Overhead Compensation in Performance Profiling

Allen D. Malony and Sameer S. Shende

Performance Research Laboratory, Department of Computer and Information Science
University of Oregon, Eugene, OR, USA
{malony,sameer}@cs.uoregon.edu

Abstract. Measurement-based profiling introduces intrusion in program execution. Intrusion effects can be mitigated by compensating for measurement overhead. Techniques for compensation analysis in performance profiling are presented and their implementation in the TAU performance system described. Experimental results on the NAS parallel benchmarks demonstrate that overhead compensation can be effective in improving the accuracy of performance profiling.

Keywords: Performance measurement and analysis, parallel computing, profiling, intrusion, overhead compensation.

1 Introduction

Profiling and tracing are the two main approaches for empirical parallel performance analysis. Parallel tracing is most often implemented as a measurement-based technique. Here, the application code is instrumented to observe *events* which are recorded in a trace buffer on each occurrence during execution [5, 28]. In contrast, performance profiling [16] can be implemented in one of two ways: 1) *in vivo*, with measurement code inserted in the program (e.g., see [6, 9, 21, 23, 24]), or 2) *ex vivo*, by periodically interrupting the program to assign performance metrics to code regions identified by the halted program counter (e.g., see [10, 13, 14, 22, 26]. The first technique is commonly referred to as *measurement-based profiling* (or simply *measured profiling*) and is an *active* technique. The second technique is called *sample-based profiling* (also known as *statistical profiling*) and is a *passive* technique since it requires little or no modification to program.

There are significant differences of opinion among performance tool researchers with regards to the merits of measured versus statistical profiling. The issues debated include instrumentation, robustness, portability, compiler optimizations, and intrusion. Ultimately, the profiling methods must be *accurate*, else the differences of opinion really do not matter. Herein lies an interesting performance analysis conundrum. How do we evaluate analysis accuracy when the "true" performance is unknown? Some technique must be used to observe performance, and profiling tools will always have limitations on what performance phenomena can and cannot be observed [17]. Is a tool inaccurate if it does not

M. Danelutto, D. Laforenza, M. Vanneschi (Eds.): Euro-Par 2004, LNCS 3149, pp. 119–132, 2004.

provide information about particular performance behavior at a sufficient level of detail? Furthermore, no tool is entirely passive, and any degree of execution intrusion can result in performance perturbation [17]. Should not all profiling tools be considered inaccurate in this case? Parallel performance analysis is no different from experimental methods in other sciences. "Truth" always lies just beyond the reach of observation. As long as this is the case, accuracy will be a relative assessment.

Until there is a systematic basis for judging the accuracy of profiling tools, it is more productive to focus on those challenges that a profiling method faces to improve its accuracy. Our work advocates measured profiling as a method of choice for performance analysis [21]. Unfortunately, measured profiling suffers from direct intrusion on program execution. This intrusion is often reported as a percentage slowdown of total execution time, but the intrusion effects will be distributed throughout the profile results. The question we pose in this paper is whether it is possible to compensate for these effects by quantifying and removing the overhead from profile measurements.

Section §2 describes the problem of profiling intrusion and outlines our compensation objectives. The algorithms for overhead removal are described in Section §3. We tested these on a series of case studies using the NAS parallel benchmarks [1]. In Section §4, we report our findings with an emphasis on evaluating relative accuracy. Our approach can improve intrusion errors, but it does not fully solve the overhead compensation problem. Section §5 discusses the thorny issues that remain. Conclusions and future work are given in Section §6.

2 Measured Profiling and Intrusion

Profiling associates performance metrics with aspects of a program's execution. Normally, it is the program's components being profiled, such as its routines and code blocks, and profile results show how performance data are distributed across these program parts. Execution time is the most common metric, but any source of performance data is valid (e.g., hardware counters [4, 6, 23, 30]). *Flat profiles* show performance data distributed onto the static program structure, while *path profiles* [11] map performance data to dynamic program behavior, most often represented as program execution paths (e.g., *routine calling paths* [7, 11]). Profiling both measures performance data and calculates performance statistics at runtime.

Measured profiling requires direct instrumentation of a program with code to obtain requisite performance data and compute performance statistics. Generally, we are interested in observing *events* that have *entry / exit* semantics. Instrumentation is done both at the event "entry" point (e.g., routine begin) and the event "exit" point (e.g., routine return). Profile statistics report performance metrics observed between event entry and exit. We typically speak of *inclusive* performance, which includes the performance of other descendant events that occur between the entry and exit of a particular event, as well as of *exclusive* performance, which does not include descendant performance.

