EFFECTIVENESS OF SAMPLING BASED SOFTWARE PROFILERS

K. V. Subramaniam and Matthew J. Thazhuthaveetil
Computer Science and Automation,
Indian Institute of Science,
Bangalore 560012, INDIA
Email: {kvs,mjt}@csa.iisc.ernet.in

Abstract

Profilers are a class of program monitoring tools which aid in tuning performance. Profiling tools which employ PC (Program Counter) sampling to monitor program dynamics are popular owing to their low overhead. However, the effectiveness of such profilers depends on the accuracy and completeness of the measurements made. We study the methodology and effectiveness of PC sampling based software profilers.

I. Introduction

Profilers are software tools that provide the programmer with estimates of how much execution time is spent in the various components of his program. Commonly used profilers are known to suffer from deficiencies. We evaluate the effectiveness of profilers based on their **Accuracy** (reflecting the closeness of the measurement made by the profiler to the actual case), and **Variation** (across profiling runs; a good profiler should show little or no variation in measurements) [1]. In this paper we study profilers based on PC (Program Counter) sampling, examples of which include Unix `prof` and Borland Turbo Profiler [2] operating in the passive mode. In the remainder of this paper, the term `prof` is used for any profiler that uses this profiling technique. In this paper, we study `prof` inaccuracy and its causes, and suggest a technique to enhance accuracy.

II. Motivation

The accuracy of the profile generated by a profiler is often important. To a programmer using a profiler as a decision making tool in optimizing portions of a program, wasted effort could result from such inaccuracies. Execution profilers such as `prof` are expected to provide accurate measurements at a low time overhead. `prof` associates an array of counters with the program to be profiled. The program itself is viewed as a sequence of equal sized chunks (typically 8 bytes of profiled code), and one counter is associated with each such chunk. On every clock tick during the execution of the program being profiled, the counter associated with the sampled PC value is incremented. This activity is initiated by the timer interrupt handler, for programs that have been compiled for profiling. The counters are later used to estimate how the total execution time of the program was spent in its various subprograms [3].

Weinberger [3] suggests the following sources of inaccuracies in the profiles generated by `prof`: (i) One or more of the counters corresponding to (8 byte) chunks of code may straddle two subprograms, in which case the counts may get attributed to the wrong subprogram. (ii) If event lifetimes are much smaller than the sampling period, a short lived subprogram may never be sampled. (iii) A resonance or beating effect could occur when the sampling period and the time taken to execute a loop are identical. Ponder and Fateman [4] also identify the resonance effect as a potential source of inaccuracies in `prof` in their analysis of inaccuracies in `gprof`, but argue that such instances of resonance are uncommon in real programs. The other two problems cannot be ignored as easily. In the presence of these and other possible sources of error, a profile generated by `prof` can be viewed as providing only a rough estimate of program behaviour. The accuracy of the measurement could possibly be increased by doing more profiling runs. This prompts the following questions: *(i) How approximate are the measurements made by* `prof`*? (ii) Can these approximate measurements also be inaccurate? (iii) How many profiling runs are required to arrive at an accurate estimate of program behaviour?*

To illustrate the degree of inaccuracy, we conducted the following experiment: Five profiling runs of dhrystone[1] were done on the same machine under identical workload conditions. The results of the experiments are shown in Table I. Observe that even after 5 profiling runs, the most frequently executed subprograms cannot be positively identified. We conclude that measurements made by `prof` can be quite inaccurate, and question the rationale of performing multiple profiling runs - with each run contributing a 100% profiling overhead - just to arrive at a "rough" estimate of program locality. There is clearly a need to better understand the causes of these inaccuracies.

[1] 20,000 iterations of the synthetic benchmark Dhrystone [5]

0-7803-2608-3

TABLE I
prof results of top 6 procedures on 5 successive runs of dhrystone - 20,000 iterations

%t	Proc	%t	Proc	%t	Proc	%t	Proc	%t	Proc
17.8	main	16.6	strcpy	14.0	Proc_1	21.1	main	16.6	Proc_1
16.1	strcpy	15.4	main	12.8	main	15.4	Proc_1	16.6	main
13.2	Proc_1	12.6	Proc_1	11.0	Proc_8	11.4	strcpy	13.0	strcpy
8.6	strcmp	12.0	Proc_8	9.9	strcpy	10.3	strcmp	11.8	Func_2
8.6	Proc_8	6.9	Func_1	8.7	Func_2	7.4	Proc_8	8.9	strcmp
7.5	_write	6.3	_write	7.6	strcmp	6.9	_write	6.5	Proc_8

III. Causes of Inaccuracy and Variation

We differentiate between *profile inaccuracy* and *variation across profiling runs*. The inaccuracy of any one profiling run is the difference between the profile it generates and the actual distribution of cycles among the subprograms of the program. For a given input data set, a program should follow the same execution path in different runs. Ideally, we would expect neither the total number of cycles taken by the program, nor the distribution of these cycles across subprograms, to vary across these runs. Therefore, given that the the prof sampling interrupt occurs at fixed, periodic intervals during the execution of a profiled program, profiles should not vary across profiling runs.

