior. It is also possible to monitor specific performance critical events like data cache misses or instruction cache misses. By combining the sampling with cache miss monitoring we can get a trace (execution path) leading to the cache miss. For this study we have used BTB sampling to study the performance of various benchmarks. For most of our work, it is enough to differentiate among the individual reference

inputs using numbers rather than names. Therefore we represent reference input "x" as "ref.x", where x is a integer.

3 Small vs. Large input sets

Small inputs like test and train are used as representative data sets to mimic the behavior of the larger reference (ref) inputs. The much longer running time for simulations using ref inputs necessitates the use of smaller inputs. Using the larger input set with instrumented binaries to generate profile information bloats the size of the internal profile data structures and increases the running time. Therefore, large input sets are not used to generate profiles needed for PBO. In this section we observe the trace coverage and detected top paths using the test, train and ref inputs for each of the benchmarks under consideration. The objective is to show that smaller input sets cannot accurately represent the ref input set.

3.1 Comparing Small input sets vs. Combined ref input set

Frequent execution paths for test, train and reference inputs were found by sampling the BTB. Table 1 shows the percentage of the ref input covered by traces accounting for 50, 80 and 90% of the smaller inputs. We use the term "coverage" as a measure of how faithfully one input represents another. To calculate coverage of input X with reference to input Y, we sort the traces in X according to their percentage contributions and note the percentage that it would account for in the input Y. We simply accumulate the percentages of traces in X till we cover 50% and note the corresponding accumulated percentage for Y. Thus each cell in Table 1 shows how many percent traces in the ref input are covered by 50% traces of the test or train input. For obtaining 80 and 90% coverage, we simply keep accumulating till the corresponding mark.

From Table 1 we can see that, in general, the test input has lesser coverage than the train input, especially for perl. As coverage of ref input increases, the test and train coverage does not rise proportionately (For example, the train input of gcc covers 37.01% of execution time at 50%

Benchmarks & Input sets		ref		
		50%	80%	90%
vpr	test	36.59	50.51	53.24
	train	34.87	54.19	60.02
bzip2	test	16.61	46.48	58.29
	train	22.30	55.15	66.28
vortex	test	42.56	47.60	49.30
	train	43.86	50.817	52.56
gzip	test	34.62	42.73	44.20
	train	48.83	69.87	77.27
gcc	test	28.03	30.23	30.89
	train	37.01	39.77	40.42
perlbmk	test	9.08	9.09	9.09
	train	33.79	36.17	37.16

Table 1

mark but just 40.42% at 90% mark). This is due to the comparatively shorter execution time, which limits the number of unique execution paths that can be exercised.

The basic principle of optimization is to optimize for the most frequent cases. We studied the top 10 execution paths detected by test and train inputs and found that for bzip2, gcc, vortex, perlbmk and vpr, the smaller inputs could identify the frequently executed paths. However, they could not accurately gauge the percentage of execution time that these paths covered. The test input of gzip could not identify the two most frequently executing paths accounting for nearly 24% of the total execution time.

3.2 Comparing Small input sets vs. Individual ref input sets

From the above data, one can reason that though the train input does not provide high coverage, it does identify the most frequently executed paths and is suitable for PBO. However, the ref data set used above was a combination of all the different ref sets for each input. The individual reference sets are supposed to represent the diverse ways in which the program may be used. Therefore, it would make more sense to compare the test and train inputs against the various reference inputs. Table 2 shows the coverage of each reference input for 50% execution time of the smaller input sets.

From the results shown in Table 1, we had concluded that the test and train inputs of vpr could be used to represent the ref input. However, from Table 2, we can see that the smaller inputs cover just one of the two ref inputs. For other benchmarks, the execution coverage varies widely. For example, gzip train input varies from 28% for ref.4 to 60% for ref.5. The blank entries in the table indicate that the corresponding reference input does not exist.

Benchmarks & Input sets		ref.1	ref.2	ref.3	ref.4	ref.5	ref.6
		50%	50%	50%	50%	50%	50%
vpr	test	55.60	0				
	train	52.97	0				
bzip2	test	21.53	4.29	20.90			
	train	27.24	10.84	25.98			
vortex	test	44.27	38.29	44.80			
	train	43.54	43.39	43.69			
gzip	test	33.53	28.82	31.38	35.57	37.28	
	train	50.29	37.40	28.70	28.71	60.49	
gcc	test	41.05	25.41	26.39	37.93	22.63	
	train	52.60	33.99	35.11	48.39	30.29	
perlbmk	test	0	0	0	0	0	0
	train	38.30	38.02	32.56	36.34	33.20	13.12

Table 2

Bzip2 and Gcc train inputs can detect most of the frequent paths for each of the ref inputs though their percentages may vary. For Gzip, the test input doesn't show top two paths of ref.1 and ref.2. Even the train input does not detect some important paths. For example, a path accounting for 17% execution of ref.4 is not detected as a top10 path. The train input for perl, does not detect top paths accounting for around 4-5% for ref inputs ref.2 to ref.5. It cannot detect most of the top paths of ref.6. The test and train inputs of vortex have similar characteristics and can detect most paths fairly well. Similarly, the test and train inputs of vpr are quite good for the solitary reference input that they can predict.

