Profile-Based Optimization with Statistical Profiles

Citation

Gloy, Nick, Zheng Wang, Cathering Zhang, Brad Chen, and Mike Smith. Profile-Based Optimization with Statistical Profiles. Harvard Computer Science Group Technical Report TR-02-97.

Permanent link

http://nrs.harvard.edu/urn-3:HUL.InstRepos:23518801

Terms of Use

This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

Share Your Story

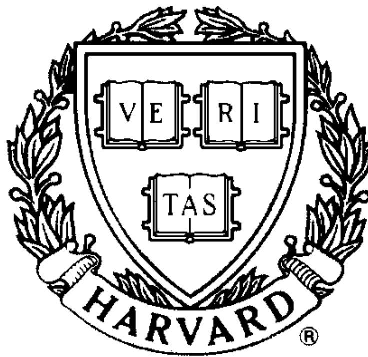
The Harvard community has made this article openly available.
Please share how this access benefits you. Submit a story .

Accessibility

Profile-Based Optimization with Statistical Profiles

Nick Gloy, Zheng Wang, Catherine Zhang,
Brad Chen, and Mike Smith

TR-02-97
April 1997



Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

Profile-Based Optimization with Statistical Profiles

Nick Gloy, Zheng Wang, Catherine Zhang, Brad Chen, and Mike Smith
Division of Engineering and Applied Sciences
Harvard University

Abstract

An important barrier to the use of profile-based optimization is the generation of high-quality profile data. In this paper we describe our experience with a prototype system for continuous and automatic generation of profile data. Our system collects profile data continuously for all computation on a system by sampling the program counter once per hardware clock interrupt. We have found that the overhead of our prototype monitor is negligible, making it practical to collect profile information continuously for all activity on the system. We used several qualitative and quantitative methods to explore how the quality of these sampled profiles improves over time, and demonstrate that profiles of adequate quality for optimization can be obtained after a small number of monitored runs. For a selection of instruction-cache intensive benchmarks, statistical monitoring provides profiles of a quality comparable to that of complete profiles obtained with program instrumentation while avoiding the large run-time overheads of instrumentation-based profile collection. Our monitor is a component of the Morph System, which will make feedback-based and architecture-specific optimizations automatic and transparent to the user.

1. Introduction

Although potential performance improvements from profile-based optimizations (PBOs) are well documented [Chang et al. 1991, Chen et al. 1994, Fisher et al. 1984, McFarling and Hennessy 1986, McFarling 1989, Pettis and Hansen 1990, Santhanam and Odnert 1990, Wall 1986], the use of these optimizations has been limited. The key problem with PBOs is the difficulty in obtaining and applying high-quality profile data. In current systems, the creation of an optimized executable is typically a multi-step process, requiring a programmer to:

1) create a representative set of test inputs;

2) create an instrumented version of the application executable;

3) run the instrumented executable on the test inputs to generate profile data;

4) process the profile data for use by a profile-driven, optimizing compiler;

5) create a new executable by applying the profile data during (re)compilation;

6) test/evaluate performance of the new executable.

From these steps, it is clear that the use of PBOs requires knowledge of the intended use of the application and a substantial background in compilers and performance evaluation. Typically, end-users of an application do not have the technical sophistication required to perform these tasks (tasks 2, 4, and 5 in particular), and thus end-users often do not apply PBOs. Furthermore, most profile-driven optimization systems require access to the source code or at least more information than is found in the stripped executables sold to end-users. Because of the difficulties in applying PBOs, these techniques are generally available only to sophisticated users, and even among these users, PBOs are frequently not used due to complex manual process required to apply them. We are developing the Morph Environment to address all of these problems. In this paper we discuss and evaluate one component of this larger system, the Morph Continuous Monitoring System (MCMS), that solves the problems in the first three steps in of PBO.

Continuous profiling of all applications as they are used eliminates the need to generate representative training input. To make the continuous collection of profile information practical, the overhead of profile collection in MCMS is kept low, on the order of 0.1% of the application's execution time. We achieve this low overhead by periodically collecting profile samples, rather than instrumenting the executables to collect complete profile information. The overhead of instrumentation is in the range of 10% to 400% [Ball and Larus 1994], several orders of magnitude greater than the overhead of the MCMS. Given such low sampling overhead, we can collect profile information continuously for all computations, rather than requiring special instrumented runs. With sampling however we do not obtain a complete profile of a program execution as we do with instrumentation. As our results show, even a single sampled run of an application often provides enough profile information to apply an optimization. Our results also show that MCMS is able to collect profiles of a quality comparable to a complete profile over a small number of application runs.

In the next section we discuss several characteristics of statistical profiles and their implications for MCMS. Section 3 describes the high-level design of MCMS and gives details on our prototype

implementation. In Section 4, we give experimental results for statistical profiles. We present both visual and quantitative results to support our claims of low overhead and high quality for statistical profiles. We discuss related work in Section 5, and Section 6 concludes.

2. Characteristics of Statistical Profiles

In current practice, the profiles used for PBOs are typically generated using an instrumentation system [Smith 1991, Wall and Powell 1987, Wall 1992, Srivastava and Eustace 1994]. Instrumentation code is added to the original executable so that it can produce a record of the program activity when the program is executed. The key property of this type of profile generation is that it can capture a complete profile including all application activity for a given input data set. A PBO system uses this complete profile to direct its optimization efforts. The downside of this approach however is the significant performance penalty experienced during profile collection. In contrast, the MCMS does not use instrumentation, and thus is able to collect profile information without a noticeable degradation in application performance.