Unfortunately, the measurement code alters the program's execution, primarily in execution time dilation, but it also affects hardware operation. Intrusion will be reflected in the profile results unless techniques can compensate for it. Other research work has sought to characterize measurement overhead as a way to bound intrusion effects or to control the degree of intrusion during execution. For instance, the work by Kranzlmüller [15] quantifies the overhead of MPI monitors using the benchmarking suite SKaMPI, and Fagot's work [8] assesses systematically the overhead of parallel program tracing. The work by Hollingsworth and Miller [12] demonstrates the use of measurement cost models, both predicted and observed, to control intrusion at runtime via measurement throttling and instrumentation disabling.

Our interest is to compensate for measurement intrusion. Thus, we need to understand the intrusion effects are manifested in performance profiles. Let us first consider inclusive execution time profiles, using routines as our generic events. Two timing measurements are necessary to determine inclusive time for a routine every time it is called: one to get the current clock value at time of entry and one to get the clock value at exit. The difference between the clock samples is the inclusive time spent in this call. Let Δ_i represent the overhead to measure the inclusive time. If the routine A is executed N times, its inclusive time is increased by $N * \Delta_i$ measurement overhead.

However, A's inclusive time is also increased by the overhead incurred in the inclusive time measurement of descendant routines invoked after A is called and before it exits. If M routines are called while A is active (M covers all calls to A), the inclusive time is increased additionally by $M * \Delta_i$ because each descendant routine profiled will incur inclusive overhead.

To calculate A's exclusive time, the inclusive time spent in each direct descendant is subtracted from A's inclusive time. Since this occurs at A's exit, the inclusive times for all direct descendant calls must be summed. If Δ_e is the measurement overhead to do the summation upon each direct descendant exit, the total overhead that gets reflected in A's exclusive time is $L * \Delta_e$ for L direct descendant calls. (The subtraction from A's inclusive time is small and will be ignored for sake of discussion.) It is important to observe that all overhead accumulated in the inclusive times of A's direct descendants gets cancelled in the exclusive time computation. Unfortunately, if we calculate the inclusive and exclusive times simultaneously, as is normally the case, the $L * \Delta_e$ overhead must be added to A's inclusive time.

To summarize, the total performance profile of routine A will show an increase in inclusive time due to measurement overhead of $N * \Delta_i + M * \Delta_i + L * \Delta_e$, where N is the total number of times A is called, M is the number of times descendant routines of A are called, and L is the number of times A's direct descendant routines are called. The performance profile of routine A will also show an increase in exclusive time of $L * \Delta_e$ due to overhead.

Standard profiling practice does not report the measurement overheads included in per event inclusive and exclusive profiles. Typically, overhead is reported as a percetage slowdown in total execution time, with the implicit as-

sumption that measurement overhead is equally distributed across all measured events. Clearly, this is not the case. Events with a large number of descendant event occurrences will assume a greater proportion of measurement overhead. If we cannot compensate for this overhead, the performance profile will be distorted as a result. Furthermore, if processes of a parallel application exhibit different execution behavior, the parallel profiles will show skewed performance results.

3 Overhead Analysis and Compensation

The above formulation allows us to quantify the measurement overheads that occur in inclusive and exclusive times as a result of measured parallel profiling. If we were to actually compute these overheads, it must be done at runtime because the overheads depend on the dynamic calling behavior. However, our goal is to additionally *remove* the measurement overhead incurred in parallel profiling. Thus, the profiling system must both track the overhead and compensate for it dynamically in profile computations.

As a first step, we must determine the values Δ_i and Δ_e. These overhead units will certainly depend on the machine and compilers used, but could also be influened by the application code, the languages and libraries, how the application is linked, and so on. Also, it should not be assumed that Δ_i and Δ_e are constant. In fact, they may change from run to run, or even within a run. This means that we must measure the overhead values at the time the application executes, and even then the values will be approximate. The approach we propose is to conduct overhead experiments at application startup.

3.1 Overhead Analysis

To ascertain the overhead values Δ_i and Δ_e, we must measure the instrumentation code that calculates inclusive and exclusive performance. However, the Δ_i and Δ_e values may be at the same scale as the measurement precision. Thus, we must construct a test that guarantees statistical accuracy in our estimation of Δ_i and Δ_e.

Our approach is to conduct a two-level experiment where the inclusive measurement code is executed k number of times (in a tight loop) and the performance data being profiled is measured and stored. This procedure is then repeated j times and the minimum performance value across the j experiments is retained. We use that value divided by k to compute Δ_i. While this does not entirely insure against anomalous timing artifacts, its statistical safety can be improved by increasing j. The two-level experiment is then repeated, this time with the exclusive measurement code included, resulting in an approximation of $\Delta_i + \Delta_e$. We can substract our Δ_i approximation to find Δ_e. It is important to note that we need to do this overhead evaluation at the beginning of the application execution for the specific set of profile performance data being measured[1].

[1] In this regard, Δ_i and Δ_e are really vectors of overhead values.