Some PBO methodologies rely on using multiple smaller inputs, similar to train, in place of ref inputs to obtain a more representative profile. Table 3 shows the effect of combining two smaller inputs to predict the larger inputs and the individual train input. The results show that

using two smaller inputs does not provide any significant advantage over using a single train input.

Benchmarks & Input sets		train	ref.1	ref.2	ref.3	ref.4	ref.5	ref.6
		50%	50%	50%	50%	50%	50%	50%
Vpr	test + train	50.65	53.34	0				
Bzip2	test + train	47.85	27.24	10.84	25.98			
Vortex	test + train	44.99	47.58	46.11	47.67			
Gzip	test + train	50.44	50.31	37.44	29.03	29.29	60.50	
Gcc	test + train	52.58	53.63	35.10	36.48	49.20	31.40	
Perlbmk	test + train	39.41	38.28	38.79	33.25	37.01	33.94	12.59

Table 3

The widening gap between processor and memory speeds has increased the importance of data cache misses, especially since the IA-64 ISA provides instructions that can hide some of the latency. We studied the paths leading to data cache misses by configuring the HPM to monitor data cache miss events and the last four branches leading to it. The results are shown in Table 4.

Benchmarks & Input sets		ref.1	ref.2	ref.3	ref.4	ref.5	ref.6
		50%	50%	50%	50%	50%	50%
Vpr	test	46.30					
	train	57.18					
Bzip2	test	33.03	5.64	22.34			
	train	46.46	27.06	51.89			
Vortex	test	67.15	59.58	67.79			
	train	39.15	49.87	37.93			
Gzip	test	42.66	34.14	42.74	56.48	44.17	
	train	70.08	52.52	14.98	12.89	72.57	
Gcc	test	36.65	31.03	38.28	40.56	31.32	
	train	47.13	35.31	42.59	50.07	35.55	
Perlbmk	test	0.01	0.01	0.03	0.02	0.01	0.00
	train	48.95	66.71	42.56	48.01	51.41	16.54

Table 4

The data cache miss path coverage shows wide variation for all benchmarks. For example, 14.98% for gzip ref.3 to 70.08% for gzip ref.1. In many cases, 50% of the test input can predict more than 50% of the data cache miss paths of ref input. The better coverage is due to the smaller number of data cache miss paths compared to the execution paths. The data cache misses are

closely related to the cache hierarchy of the host machine. Therefore, it is difficult to make train inputs that would accurately match the data cache miss paths of the ref input on different host machines.

In Summary, for execution paths, the test input is not good for predicting the ref input. Train is a good compromise but is not reliable in some cases, for example vpr. Also, the percentage contribution of predicted paths and their ranks vary. For data cache miss paths, the smaller inputs are unreliable in predicting the behavior of ref inputs.

4 Using single ref input as a compromise between train and combined ref inputs

As we can see from Section 3.2, the results are quite different when we consider individual reference inputs for comparing the effectiveness of smaller input sets. This leads us to the familiar problem of determining a suitable representative input set. From the above data, we can see that as the length of the input set increases, the ability to predict frequent paths also increases. Extrapolating this concept, we can have prefect predictability, if our input set is the reference input set itself. Of course, this is true only for the set of reference inputs under consideration. If a new one were to be added later, we would have two cases. If the new input exercises the same execution paths as the older inputs, then we still have perfect predictability. On the other hand, consider the case that it executes different execution paths. By virtue of the longer runtime of the older ref inputs there is a high probability that we would have encountered that path, though it may not have been "hot". We claim that using larger input sets can represent program behavior more faithfully. For the rest of this paper, we will be considering only the ref input set.

Combining all the ref inputs to generate profiles for PBO is a difficult and time-consuming task. Instead, in Section 1, we proposed HPM based sampling of larger input set. In this section, we analyze the feasibility of this approach by looking at various aspects of program behavior like IPC, pipeline stall cycles breakdown, frequent execution paths, program phases and by adding new ref input sets.

4.1 IPC Comparison

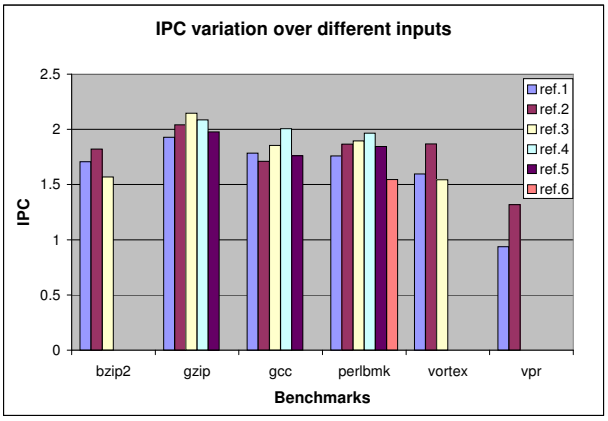


Figure 1

Figure 1 shows IPC (Instructions per cycle) as a coarse way of comparing the performance of a binary for different inputs. The variation in the IPC indicates that the performance of a program depends on the characteristics of the inputs and that of the program itself. Figure 1 does not show much variation except for vpr. This is true for the set of ref inputs provided by SPEC. What if we generated new inputs to simulate more usage patterns? The simplest option is to interchange the ref inputs for gzip and bzip2 since both are compression programs (three of the inputs are infact common).