MCMS produces a *statistical* rather than a *complete* profile. During a single program run, the MCMS samples a small (but hopefully representative) fraction of the activity of an application. In contrast, a complete profile provides a complete record of all program events. We refer to the profile information produced by sampling a single program execution as a *sample set*. A single sample set may not include enough information to achieve the full potential benefit of a given PBO. This makes it desirable to collect and combine multiple sample sets into a single profile. The goal of this process is to create an aggregate statistical profile that is representative of the program behavior described by a complete profile.

We aggregate sample sets by a simple combining process. The contribution of a sample set to the overall profile is in proportion to the ratio of program run time for that sample set to total time for all sample sets. For the code-layout optimization discussed later in the paper, we sum the execution counts from each program execution for each basic block to create an aggregate basic block execution profile. Combining sample sets in this way models the actual usage of the application, as each run of the application is represented in the cumulative profile in proportion to the amount of execution time it required.

The issue of combining profile data is not unique to our periodic sampling methodology. It is also desirable to combine information from multiple complete profiles in support of a PBO. Complete profiles are typically collected from a limited number of *scripted* training inputs sets, in contrast to the continuous profiling of a user's day-to-day activity as occurs with MCMS. Several methods have been proposed and evaluated for aggregating the profile information from the multiple scripted test inputs. For example, Fisher and Freudenberger [1992] evaluate three techniques for combining multiple branch profiles: *unscaled*, *scaled*, and *polling*. They found little difference between the scaled and unscaled results. The MCMS implements unscaled combining for the reasons discussed above. The continuous nature of MCMS enables us to build a PBO system that tracks changes in application usage patterns, accommodates idiosyncratic usage, and optimizes for behavior not covered by test input scripts.

The continuous generation of profiling data also suggests the issue of how to age profile data as it is collected over an extended period of time. There are a number of simple approaches. We leave this as an open area for future work.

Our profiles tend to give more weight to basic blocks with higher latency and higher average CPI, as they have a higher relative probability of being sampled; we call this a *time-based profile*. In contrast, profiles generated by instrumented program execution give equal weight to all instructions, regardless of their relative latency, a *frequency-based profile*. Because of this difference, time-based profiles and frequency-based profiles will not have an identical shape. We could attempt to adjust the time-base in statistical profile collection so that the results would be more similar to that of a complete profile. However, our experience to date suggests that time-based profiles may be more effective for profile-based optimizations (see Section 4.3). Readers should also note that several modern processors (including Digital's Alpha [Digital 1996] and the Intel Pentium [Intel 1995] and Pentium Pro processors [Intel 1996]) provide hardware support for collecting statistical frequency-based profiles in the form of an interrupt driven by a countdown event counter. This could be done by sampling the PC every period of n instructions, rather than sampling on a time interval as with our current profiler.

3. System Design

For this paper, we implemented a simple prototype monitoring system that samples the program counter on each clock interrupt. The simplicity of the logging operation and the relatively low

frequency of clock interrupts provide sampling overhead low enough that profile data can be collected continuously with no significant performance penalty. For reference, Table 1 lists clock interrupt rates for several popular systems. For our experiments we used an AlphaStation 400-4/233 from Digital Equipment Corporation. This system contains a 2MB second level cache and 192 MB of main memory. Our system ran DigitalUnix (version 4.0) and had a clock interrupt rate of 1000Hz.

System	clock interrupt rate
Windows 3.11	50 Hz
NetBSD Unix	100 Hz
Windows NT	100 Hz
Digital Unix	1000 Hz

Table 1: Clock interrupt rates for several common systems.

Because we sample the program counter using standard hardware support for interrupt handling, we can collect profile information while avoiding the overhead of software instrumentation.

3.1. Prototype Implementation

In this section we describe our prototype implementation of MCMS. We have made a number of simplifications for this first implementation. They are justified in that the focus of this study is on understanding how to handle the samples after they have been collected, rather than providing a definitive implementation of the sample collection system. We accompany our description of the prototype with an analysis of the overhead that could be expected from a production implementation of the MCMS. For this analysis, we have (somewhat arbitrarily) chosen 0.1% as an upper bound on acceptable overhead for continuous monitoring. At this level, the overhead of monitoring will be undetectable for most users, and at least an order of magnitude smaller than the benefit anticipated from PBOs.

We used the Atom tool-building system [Srivastava and Eustace 1994] to implement our prototype monitor. Atom provides a convenient way to implement our monitor as a kernel extension without the need to configure a new kernel or modify the kernel source code. We used Atom to instrument a single kernel routine, `hardclock()`. The pseudo-code for our instrumentation routine is:

if (*the interrupted computation was in user mode*)

if (*the interrupted process is the measurement subject*)

save the program counter sample to the log

To keep our prototype kernel implementation simple, we sample PC values from only a single target application, rather than sampling PC values from the operating system kernel and all active applications. In our continued work we are developing a monitor which samples all execution contexts, and maintains information for mapping sampled addresses back to the corresponding instructions in executable images.

Clock interrupts on our test machine occur at 1000Hz, or 1 millisecond per clock interrupt. Given an overhead goal of 0.1%, the maximum overhead during the clock interrupt routine for saving a PC sample is 1 us, which corresponds to 200 machine cycles on our test machine. Achieving this goal is straightforward with conventional coding techniques. For lower clock interrupt rates, the logging operation could be slower and the 0.1% overhead performance goal would be easy to achieve.