We identify the following as causes of profile inaccuracy:

Beating Effect:. As mentioned earlier, it is possible that the sampling interval and the time taken to execute a loop are identical. As a result, an inordinate proportion of the PC samples taken might be to a particular subprogram.

Granularity of Sampling:. Consider a system operating at 12 Mhz, with the clock tick (timer interrupt) occurring every 10 ms. For a processor that can execute one instruction per cycle, about 120,000 instructions would be executed between two PC samples. Many subprograms might have single invocation instruction path lengths that are much shorter than this. Such short lived events occurring between sampling interrupts might escape capture altogether. Figure 1 illustrates this problem.

Profile variations across multiple runs can be attributed to external events – events caused by other activities executing on the system, resulting in changes to the effective sampling interval. This background load on the system can affect the sampling interval in the following ways:

Processor Pre-emption:. I/O activity on the system generates interrupts that are serviced at high processor priority, pre-empting the execution of the profiled process. From the perspective of the profiled process, this has the effect of changing the sampling interval, as illustrated in Figure 2. The fluctuation in sampling interval is a function of the interrupt frequency, which is related to the load on the system.

Cache Effect:. At the end of a CPU quantum, when another process gets scheduled instead of the profiled process, the cache context built up by the profiled process is replaced by that of newly scheduled process. Thus, when the profiled process is re-scheduled, its cache context must be reconstructed. The time taken for the profiled process to rebuild it's cache context will depend on (a) how much of the cache is occupied by a process during one quantum, and (b) the background load, which characterizes the number of quanta after which the profiled process is scheduled again.

Having identified the possible causes of error and variation in results reported by prof, we studied the effect of each through trace driven simulation.

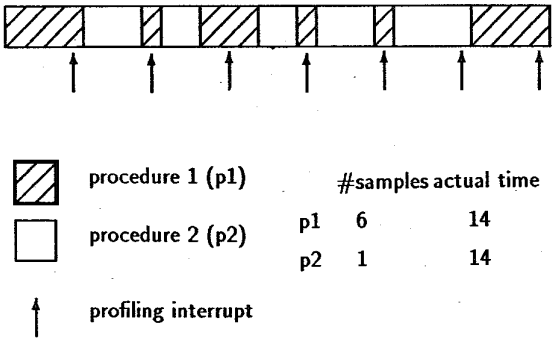

Fig. 1. Error in prof results due to large sampling interval

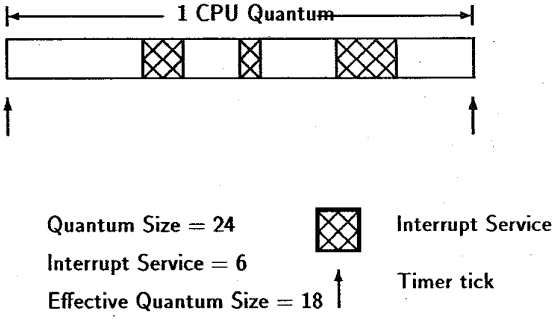

Fig. 2. Reduction in the sampling interval due to interrupts generated by background load

IV. Simulation

In our simulations, we assumed a similar computing environment to that on which the experiment described earlier was conducted – a Unix workstation based on the MIPS 2000 processor. The simulations were driven by traces generated on that system using `pixie` [6] with the `-idtrace` option. We used the traces of 20,000 iterations of the synthetic benchmark dhrystone[2] (referred to as dhry), the SPICE benchmark from the Perfect benchmark suite (referred to as css) [7], and the Monte Carlo simulation benchmark QCD2 from the Perfect benchmark suite (referred to as lgs). Each trace file was approximately 40MB in size, corresponding to 15-20 million instructions executed.

Simulation Model

System modelling:. We assume that the processor operates with a 12MHz clock, with an 8KB direct mapped data cache and a 16KB direct mapped instruction cache. The line size of the data cache is 4 bytes, and there is a cache miss penalty of 5 cycles. We assume that the same parameters hold for the instruction cache as well. There is a 4 cycle stall between successive memory store instructions. A context switch is associated with every profiling interrupt. This assumption was made after we observed that the `getrusage()` report for the number of context switches during a profiling run is identical to the number of samples collected by `prof`. The prof sampling period is 10 ms. We assume an overhead of 4500 cycles for every sampling period. This overhead can be attributed to the execution of the profiling interrupt handler and the context switch that occurs at the end of the sampling period.