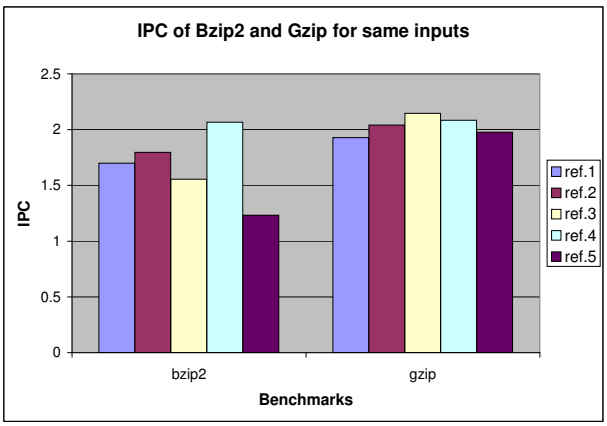


Figure 2

As seen in Figure 2, the same input applied to two different compression programs shows a large variation in IPC values. The variation in IPC of bzip2 ranged from 1.2 for ref.5 to 2.06 for ref.4 due to the addition of two new inputs (ref.4 and ref.5). It is reasonable to expect that the deviation would increase as we keep adding more varying inputs. This paper addresses the issue of predicting the variation of a program performance for various inputs using a single one.

4.2 Stall Cycle Breakdown

A typical PBO consists of two steps, generating the profile and using it to guide the optimization. Using one ref input to guide the optimization of the program will work if and only if the

bottleneck does not vary across the inputs. IA-64 HPM breaks the backend stalls into five categories viz. Flush cycles (Flush), Register Stack Engine (RSE), L1D, Execution stage (Exe) and Front End stalls as shown in Figure 3.

The actual number of each type of stalls is not important as long as the bottleneck is the same across different inputs. In Figure 3 we are mainly interested in the variation of the relative percentages of the stall category for each benchmark. For example, bzip2 and vortex exhibit very stable behavior, which means that one input can be used to predict the bottleneck for another. On the other hand, the rest of the benchmarks show some variation. For example, RSE stalls are significant for perl inputs 1, 2, 5 & 6 but not for 3 and 4. Interestingly, perl input 2, 3, 4 and 5 are the same input file run with different parameters, which reinforces the fact that program behavior changes with input parameters.

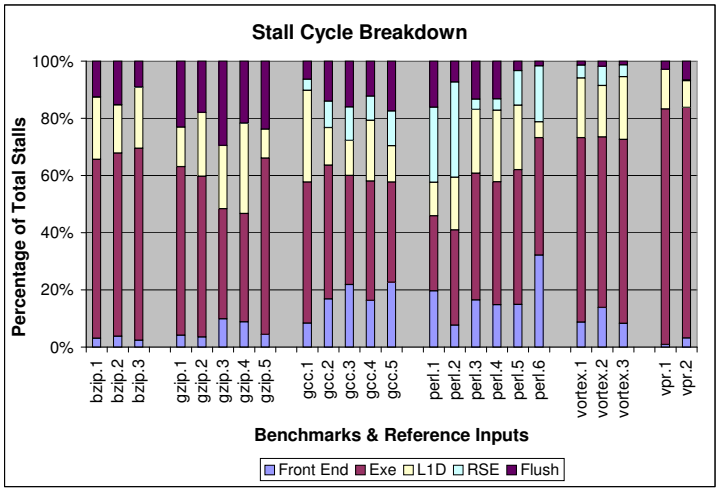


Figure 3

4.3 Execution path analysis for different reference inputs

One way of looking at bottlenecks is by analyzing and comparing the frequently executed paths and finding stall reasons for those. Figure 4 shows the top 5 "hot" traces for each of the benchmarks when running different inputs. A "hot" trace is one that is sampled most often by the BTB, using a sampling rate of one sample every 100,000 cycles. The number following the benchmark name on the X-axis is the reference input number.

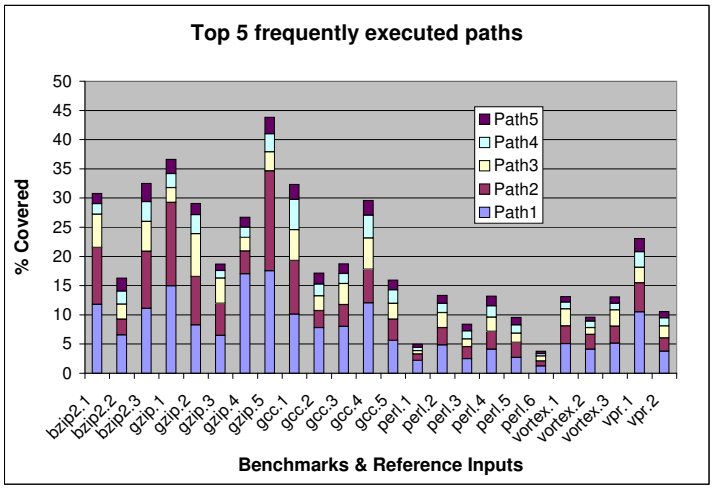


Figure 4

The top 5-execution paths account for a large percentage of the execution time, especially for bzip and gzip. For any particular benchmark, the percentage of the top paths changes with the input used. But do the top paths themselves change? If the top execution paths remain stable across different inputs then PBO can use a profile generated by one input run for optimizing other input runs. Our results show that for most of the benchmarks the topmost path is not the same. For example, the top path in bzip2.1 lies in procedure getAndMoveToFrontDecode, but for bzip2.2 it lies in sortIt and for bzip2.3 it lies in generateMTFValues. On the other hand the top trace of vortex lies in procedure Mem_Get_Word for all the reference inputs. In general, the top 10 traces for any ref input will be present in the top 10 traces for other ref inputs. However, their order and percentage execution time may be different. The notable exception is vpr since each ref input of vpr exercises completely different sections of the binary.