Another simplification made for this prototype is that our sample buffer is large enough so that it will not become full during an application run. A more realistic implementation would use a smaller buffer and empty it periodically. At 1000 Hz and 4 bytes per PC sample[1], we generate samples at a maximum rate of approximately 4K bytes per second. At this rate, a 32K byte buffer would allow monitoring for 8 seconds of CPU time before the sample buffer became full. The overhead for saving the sample buffer to stable storage must be under 8 milliseconds in order to meet the 0.1% overhead goal. This is enough time to perform a synchronous write for a modern hard disk [Seagate 1994]. An design alternative would be to use an asynchronous write; in this case the overhead for the write would be substantially less than 8 milliseconds.

Our analysis shows that it would be straightforward to achieve less than 0.1% overhead in a production version of the MCMS. The feasibility of MCMS does rely on the availability of additional disk space for storing profiles, memory for buffering the samples, and sufficient machine idle time to process samples off-line. In our continued work we are evaluating these issues with a complete MCMS prototype.

---

[1] A PC value on the Alpha is 64 bits, but the address space is sparse. This should make it possible to uniquely identify a PC value with a 32 bit trace entry as well as additional context information stored in the trace.

3.2 Processing Samples

Further sample processing is performed off-line by a low priority process. A number of data transformations are required to convert sample sets into a code layout. Figure 1 illustrates how the tools are composed to transform multiple sample sets into an optimized code layout. Table 2 lists the tools we implemented to process samples, as well as additional tools we used from Digital Unix. Our tools use the file format defined by the Digital Unix implementations of *pixie* and *cord*, enabling us to use Digital's implementation of those tools.

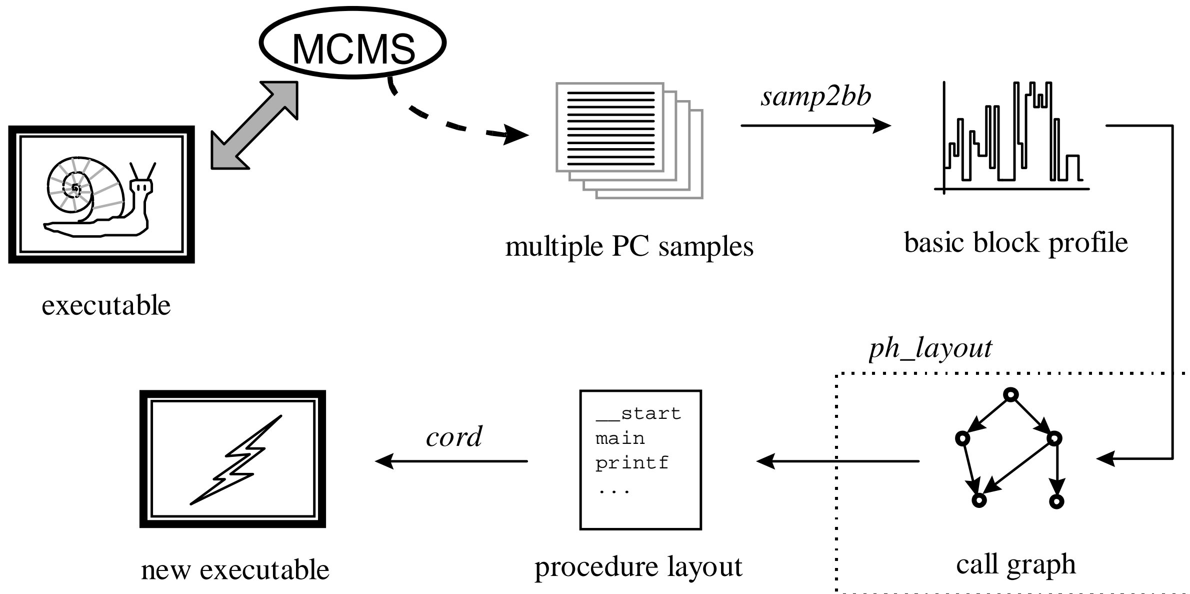


Figure 1: Generate a new executable from profile samples.

Tool	description
samp2bb	convert PC sample sets into a basic block profile in pixie format.
ph_layout	process a basic block profile in pixie format to generate a procedure ordering. Uses an algorithm based on the on proposed by Pettis and Hansen [1990].
cord (from Digital)	apply a procedure ordering to an executable to generate an optimized executable.
Pixie (from Digital)	generate a complete basic block profile.

Table 2: Profile generation and code layout tools.

When MCMS has monitored a running program and generated some PC samples, the next step in Figure 1 requires converting a PC sample set to a basic block profile. This is implemented in the tool *samp2bb* using the following simple algorithm:

for (*each pc sample*)

identify the basic block containing the sampled PC

increment the profile count for the basic block, scaling by $^{1}/_{(\text{basic block size})}$

To understand the need for scaling of PC samples, consider two basic blocks, one containing three instructions and one containing six instructions, and both of which are executed the same number of times in a given application run. Figure 2 illustrates this example. As a simplification we assume that each PC value in the two basic blocks is equally likely be sampled. A single PC sample is a witness for the entire basic block containing it. However, if we increment the basic block counter by one for each sample, the resulting profile would incorrectly suggest that block B was executed twice as often as block A. By scaling the increment of the basic block counter by the inverse of the basic block size in instructions, we get a better approximation of the complete profile.