System workload modelling:. The number of processes contending for the CPU has a significant impact on the measurements made by `prof`. Two factors that are effected by increases in load are (i) increased loss of cache context, and (ii) increased interrupt handling overheads. Other events, such as page fault handling, are assumed not to affect the effective sampling period; we assume that they are not accounted against the profiled process even if they occur in its CPU quanta. System workload is therefore modelled using two parameters:

1. Amount of interrupt activity: The number of interrupts occurring in a quantum is assumed to increase linearly with background load, reducing the effective length of the CPU quantum proportionately. We assume a loss of 10% of quantum size on a fully loaded system due to the interrupt activity caused by the background load. We assume that the load conditions are maintained at a constant level through the profiling run.
2. Loss in Cache context: The effect of context switching on cache context has been studied by Mogul *et al* [8] and others. We quantify the loss in cache context with increasing load as follows: The number of lines lost by the profiled process on a context switch is assumed to increase exponentially with the increase in background load. As both the Instruction Cache and the Data Cache are assumed to be small, we assume that the entire cache context is lost on a fully loaded machine [9]. However, on a system with no load, the profiled process is the only process on the system, and so the loss in cache context is only due to the execution of the profile interrupt handler and the context switching code. We consider only the effects of the profile interrupt handler. The loss in cache context due to the execution of the interrupt handler is viewed as a loss of a fixed set of lines in the I Cache (due to execution of the handler code) and a loss of a set of lines from the D Cache. The address of the lost lines in the D Cache depends on the address of the currently executing instruction and the loss can be attributed to the incrementing of the profile counters by the handler.

All of our simulation assumptions were validated through experiments performed under controlled conditions [10].

[2] All `printf()` statements are removed from the dhrystone code to eliminate terminal I/O

V. Results

We quantify the accuracy of a profile relative to the distribution of cycles among the subprograms of the program under certain ideal conditions, which we refer to as the *true histogram*. The true histogram is defined as the normalized number of cycles per subprograms of the profiled program executing under no load. Under these conditions, the profiled process is the only process in execution. The true histogram is estimated in the absence of operating system overheads due to context switching and profiling. We computed the true histograms for each of our benchmark program traces through simulation. The inaccuracy of a measured profile was then quantified as its deviation from the true histogram. We use the chi square measure to quantify the difference between two histograms.

Simulations of profiled execution were carried out under no load, half load and full load conditions, with load characterized by interrupt activity and/or loss in cache context. Figure 3 compares the true histogram of css and the simulation results under no load conditions. The chi square values showing the error in the results reported by `prof` under no load conditions for each of the application traces is shown in column 1 of Table III. Table II compares the measurements made by `prof` under varying load conditions using the chi square measure.

From Figure 3, it is clear that the error is not due to sampling errors in just one or two subprograms, but across many subprograms, eliminating the resonance phenomenon as source of error in this case. It appears that the likely cause of the errors is the fact that the sampling interval is too large to capture all events. To study this hypothesis, we re-ran the simulations with an initial skew of a few cycles, i.e., by shifting all the samples by a few cycles. We would expect this shift to have little effect on the measurements made, but the simulation results indicate otherwise. Table IV shows the chi-square measure between the skewed

TABLE II
chi square measure showing variation on the measurements made by prof under varying loads

load characterization	load	dhry	css	lgs
-	no	0.0	0.0	0.0
only cache effects	half	1.79 e6	7.81 e6	6.13 e5
	full	8.83 e6	5.02 e6	8.85 e5
only interrupts effects	half	5.48 e6	4.93 e6	7.63 e5
	full	1.30 e8	8.14 e6	1.61 e6
both cache and interrupt effects	half	2.21 e6	6.27 e6	6.81 e5
	full	6.90 e6	7.39 e6	1.92 e6

TABLE III
Chi square measure between the true histogram and simulation results for the prof interval of 10ms and the program dependent profiling frequency

Application	Interval 10ms	Interval $C_{Modified}$
dhry	5.72 e06	5.00 e06
css	4.50 e06	2.61 e05
lgs	9.01 e05	2.30 e06

TABLE IV
Chi square measure showing variation in prof results with load and skew using a fixed profiling interval of 10ms

Benchmark Name	Skew (in cycles)	No Load	Half Load 10% Loss	Full Load 10% Loss
dhry	0	0.0	2.2 e06	6.9 e06
	10	11.2 e06	3.1 e06	7.3 e06
css	0	0.0	6.2 e06	7.3 e06
	10	3.1 e06	6.7 e06	7.3 e06
lgs	0	0.0	0.6 e06	1.9 e06
	10	0.2 e06	0.6 e06	1.9 e06

trace under different load characterizations and prof measurements under no load conditions.