The above data would suggest that we can optimize the set of top traces for one input and expect the binary to be faster for other inputs as well (since sets of top traces intersect). However, if PBO chooses to optimize 50% of the top traces detected by ref1 for gzip, we cover only 11.53% for ref.3 and 8.66% for ref.4. A similar case was observed for perl. Of course, for vpr, we get 0% coverage for ref.2 if we optimized using ref.1 and vice versa. Thus the composition of the top traces and their weights are equally important. Figure 5 illustrates this point with an extreme case for perl benchmark. Here the trace coverage of perl ref inputs are compared against 50% coverage of ref input 2. The resulting coverage varies from 0.24% for ref.6 to 46% for ref.4.

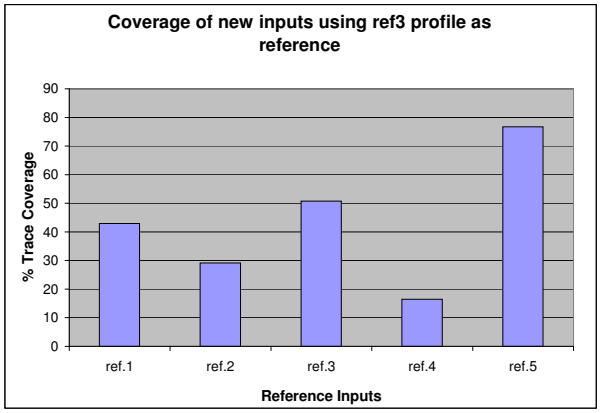


Figure 5

Let us assume that we want to optimize the top 5 traces of any benchmark in Figure 4. What are the chances that the traces we select using one reference input will show up among the top 5 traces when using another reference input? Figure 6 shows the percentage covered by each of the top 5 execution paths of gzip reference input 4 when run with other inputs in the reference data set.

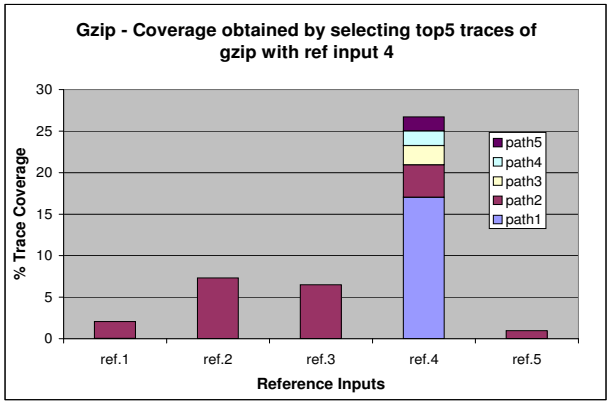


Figure 6

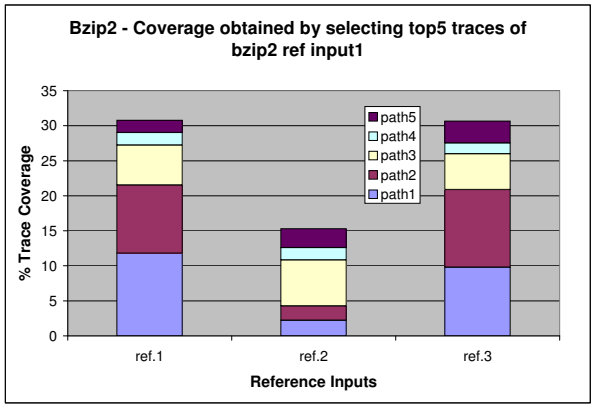


Figure 7

In Figure 6, 4 of the top 5 paths of gzip for ref input 4 do not show up for other inputs. On the other hand, for bzip2 (and vortex) the paths can be accurately predicted but their weights may vary as shown in Figure 7. The variation in weights is important since it may indicate that the trace is hard to predict. As an example, if the loop induction variable is not a constant i.e. it is passed as a parameter to a subroutine or if it is generated afresh before the start of the loop, the number of iterations of the loop can be variable. Correspondingly, the time spent in the trace can vary.

Figure 6 and Figure 7 show that some paths may not exist or their weight may be skewed when predicting the behavior of one reference input using another reference input. Of course, a case can be contrived where the execution paths are entirely predictable (for example, Vortex). On the other hand, in the case of vpr, we cover 0% when using top 5 paths of one reference input to predict for another. Some readers may dismiss the above examples as perturbations and surmise that the bulk of the ref inputs are predictable. But it is important to note that we are dealing with a fixed set of inputs. What if we added new inputs to a supposedly predictable ref input 3 of bzip2? The next section discusses this scenario.