Program code			Statistical profile	
	complete profile	sample set	without scaling	with scaling
`Loop:`				
`...`				
`block_A:`			3	1
`add a0, r2, r3`	1000	0		
`ld a1, 8(sp)`	1000	2		
`blt zero, a0, block_C`	1000	1		
`block_B:`			6	1
`sra a0, 0x38, a0`	1000	0		
`ldq_u zero, 8(sp)`	1000	3		
`bis a0, a0, v0`	1000	0		
`subl v0, 0x41, t3`	1000	2		
`cmpult t3, 0x2b, t4`	1000	0		
`add t3, t2, t1`	1000	1		
`block_C:`				
`...`				
`jmp Loop`				

Figure 2: Sample scaling for two basic blocks.

After the transformation from PC samples to basic block profile, we combine the individual profiles from each run to generate a single basic block profile. The following algorithm is used to convert the basic block profile to a weighted call graph (WCG):

for (*each basic block*)

for (*each call instruction*)

increment the edge <callee, caller> *in the WCG*

Currently, monitoring system does not maintain edges for call instructions that can have multiple targets, as can commonly occur when a procedure pointer is used. For our experimental benchmarks, this limitation excludes from 0.9% to 12% of the call instructions from the profiles used for optimization. Including all edges would improve the performance of code layout, and we are currently building a new monitor which will allow us to maintain weights for all call graph edges.

The resulting WCG can be used directly by our code layout tool. We compute an optimized procedure layout using *ph_layout*, an implementation of the Pettis and Hansen procedure layout algorithm [1990]. In this algorithm, code layout is determined by selecting edges from a weighted call graph (WCG) based on a greedy algorithm. The algorithm starts by selecting the edge with the highest edge weight and placing the caller and callee procedures adjacent in the resulting layout. The nodes in the graph corresponding to the two procedures are then merged. The algorithm continues recursively until all edges have been removed from the graph.

3.3 Workloads

We used a selection of workloads from the Spec95 integer suite to evaluate our profiles. In selecting these workloads, we avoided workloads (such as *ijpeg* and *compress*) that have small instruction-cache working sets as they do not benefit from code layout optimization. We also included *ghostscript*, a popular public-domain PostScript interpreter. Details for these workloads are given in Table 3.

Benchmark	Description	Text section size (KB)	Execution time (sec)
ghostscript	Postscript document viewer	1,898	68
gcc	The GNU C compiler	1,414	106
go	An oriental board game	326	362
li	LISP language interpreter	152	745
m88ksim	Simulation of the Motorola 88000 processor	207	814
perl	The perl scripting language	446	619

Table 3: Experimental Benchmarks. Execution times are for training inputs. All benchmarks were statically linked.

In our experiment we used different inputs for training and testing. For the SPEC95 benchmarks, we used inputs provided with the SPEC95 suite[2]. For *ghostscript* we trained using the Postscript source for a Ph.D. thesis, and tested using documentation from a public-domain compiler system.

4. Experimental Results

In this section, we will present experimental results to support three conclusions.

- The overhead of MCMS profiling is very low. (Section 4.1)
- Statistical sampling produces high quality profiles. (Section 4.2)
- Statistical profiles are effective for driving a procedure layout optimization. (Section 4.3)

From the above points we will argue that statistical profiles provide a sound foundation for implementing profile-based optimization.

4.1 Overhead of Profiling

The majority of the runtime overhead from MCMS is due to the PC sampling that occurs with each clock interrupt. For this operation, our prototype Atom implementation has higher overhead than would occur with a production implementation, due to the additional operations required to save and restore processor state before and after the invocation of an Atom instrumentation routine[3].

Benchmark	Without profiling	With profiling	Overhead
ghostscript	66.9	67.7	1.2%
gcc	109.5	106.2	-3.0%
go	362.5	362.0	-0.1%
li	743.5	744.6	0.1%
m88ksim	812.7	814.1	0.2%
perl	625.2	619.4	-0.9%

Table 4: Execution times (in seconds) with and without MCMS. All figures are the average of eight runs in single-user mode with cold cache. Standard deviation is less than 1% for all measurements.

We evaluated the overhead of sample collection for our prototype monitor by comparing execution time for our experimental workloads for two versions of the operating system kernel: the

---

[2] We used the SPEC95 reference inputs for training. We found that the training inputs provided with the SPEC95 suite were too short for training effectively.

[3] In particular, Atom saves and restores the thirty-one Alpha integer registers, a total of 248 bytes.

standard kernel and our Atom-based MCMS prototype. The results (Table 4) show the variation in execution time caused by side-effects of updating the kernel[4] make it difficult to identify the performance impact of monitoring. The change in execution time for the monitor kernel varied from 1.2% to -3.0%, with an average of -0.4%. Although this provides substantial evidence that the impact of monitoring is negligible, it does not provide conclusive evidence. We are preparing a set of experiments which will allow us to make a more definitive conclusion.

4.2 Statistical Profile Quality

To gain insight into the differences between profiles from different sources and how the quality of profile data changes as samples accumulate, this section presents three tests to evaluate the quality of the statistical profiles. In each of these tests, we compare our statistical profiles to a complete profile obtained by program instrumentation. Overall, we find that there is substantial similarity between the complete and statistical profiles, with the dissimilarity as would be expected in a comparison of frequency-based and time-based profiles. We now explore the similarities and differences in greater detail.

4.2.1 Visual Comparison

Figure 3 provides a visual comparison of three profiles for the *go* benchmark from the Spec95 suite: a complete frequency-based profile, a statistical time-based profile based on a single sampled run, and a statistical time-based profile from 32 sampled runs. Figure 3 demonstrates that even a single sample set provides a substantial amount of useful profile information, and that there is substantial similarity between the complete profile and the statistical profiles. However, it is not obvious from this example that the statistical profile will ever converge to have the same ‘shape’ as the complete profile. Detailed examination shows that in many cases the visible differences between the profiles can be attributed to the differences in time-based and frequency-based profiling.