We conclude that to improve the accuracy of measurements made by prof, it is necessary to use a sampling interval that is smaller than the default interval of 10ms. Further, the choice of profiling interval should be program dependent.

VI. Modification to prof

In this section we study the efficacy of of a program dependent profiling interval as a means of improving the accuracy of the measurements made by prof. Let us view the program in execution in terms of the variation in PC value with time. We assume that the program unit of interest is the subprogram, and represent each subprogram by it entry PC value. Using considerations similar to the Sampling Theorem, the ideal sampling interval would then be :

$$C_{Ideal} = C_{Min}/2$$

where

C_{Ideal} is the ideal profiling interval and

C_{Min} is the minimum number of cycles between subprogram changes.

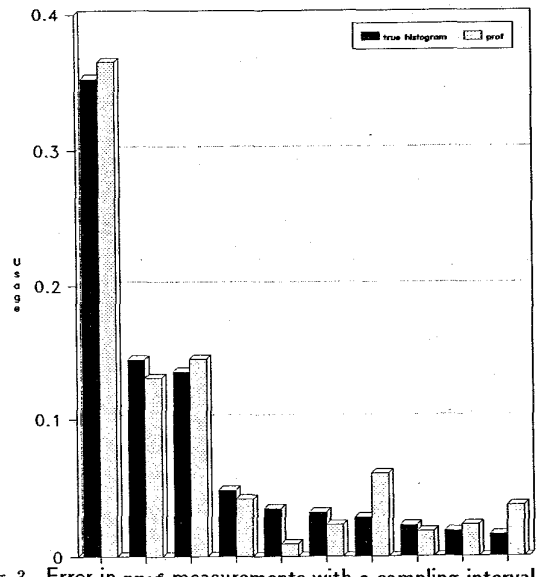

Fig. 3. Error in prof measurements with a sampling interval of 120,000 cycles

Profiling at this frequency should guarantee accurate results. However, for all three of our traces, the minimum number of cycles between subprogram changes is 2 cycles, yielding an ideal sampling interval of 1 cycle, which is equivalent to single stepping through the program! If this is caused by a routine with a low execution count, we could use a larger profiling interval without much loss in accuracy. We arrive at such a practical sampling interval using the average number of cycles between subprogram changes, calculated as

$$C_{Modified} = C_{Avg}/2$$

where:

$C_{Modified}$ is the modified profiling interval and

C_{Avg} is the Average number of cycles between subprogram changes.

The modified sampling intervals were found to be 12 cycles for dhry, 53 cycles for css and 78 cycles for lgs. Simulations of profiling under these sampling intervals were then conducted. Since the new sampling interval is orders of magnitude smaller than the default value used earlier, we modified our simulation assumptions as follows: (i) We no longer assume that a context switch will occur at the end of every sampling interval. Under such an assumption, with a small sampling interval, many cycles will be wasted in undoing the effects of the context switch. We therefore assume that a context switch occurs every 30 ms, the default CPU quantum for a process on our Unix system. (ii) We assume that the overhead associated with profiling is 50 cycles. Earlier, both the profiling handler and the context switch code were executed simultaneously, so it is possible that the profile interrupt handler was executing some extra code. Based on an analysis of the activity involved in profiling, we estimate that a carefully hand coded profiling interrupt handler should be able execute in 50 cycles.

Results with modified sampling interval

Figure 4 compares the true histogram of css with the measurements made by `prof` under no load with the modified sampling interval. Unfortunately, the behaviour illustrated in this figures was shown only by the css benchmark. A comparison of the chi square measure for modified sampling interval against the fixed interval under no load conditions is shown in Table III. While there is a significant reduction in the error for css, and a marginal reduction for dhry, the error for the lgs program actually increased with higher sampling frequency. A similar comparison of the variation in `prof` results with load (see Table V) indicates a drop in the variation in the case of lgs, but wide variation for css and dhry.

We also studied the affect of initial skew in the variation in measurements, under different load conditions The chi-square measures for the three benchmarks are tabulated in Table VI. We conclude that the modified sampling interval is capable of capturing most events, but does *not* guarantee more accurate measurements than those obtained with the default sampling interval. Hence, to guarantee accurate measurements using `prof`, we have no choice but to use the ideal sampling interval - even if it implies interrupting program execution every cycle, which amounts to single stepping through the execution of the program.