4.4 Adding new ref inputs

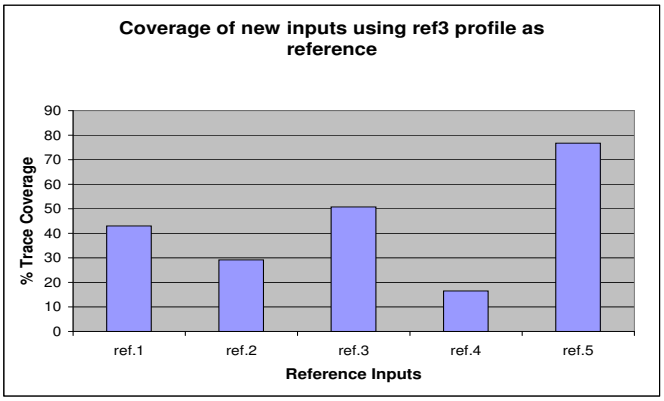


Figure 8

To simulate the performance of a program for newer input sets, the log and random input of gzip were fed to bzip2. From Figure 8, we can see that if we had used the profile collected by ref.3 to optimize a binary, we see two extreme cases – very low coverage for ref4 and very high for ref5 (the 2 new inputs). The top10 paths of ref 4 do not include the two topmost paths of ref input 1 and 3 indicating that different execution paths were executed. Thus, the fact that a given program is predictable for the given input set does not imply that it will be so for any input set.

4.5 Phase Change Behavior

A phase is defined as a period of stable behavior [Hsu02]. At regular intervals, the set of top traces in a profile is calculated. If the current set is different from the one in the previous interval, a new phase has been detected. Lower number of phase changes signifies stable behavior at

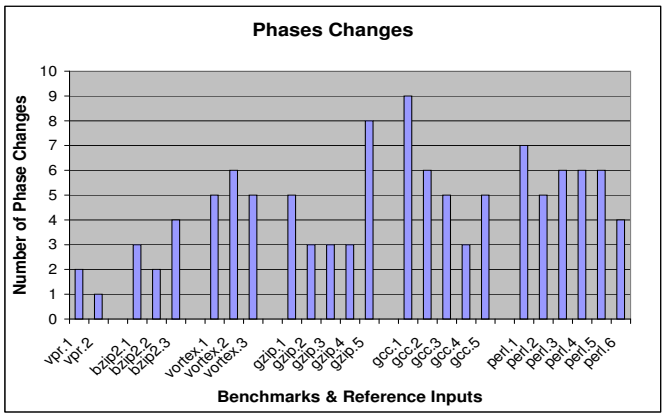


Figure 9

runtime. From Figure 9, we can see that the number of dynamic phase changes for a benchmark run varies with the input. Though this is not of much significance for static compilation, a runtime system can use this information for power saving or adaptive optimization. In the above figure, the variation in the number of phase changes for bzip2 is from 2 to 4. However, if we feed the "log" ref input of gzip to bzip2, the number of phase changes detected is 147!

5 HPM Profile Feedback Implementation

Section 3 showed that the smaller inputs cannot adequately represent the ref input. Section 4 established that significant differences exist even between the different ref inputs. We claimed that a larger input, say a single ref input, can predict other ref inputs better than a smaller input. To validate our claim, we used the ref input to drive the PBO. Surprisingly, we found that there was no discernible speedup. There were two reasons for this – first, PBO techniques depend heavily on edge profiles. The better quality path profiles obtained by the longer inputs are not utilized since path profile feedback is not yet implemented in most compilers. Secondly, current compiler implementations cannot take advantage of the plethora of counters provided by the IA-64 HPM. Implementing a full-fledged HPM feedback compiler is a non-trivial task. Therefore to test our concept, we decided to consider an optimization that would benefit most from the path profile information. We chose to implement the procedure inlining optimization based on dynamic call path information. The ORC compiler is used for HPM based inlining primarily because it is open source and has a caller-callee script feedback mechanism in place. Our task was to use information collected by HPM to generate a caller-callee script and feed it to