---

[4] Such side-effects normally occur any time an executable module changes.

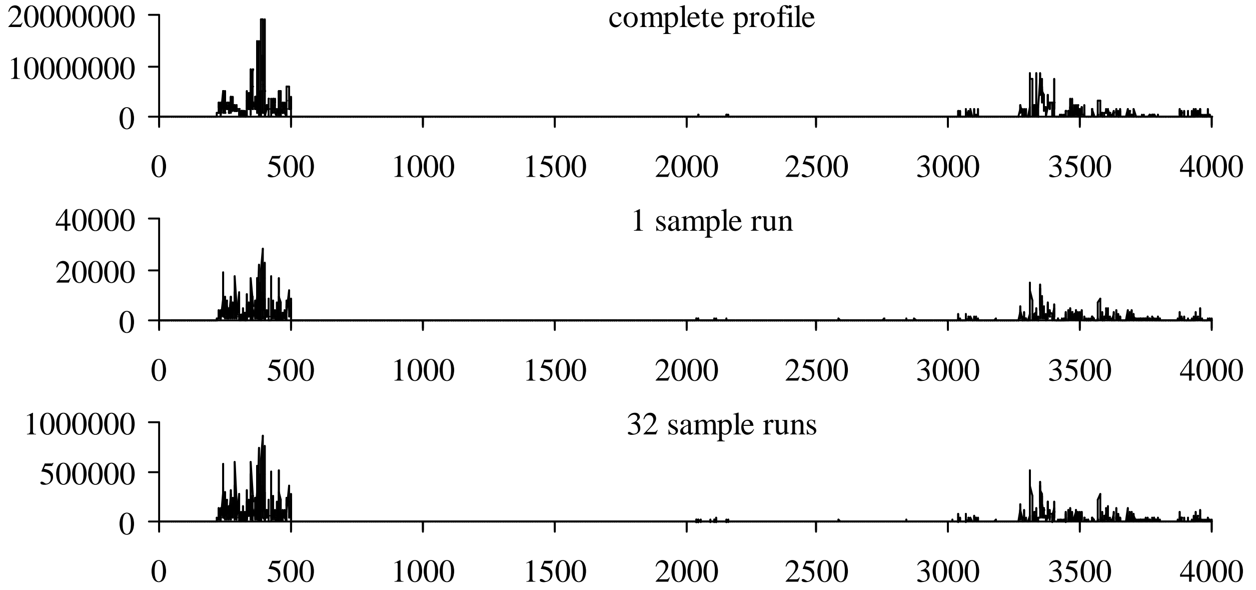


Figure 3: Complete and statistical profiles for the benchmark go. In each graph, the x-axis is the basic block index in the text section, and the y-axis is the dynamic execution count. This graph shows counts for the first 4000 basic blocks.

4.2.2 χ^2 Goodness-of-Fit Test

The χ^2 goodness-of-fit test is commonly used in statistical analysis to evaluate the similarity of two probability distributions [Lindgren 1976]. It compares observed occurrences of an event to a distribution of expected event frequencies. For a perfect match, the χ^2 test will give a result of zero. The larger the test result, the less we believe that the observed experiments follow the expected distribution. We used the χ^2 test to evaluate the similarity of statistical and complete profiles, computing the expected distribution from the basic block counts for the complete profile. Table 5 shows the results of the tests. Some pre-processing of profiles is required before the χ^2 test can be applied. The data sets are normalized so that they all appear to include the same number of 'trials' or profile samples.

For reference we provide test results from two artificial comparisons, *good match* and *expected*. The good-match is computed by scaling each profile sample in the complete profile by a real random variable over the interval [1..10), using a uniform distribution. Good match is representative of the difference that would normally be expected between a frequency-based profile and a time-based profile, where the probability that a basic block is sampled varies with the CPI for that basic block.

The "expected" case is computed by doing similar scaling as for good match, and by zeroing basic block counts that are less than 2% of the greatest count for any basic block. This is meant to represent a situation where, due to statistical profile collection, some number of basic blocks have not been sampled.

	ghostscript	gcc	go	li	m88ksim	perl
1 sample set	171344	140258	209313	352007	1119542	864738
2 sample sets	156339	127628	202976	361188	1122120	880994
4 sample sets	150473	121702	198392	359812	1124894	879797
8 sample sets	146519	118460	195273	357764	1110807	885162
16 sample sets	144895	116652	193291	357891	1111377	888644
32 sample sets	144226	114937	191692	356625	1108415	885387
"expected"	66753	130874	399505	454062	3272443	216883
good match	15802	22563	90597	165830	176846	130612

Table 5: χ^2 test for statistical profiles. This table shows results from using the χ^2 goodness-of-fit test to compare complete profiles and various statistical and reference profiles.

The tests show that the similarity of the statistical profile to the complete profile generally increases with larger numbers of sample sets. It also shows that the similarity of the sampled profiles to the complete instruction counts is consistent with our expectations, as indicated by the good-match and "expected" tests. Four of the benchmarks give test results within the bounds indicated by the "expected" test. The two exceptions are *ghostscript* and *perl*. In the case of *perl*, a single 1-instruction basic block[5] is responsible for a large part of the difference between complete and sampled profiles. This basic block is the target of seven distinct C goto statements and was sampled with a relative frequency that is 5 times higher than its frequency in the complete profile, possibly the result of branch prediction or instruction cache penalties. In Section 4.3 we will see that the *perl* statistical profile can be used effectively for optimization, despite the differences between the complete and statistical profiles.