VII. Conclusion

While program profilers like Unix `prof` are widely used as program tuning tools, their effectiveness in capturing actual program behaviour is not well understood. We have studied the interaction between the profiling process and the processor state for a typical RISC based Unix system. We conclude that PC sampling profilers which use a fixed sampling interval, like Unix `prof`, can not be relied on to produce accurate measurements. Although this makes them unsuitable as architecture analysis tools, if used with multiple profiling runs, they can provide useful information to programmers interested in identifying program bottlenecks. PC sample based profiling using program dependent sampling intervals maybe capable of providing accurate measurements, provided the ideal sampling interval is used. This carries a high time overhead with it; in the cases studied, accurate profiling would involve single stepping through program execution. A less restrictive sampling interval was also studied, but could not guarantee accurate measurements.

TABLE V

chi square measure showing variation in results reported by `prof` with load using a program dependent sampling interval

load characterization	load	dhry	css	lgs
-	noload	0.0	0.0	0.0
only cache effects	half load	227.5	301.0	440.0
	full load	423.9	417.7	216.0
only interrupts effects	half load	0.4	52.6	367.7
	full load	0.2	24.1	262.6
both cache and interrupt effects	half load	248.9	270.6	261.9
	full load	640.2	462.0	348.2

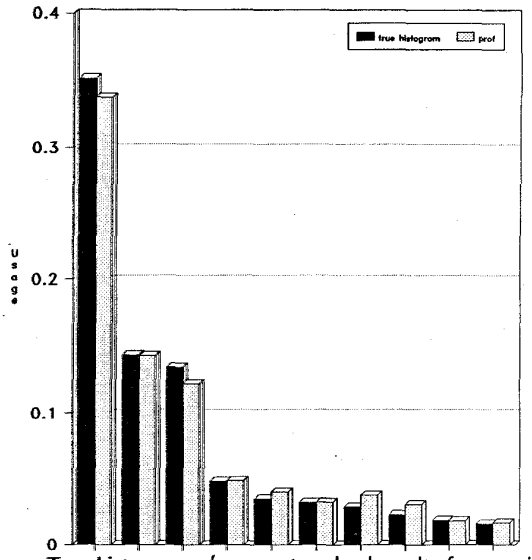

Fig. 4. True histogram v/s `prof` at no load results for css with modified profiling interval

TABLE VI

Chi square measure showing variation in `prof` with load and skew using a program dependent profiling interval

Benchmark Name	Skew (in cycles)	No Load	Half Load 10% Loss	Full Load 10% Loss
dhry	0	0.0	248.9	289.0
	10	3.0	251.9	292.7
css	0	0.0	270.6	356.7
	10	11.1	301.1	289.5
lgs	0	0.0	261.9	200.1
	10	1.9	263.8	214.7

References

[1] C.B. Stunkel, B. Janssens, and W.K. Fuchs. Address Tracing for Parallel Machines. *IEEE Computer*, pp 31–38, Jan 1991.

[2] Borland International Inc. *Turbo Profiler Version 1.0 User's Guide*, 1990.

[3] P.J. Weinberger. Cheap Dynamic Instruction Counting. *AT&T Bell Lab Tech Journal*, 63(8):1815–1826, Oct 1984.

[4] C. Ponder and R. J. Fateman. Inaccuracies in Program Profilers. *Software - Practice and Experience*, 18(5):459–467, May 1988.

[5] R.P. Weicker. An Overview of Common Benchmarks. *IEEE Computer*, 1990.

[6] M.D. Smith. Tracing with pixie. Tech Report CSL-TR-91-497, Computer Systems Lab, Stanford University, Nov 1991.

[7] G. Cybenko, L. Kipp, L. Pointer, and D. Kuck. Supercomputer Performance Evaluation and the Perfect Benchmarks. In *Proc Intl Conf on Supercomputing*, pp 254–266, 1990.

[8] J.C. Mogul and A. Borg. The Effect of Context Switches on Cache Performance. In *Proc 4th ACM/IEEE Conf on Arch Support for Prog Lang and Operating Systems*, pp 75–84, 1991.

[9] S. Laha, J.H. Patel, and R.K. Iyer. Accurate Low Cost Methods for Performance Evaluation of Cache Memory Systems. *IEEE Trans Comp*, pp 1325–1336, Nov 1988.

[10] K. V. Subramaniam. Monitoring Program Dynamics: An Evaluation of Software Profiler Effectiveness. Master's thesis, Computer Science and Automation, Indian Institute of Science, Nov 1993.