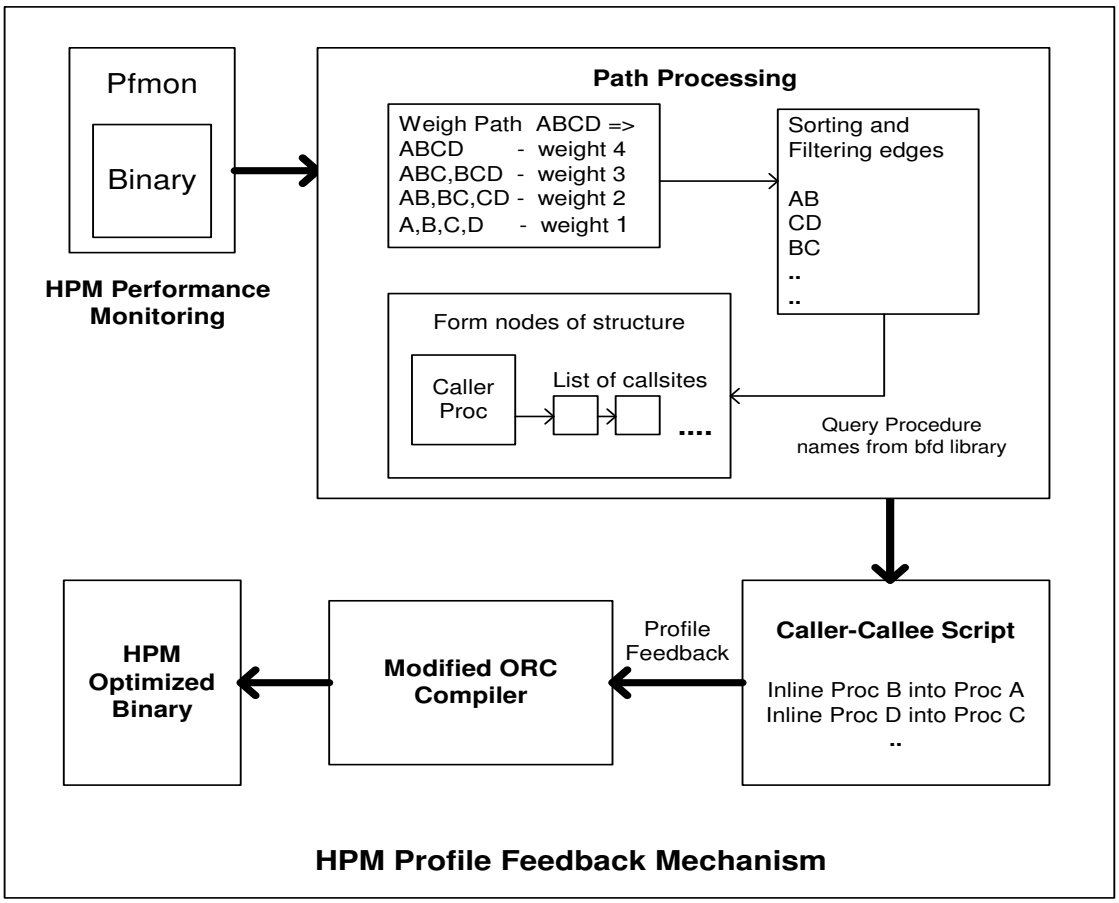


Figure 10

the ORC compiler to produce the optimized program. The ORC component that reads the script was modified to add certain checks necessary when inlining e.g. to guard against code size bloat. The programs were compiled at O3 with IPA enabled. The IPA phase of ORC generates a call graph to analyze and optimize inter and intra-file procedure calls. Our script file provides "hints" to the IPA optimizer when it decides which call-sites to inline. Since we based the enable inlining decision upon the dynamic call structure, we believed that the caller-callee script would enable IPA to make better decisions. Figure 10 shows the entire mechanism.

5.1 Generating the caller-callee script file

The opcode-matching feature of the IA-64 HPM was used to restrict sampling to "Call" instructions. Every sample lists a set of the last four "Call" instructions encountered along with

their target addresses. In other words, each sample is an execution path. These execution paths are sorted by frequency to enable us to locate the "hot" paths. It is possible that though an execution path is not hot, a portion contained in it is hot. For example, consider Path A-B-C-D where each literal is a procedure. Here it is possible that there are paths like E-F-C-D and X-Y-C-D, making the subpath C-D hotter than either of the paths. In order to account for such cases, we need to break the execution paths into all possible combinations of its literals and weigh each such path by a combination of its path length and frequency. This approach ensures that longer paths have higher weight than shorter ones. Therefore, shorter paths will show up as frequent paths only if their frequency is high enough. The entire weighing mechanism ensures that we get a "path profile effect" though we deal with individual edges.

Each variable length path is now broken into single caller-callee pairs and the weight assigned in the previous stage is carried forward. All call sites belonging to a single procedure are grouped by querying the bfd library to get procedure names. Thus, we have a node corresponding to each caller procedure and a linked list of all call sites within that procedure along with their weights and approximate invocation counts. This information can be used to fine-tune the script by selecting an appropriate subset of the call sites to inline. Using heuristics, we have opted to inline all call sites as long as they satisfy some minimum frequency and invocation count requirements.

5.2 Results

The above script is fed to the modified ORC compiler and the performance of the resultant binary is compared with the one generated using PBO (train input). Figure 11 shows the results. HPM feedback optimized binaries are just as fast or faster than PBO based binaries (except for gcc) though they implement just the inlining implementation. It is possible to make HPM gcc run faster than PBO gcc by tweaking the caller- callee script generation parameters. By adjusting these parameters, it is possible to speedup bzip2 and vpr even further. However, an inlining edge

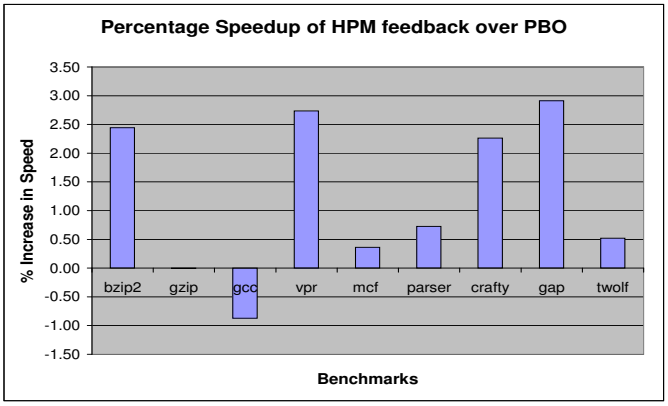


Figure 11

selection criterion that speed up one benchmark may slow down the other benchmarks. Therefore, Figure 11 uses a set of parameters, which balance the speedups across all the benchmarks. With more intensive analysis, it may be possible to come up with parameters that produce optimum binaries for each benchmark.