[5] The basic block is in the routine `cmd_exec()`, at the label `tail_recursion_entry`.

4.2.3 Key Matching

In an earlier paper by Wall [1990], *key matching* was used to compare complete profiles to estimated profiles from a static program analysis. In key matching, the basic blocks in a program are ordered according to their counts in the profile. Two orderings are then compared by computing a function $f(n)$, called the *score*, which is the number of basic blocks that are in the top n entries in both profiles. If the basic block orderings are the same for both profiles, then $f(n) = n$ for all values of n. When $f(n)$ is less than n, it indicates dissimilarity of the profiles. Figure 4 shows the key matching results for statistical profiles based on 32 sampled runs of our test workloads. Note that most of the benchmarks have key-matching results close to $f(n) = n$. The key matching test suggests that there is substantial similarity between the complete frequency-based profiles and the statistical time-based profiles.

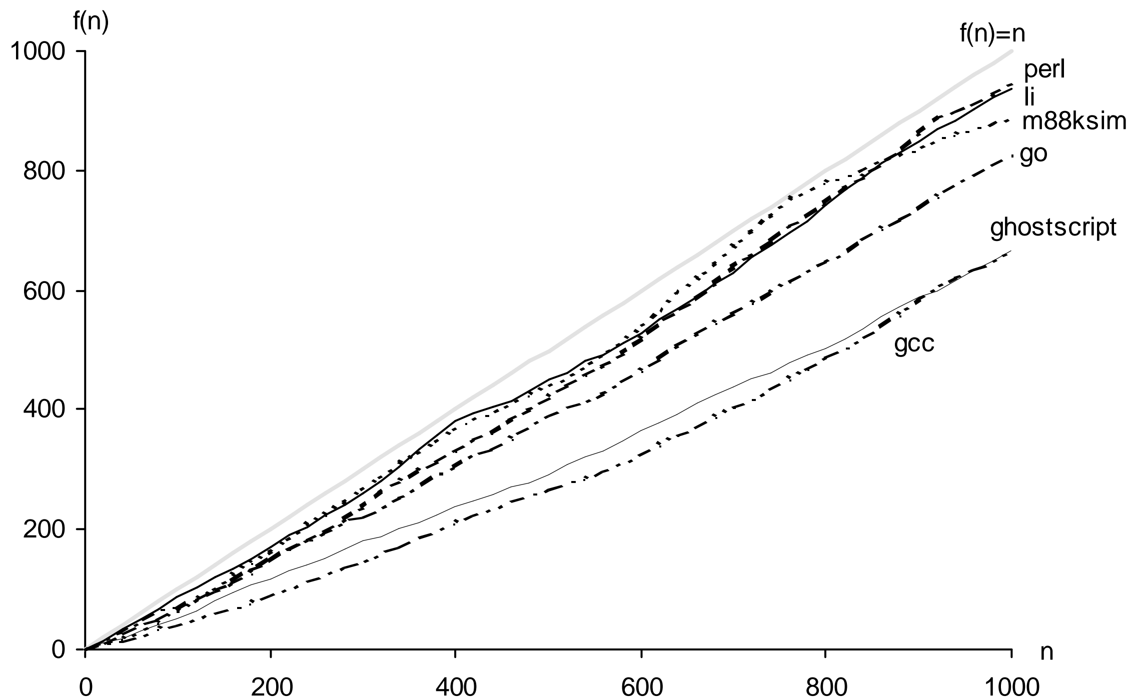


Figure 4: Key matching results.

4.3 Optimization with Statistical Profiles

As an end-to-end evaluation of the quality of our statistical profiles, we used the profiles to drive a code-layout optimization. We measured the effectiveness of the optimization by using an instruction-cache simulation and by measuring the execution time of optimized executables. We compared the execution time of each workload using four different procedure orderings: the original (default)

procedure order, an optimized procedure order derived from a complete frequency-based profile, and optimized orderings from statistical time-based profiles using one and 32 sample sets. For the optimization using a single sample set, there was significant variation in the effectiveness of optimization depending on which sample set was chosen. Rather than trying to identify a single representative run, we generated 32 optimized programs using each of the sample sets and tested each of them. We report maximum, minimum, and average behavior for these 32 separate experiments. All experiments used different inputs for training and testing.

Table 6 gives the cache simulation results. The improvement in miss rate achieved with statistical time-based profiles for 32 sampled runs is comparable to that of the complete profile; this is evidence of the effectiveness of optimization with statistical profiles. The statistical time-based profiles give superior performance to complete frequency-based profiles in four of six cases.

	no	1 sample set			32 sample	complete
Benchmark	optimizations	maximum	average	minimum	sets	profile
ghostscript	1.367%	1.981%	1.277%	0.912%	1.361%	1.081%
gcc	1.907%	2.201%	1.820%	1.557%	1.689%	1.757%
go	1.529%	0.712%	0.649%	0.570%	0.636%	0.702%
li	0.039%	0.100%	0.033%	0.000%	0.013%	0.167%
m88ksim	3.578%	0.055%	0.028%	0.004%	0.013%	0.050%
perl	1.061%	1.224%	0.641%	0.327%	0.680%	0.619%

Table 6: Simulated instruction cache miss rates. These results are for a 32KB direct-mapped cache with block size of 32 Bytes.

The results for a single sampled run demonstrate two points. First, for six of six benchmarks the average miss rates for a single sample set[6] is lower than the miss rate of the original program. This suggests that profile information from even a single sampled run is better than no information in the case of procedure-layout. A more interesting phenomenon is the wide range of miss rates across the 32 sample sets. Although the worst performance for one sampled run is often worse than the original program, in six of six cases at least one of the procedure layouts derived from a single sampled run provides the best performance of all measured cases. For three benchmarks the average is lower than the miss rate based on the complete profile. This suggests that time-based profiles can be more effective for optimization than complete profiles. Further evidence towards this conclusion is given in the next section.

In the case of *li,* the cache miss rate actually increases for the code layout derived from the complete profile. Investigation showed that this anomaly is due to situations where one procedure makes frequent calls to one or a number of other procedures, but the edges are omitted from the call graph due to a call through a procedure pointer. These missing edges can lead to poor choices for procedure layout. We are planning modifications to our monitor to address this problem by gathering information for calls through procedure pointers.

Benchmark	no optimizations	1 sample set maximum	1 sample set average	1 sample set minimum	32 sample sets	complete profile
ghostscript	86.2	93.3	81.1	75.0	78.6	77.7
gcc	28.2	31.3	29.2	28.1	29.8	28.9
go	364.4	385.7	361.9	344.9	344.6	358.9
li	65.7	64.1	63.3	62.7	63.2	63.8
m88ksim	338.1	314.4	266.6	241.2	255.4	261.1
perl	77.4	79.2	75.5	71.9	72.6	76.3

Table 7: Optimization results: execution time in seconds. These experiments were run in single-user mode with cold cache. All results are the average of five trials. In all cases standard deviation is less than 1%.

Table 7 gives optimization results in terms of execution time. The results for these experiments are similar to those for the simulation: statistical profiles yield comparable performance improvements to complete profiles, and in four of six cases the statistical profile using 32 sample sets provides a bigger performance improvement than the complete profile. For one benchmark, *gcc*, optimization with statistical profiles has a small negative impact on performance. Note that for this case, optimization with the complete profile also failed to give a performance improvement. A possible explanations to this failure is the heuristic nature of the code layout optimizations.

5. Related Work

Statistical profilers have long been a popular tool for performance tuning. The BSD UNIX *gprof* utility generates statistical execution profiles with the effect of called routines incorporated into the profile of each caller [Graham et al. 1982, Graham et al. 1983]. It uses compiler support to maintain call-graph information and statistical PC sampling to derive timing information. *GProf* uses the *profil* system call to collect profile samples. Arguments to *profil* specify how to divide the text segment of

[6] Computed by optimizing and running the given benchmark thirty-two times, each with a different sample set from a single sampled run.

the program into equal-sized PC ranges, with each entry in the array of shorts representing one PC range.

There are a number of problems with the *profil* interface when applied in current systems for continuous monitoring. As a single array must represent the entire range of PC values, *profil* does not immediately accommodate shared libraries. Because the *profil* interface collects information for fixed size PC ranges, it is less precise than the per-basic-block information that can be derived from our PC sample log. Use of *profil* requires that a tradeoff be made between profile resolution and buffer size. For a fixed resolution, the amount of memory required increases linearly with the text segment size. This makes it difficult to handle large text segments. The use of two-byte counters introduces the potential for overflow. At a sampling frequency of 1000Hz it takes a little over a minute of iterating on a single basic block to overflow a two-byte counter. Lastly, the memory locality of applying updates to a large array of counters could potentially be worse than recording PC samples to a log.

Wall [1990] explored the use of estimated profiles using static analysis only as an alternative to run-time profile collection. If the estimated profiles were of sufficient quality, then it might be possible to use estimated profiles in place of complete profiles to achieve many of the same goals as our low-overhead profile collection. Wall found that estimated profiles were not of sufficient quality to replace run-time profile collection.

In a related study, Ball and Larus [1993] examined heuristics branch prediction using static analysis. They found that it did an adequate job of predicting branch direction for a large and diverse collection of C and Fortran programs. It is not clear if their techniques could be applied to optimizations such as procedure layout. We are currently exploring this question.

Many instrumentation tools have been used to generate complete profiles for PBOs [Smith 1991, Srivastava and Eustace 1994, Wall 1992, Wall and Powell 1987]. Ball and Larus [1994] discuss efficient techniques for collecting complete profile data. A good review of instrumentation tools is given by Pierce et al. [1995]. Even with efficient techniques, the overhead of these tools is orders of magnitude greater than that of MCMS.

Conte et al. [1996] described an approach in which the operating system periodically samples the state of branch prediction hardware. They use the resulting profiles to drive a superblock scheduling optimization. Their work differs from ours in several important ways. It has higher run-time overhead, from 0.4 to 4.6%. It requires modifications to existing programs to facilitate interpretation

of the data from branch hardware. Lastly, their approach requires hardware modifications to enable the system to read branch prediction state.

Aside from performance debugging and PBO, we are aware of several projects where statistical profiles have been used to run cache simulations and other architectural studies [Menezes 1995].

6. Conclusions

Our experience with statistical profiles demonstrates that it is possible to collect profiles of comparable quality to complete profiles with negligible runtime overhead. Our profile system requires no modifications to the executable. Given these two characteristics, it becomes practical to collect profiles continuously for all activity in a computer system. Continuous monitoring holds the promise to significantly improve the quality and availability of training data for PBOs. The Morph Continuous Monitoring System is a key technology for automating the application of PBOs, which will allow the co-evolution of software with hardware.