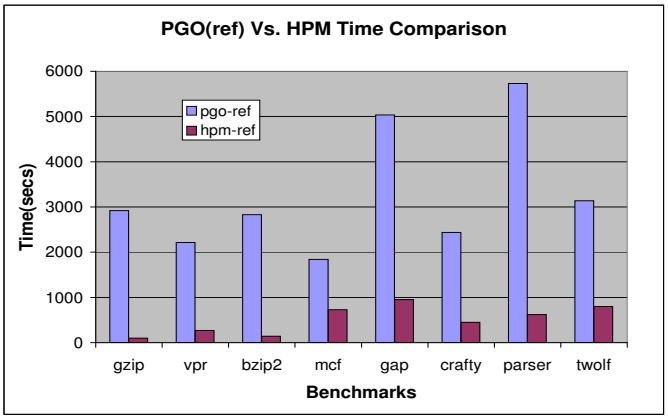


Figure 12

One of the main objectives of this paper is to show that the ref input can be used for generating profiles without high overhead. Figure 12 shows the time overhead of using the ref input for generating profiles using instrumentation (PGO) and HPM. The system "time" command is used for the above comparison. The PGO time is the sum of the time needed to instrument and collect profile and to recompile the binary. The HPM time consists of the time needed to compile, run and recompile the program. HPM sampling overhead is miniscule in comparison with PGO, primarily because HPM sampling does not use instrumented binaries. PGO optimization requires "instrument -> run -> recompile" cycles while HPM requires "compile -> run -> recompile" cycles. The "recompile" stages of both approaches take approximately the same time. Comparing the above two cycles, it is evident that the "instrument -> run" stages of PGO will always take more time than the "compile -> run" stages of HPM. This accounts for the huge difference among the two in Figure 12 and is precisely the reason that the ref input is not used for PGO. In Figure

12, a profile could not be generated for gcc since the large number of small code segments caused the instrumented binary to fail. In actual practice, the train input is used to generate profiles. Figure 13 shows the time comparison for PGO (test and train) and HPM sampling. In this figure the time advantage of HPM is not as evident, especially for benchmarks whose train and test input runtimes are much smaller than the ref input runtimes. For such benchmarks, PGO using train input would be a better option if the train input faithfully represents the ref input; but our data so far shows that this is not always the case. In the case of gcc, the IPA analysis phase takes up a large chunk of the runtime. HPM feedback script hints prevent unnecessary IPA analysis by forcing the compiler to inline certain functions.

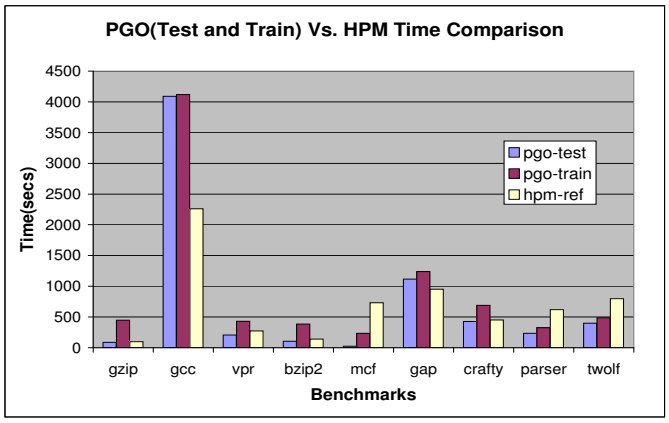
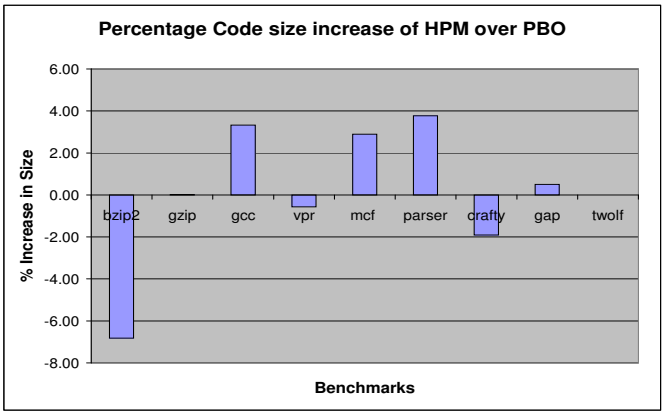


Figure 13

The memory requirements of the two approaches fall in different domains. PGO uses internal profile data structures whose size is bound by the size of the instrumented binary. HPM maintains a log file for profile collection. The size of the log file depends on the granularity of sampling and the type of performance events captured. HPM profiles need to be processed before feeding them to the optimization phase whereas PGO accumulates profile information in the instrumented binaries. If online processing of the log files is used, the space overhead can be considerably reduced.

Aggressive procedure inlining can benefit certain programs like bzip2 or backfire for certain programs like gcc. Thus, inlining more procedures can make programs like bzip2 faster. Figure 14 compares the code size of the binaries generated by PGO and HPM. The decrease in the size of the bzip2 binary shows that the speedup shown in Figure 11 was obtained by accurately selecting “hot” paths and not by increasing the number of inlined procedures. For the



other benchmarks, some of the "hot" procedures were not inlined by PGO. Hence the code size of those benchmarks increase. The instruction cache performance was not adversely affected by the change in code size.

Figure 14

5.3 Scope for additional speedup

Readers may note that procedure inlining is just one of the many optimizations that can benefit from HPM sampling. Procedure inlining allowed us to build a baseline binary which is faster than traditional PBO. The potential speed up can be much higher if additional optimizations to take advantage of speculation and advanced load failure counters, cache miss rate counters, TLB miss rate counters etc are implemented. Traditional instrumentation cannot provide such a rich variety of feedback. Moreover, it is difficult to model the train input to accurately represent these counters for the various ref inputs. Therefore, using a single ref input rather than train for HPM sampling gives us a better launch pad for future optimizations. For example, Mcf suffers from excessive data cache misses. HPM sampling of data cache miss addresses shows that three memory loads account for more than 30% of the total data cache misses. The train input detects these but lowers their contribution to around 20%. The test input cannot detect any of the topmost data cache misses. By manually inserting instructions to prefetch these lines into cache, Mcf can be speeded by around 20%.

6 Conclusion

We have shown that the smaller input sets cannot accurately characterize the behavior of the larger input sets. Significant differences exist between the individual ref inputs, as evidenced by

an analysis of various parameters like the stall cycle breakdown, execution paths, and phase changes. Although using multiple larger input sets can more accurately predict the program behavior, it is not suitable for traditional instrumentation based profiling approach. To overcome this barrier, we propose using hardware performance monitor (HPM) sampling based profiles to guide compiler optimizations.

We implemented a HPM based sampling system on the Itanium processor based systems. Since current compilers do not support such feedback, scaffolding is implemented to process and feed the sampling profile to a modified ORC compiler. The results obtained show that the optimized binaries are faster than those obtained with traditional PBO by up to 3% with much lower overhead. This approach would allow the use of multiple, larger input sets to guide various optimizations. Furthermore, by taking advantage of the rich variety of performance critical information available from the HPM (e.g. Data cache miss addresses and data speculation failures), potentially, more optimization opportunities can be effectively exploited by the compiler.

7 References

[Cont96] Conte, Patel, Menezes, Cox, "Hardware-Based Profiling: An Effective Technique for Profile-Driven Optimization", Intl. Journal of Parallel Programming, Vol.24, pp. 187-206,Apr 96.

[McFa03] Scott McFarling, "Reality Based Optimization", CGO 2003.

[Haze03] Hazelwood & Grove, "Adaptive Online Context-Sensitive Inlining", CGO 2003.

[Hsu02] Wei C. Hsu, Howard Chen, Pen Yew, D-Y. Chen, "Phase Locality Detection and Exploitation Using Branch Trace Buffer in the IA-64 Architecture", TR-02-006, Department of Computer Science, University of Minnesota, 2002.

[Wall00] David Wall, "Predicting Program Behavior Using Real or Estimated Profiles", Proc. of ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization, pages 1-11, 2000.

[Pfmo02] Pfmon documentation, http://www.hpl.hp.com/research/linux/perfmon/pfmon-common.php4, HP Labs, 2002.

[Itan03] Intel® Itanium® 2 Software Development and Optimization Reference Manual, Intel Corp. April 2003.

[ORC02] Open Research Compiler for ItaniumTM Processor Family, http://ipf-orc.sourceforge.net, and 2002.

[ECC02] Intel C++ Compiler 7.1 for Linux, http://www.intel.com/software/products/compilers/clin, Intel Corp., 2002.

[Chan92] Chang, Mahlke, Hwu, "Profile-guided Automatic Inline Expansion for C Programs", Software-Practice and Experience, 22(5): 349–369, 1992.

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

[Ammo97] G. Ammons, T. Ball, and J. T. Larus, "Exploiting Hardware performance counters with flow and context sensitive profiling", ACM SIGPLAN Conference on Programming Language, Design and Implementation, 1997.

[Cohn97] R. Cohn, D. Goodwin and P.G. Lowney, "Optimizing Alpha Executables on Windows NT with Spike," Digital Technical Journal, vol. 9, No. 4, pp. 320, 1997.

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

[Pett90] Karl Pettis and Robert C. Hansen, "Profile guided code positioning", Proceedings of the conference on Programming language design and implementation, Pages 16 – 27, 1990.

[Kris02] A. Krishnaswamy and R. Gupta, "Profile Guided Selection of ARM and Thumb Instructions", ACM SIGPLAN Joint Conference on Languages Compilers and Tools for Embedded Systems & Software and Compilers for Embedded Systems (LCTES/SCOPES), pages 55-63, Berlin, Germany, June 2002.

[Coll01] Collins, H. Wang, D. Tullsen, C, Hughes, Y. Lee, D. Lavery, J. Shen, "Speculative Precomputation: Long-range Prefetching of Delinquent Loads", In ISCA 28, July 2001.

[Chil02] Trishul M. Chilimbi and Martin Hirzel, "Dynamic hot data stream prefetching for general purpose programs", PLDI'02, pages 199-209, ACM Press, 2002.