References

[Ball and Larus 1993]
T. Ball, J. Larus, "Branch Prediction for Free," *Proceedings of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, June 1993. Also available as Technical Report #1137, Computer Sciences Department, University of Wisconsin - Madison.

[Ball and Larus 1994]
T. Ball and J. Larus. "Optimally Profiling and Tracing Programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1319-1360, July 1994.

[Chang et al. 1991]
P. P. Chang, S. A. Mahlke, and W.W. Hwu. "Using Profile Information to Assist Classic Code Optimizations," *Software Practice and Experience*, Dec. 1991, Vol. 21, No. 12, pp. 1301-1321.

[Chen et al. 1994]
W. Y. Chen, S. A. Mahlke, N. J. Warter, S. Anik, and W.W. Hwu. "Profile-Assisted Instruction Scheduling," *International Journal for Parallel Programming*, Vol. 22, No. 2, April 1994, pp. 151-181.

[Conte et al. 1996]
T.M. Conte, B.A. Patel, K. Menezes, and J.S. Cox, "Hardware-Based Profiling: An Effective Technique for Profile-Driven Optimization," *International Journal of Parallel Programming*, Vol 24, February 1996.

[Digital 1996]
Digital Semiconductor, *Digital Semiconductor 21164 (366 MHz Through 433 MHz) Alpha Microprocessor Reference Manual, Hardware Reference Manual*, Digital Equipment Corporation, Maynard MA, Order Number EC-QP99A-TE, March 1996.

[Fisher et al. 1984]
J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau. "Parallel Processing: A Smart Compiler and a Dumb Machine," *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pp. 37-47. 1984.

[Fisher and Freudenberger 1992]
J. A. Fisher and S. M. Freudenberger. "Predicting Conditional Branch Directions from Previous Runs of a Program," *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 85-97, 1992.

[Graham et al. 1982]
S. Graham, P. Kessler, and M. McKusick. "gprof: A Call Graph Execution Profiler," *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pp. 120-126, June 1982.

[Graham et al. 1983]
S. Graham, P. Kessler, and M. McKusick. "An Execution Profiler for Modular Programs," *Software - Practice and Experience*, 13, pp. 671-685, 1983.

[Intel 1995]
Intel Corporation, *Pentium Processor Family Developer's Manual. Volume 3: Architecture and Programming Manual*, Intel Corporation, Order Number 241430-004, 1995.

[Intel 1996]
Intel Corporation, *Pentium Pro Family Developer's Manual. Volume 3: Operating System Writer's Manual*, Intel Corporation, Order Number 242692-001, 1996.

[Lindgren 1976]
B. Lindgren, *Statistical Theory*, Third Edition, Macmillan Publishing Company, New York, 1976.

[McFarling and Hennessy 1986]
S. McFarling and J. Hennessy. "Reducing the cost of Branches," *Proceedings of the 13th International Annual Symposium on Computer Architecture*, pp. 369-403, June 1986.

[McFarling 1989]
S. McFarling. “Program optimization for instruction caches,” Proceedings of the Third International Symposium on Architectural Support for Programming Languages and Operating Systems, pp. 183-191, April 1989.

[Menezes 1995]
K. Menezes. “Sampling for Cache and Processor Simulation,” *Fast Simulation of Computer Architectures*, edited by T. Conte and C. Gimarc, Kluwer Academic Publishers, Boston, MA, pp. 171-204, 1995.

[Pierce et al. 1995]
J. Pierce, M. Smith, and T. Mudge. “Instrumentation Tools,” *Fast Simulation of Computer Architectures*, edited by T. Conte and C. Gimarc, Kluwer Academic Publishers, Boston, MA, pp. 47-86, 1995.

[Pettis and Hansen 1990]
K. Pettis and R. Hansen. “Profile Guided Code Positioning,” Proceedings of SIGPLAN `90 Conference on Programming Language Design and Implementation, pp. 16-27, June 1990.

[Santhanam and Odnert 1990]
V. Santhanam and D. Odnert. “Register allocation across procedure and module boundaries.” *Proceedings of the SIGPLAN ’90 Conference on Programming Language Design and Implementation*, pp. 28-39, 1990.

[Seagate 1994]
Seagate Technology, Inc., Barracuda 2LP Disc Drive ST32250W/WD/WC Installation Guide, Publication Number: 83328920, Revision B, July 1994.

[Smith 1991]
M. Smith. Tracing with Pixie, Technical Report CSL-TR-91-497, Stanford University, November 1991.

[Srivastava and Eustace 1994]
A. Srivastava and A. Eustace. “ATOM: A System for Building Customized Program Analysis Tools,” *Proceedings of the SIGPLAN 1994 Conference on Programming Language Design and Implementation*, 1994. See also DEC WRL Research Report 94/2.

[Wall 1986]
D. Wall. “Global register allocation at link-time,” *Proceedings of the SIGPLAN ’86 Symposium on Compiler Construction*, pp. 264-275, 1986.

[Wall and Powell 1987]
D. Wall and M. Powell. “The Mahler Experience: Using an Intermediate Language as the Machine Description,” *Second International Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 100-104, 1987. See also DEC WRL Technical Report 87/1.

[Wall 1990]

D. Wall. Predicting Program Behavior Using Real or Estimated Profiles, DEC WRL Technical Note TN-18, December 1990.

[Wall 1992]

D. Wall. "Systems for Late Code Modification," *In Code Generation --- Concepts, Tools, Techniques*, Springer-Verlag, pp. 275-293, 1992.