3.2 Overhead Compensation

Given the approximations for Δ_i and Δ_e, we are ready to apply overhead compensation. There are several ways to go about it. One way is to use the formulas above and remove the overhead at the end of the execution. To do so, however, we must determine the variables N, M, and L for every event. Without going into details, calculating N, M, and L amounts to maintaining a dynamic call graph for every currently active event as the root of its own call tree. Knowing this allows us to consider a second way that removes inclusive overhead on-the-fly with every event exit. In this case, we need to only determine m_i and l_i values for each ith event occurrence ($M = \sum m_i$ and $L = \sum l_i$, for $1 \leq i \leq N$). Since we must calculate profile values at event exit, it is reasonable to have these be compensated calculations.

Using compensated inclusive calculations at event exit, we now have a choice for calculating compensated exclusive profile values. Without loss of generality with respect to other performance metrics, consider only execution time. The exclusive time for an event A is the difference between A's inclusive time and the sum of the inclusive times of all of A's direct descendants. Regardless of compensated or uncompensated inclusive times, we have to accummulate the inclusive times of A's direct descendants. However, for uncompensated inclusive times, an additional subtraction at A's exit of $l_i * \delta_e$ is needed to calculate the exclusive profile time for this invocation of A. Thus, the scheme we advocate for on-the-fly overhead compensation of exclusive time is to subtract the compensated inclusive times of A's direct descendants (as they exit) from A's running exclusive time, and add A's compensated inclusive time back in when A finally returns.

4 Experiments with Compensation Analysis in TAU

To evaluate the efficacy of overhead compensation, we must implement the methods described above in a real parallel profiling system and demonstrate their ability to improve application profiling results. To this end, we have implemented the overhead compensation techniques in the TAU parallel performance system [21]. TAU uses measured profiling to generate both flat profiles and callpath profiles. Inclusive and exclusive overhead compensation is implemented in TAU for both profiling modes.

How will we know if our overhead compensation works successfully? The standard measure for intrusion error is usually given as a percentage slowdown in total execution time. Thus, one can test the ability of a compensation-enabled profiling tool to accurately recover total execution time from profile measurements with varying levels of instrumentation. As the instrumentation increases, so likely will the intrusion and the overhead compensation techniques will be more stressed. However, it is important to understand that accurate performance profiling will also depend on the precision of measurement, in particular, the ability to observe small performance phenomena. Overhead compensation

can improve measurement accuracy, but it cannot remove measurement uncertainty for small events.

For any level of instrumentation, it is reasonable to expect that less instrumentation leads to more accurate profiling results than more instrumentation. Thus, if the total execution time is accurate, we might assume the rest of the profile statistics are also. However, there are two issues to keep in mind. First, performance variability due to environmental factors can arise even in un-instrumented applications. Second, the success of overhead compensation on profile statistics is difficult to assess given that the "real" profile values are not known. The best we can do then is to compare profiling results at one level of instrumentation with results from using less instrumentation, under the assumption that the execution is relatively stable and the results from less instrumented runs are more reliable and accurate.

4.1 Experimental Methodology

The experimental methodology we use to evaluate overhead compensation characterizes the profiling measurement for an application with respect to levels of instrumentation and sequential versus parallel execution. For the experiments we report here, we used three levels of instrumentation. The *main only (MO)* instrumentation is used to determine the total execution time for the "main" routine. This will serve as our standard estimate for overall performance using as little instrumentation as possible. The *profile all (PA)* instrumentation generates profile measurements for every source-level routine of the program. The *callpath all (CA)* instrumentation uses TAU's callpath profiling capabilities to generate profile measurements for routine callpaths of the program. Obviously, this *CA* instrumentation is significantly greater than *PA* and will further stress overhead compensation.

Five experiments are run for an application using the three levels of instrumentation. The *MO* experiment gives us a measure of total execution time. For parallel SPMD applications, we profile the "main" routine of the individual processes, using the maximum as the program's total execution time. The per process times can also be used for evaluation under the assumption the program's behavior is well-behaved. The *PA* experiment returns profiling measurements without compensation. We let *PA-comp* represent a *PA*-instrumented run with compensation enabled. Similarly, a *CA* experiment returns callpath profiling measurements without compensation and a *CA-comp* experiment returns callpath profile results after overhead compensation.

We can compare the "main" profile values from *PA*, *PA-comp*, *CA*, and *CA-comp* runs to the *MO* run to evaluate the benefit of overhead compensation. However, we can also look at other indirect evidence of compensation effectiveness. Assuming the *PA-comp* run delivers accurate profile results, we can compare the associated statistics from the *CA-comp* profile to see how closely they matched. This can also be done for the *PA* and *PA-comp* runs with different levels of instrumentation. Per process values can be used in all parallel cases for comparison under SPMD assumptions.

Ten trials are executed for each experiment. We have a choice of using profile results with the minimum "main" values or the average "main" values in the evaluation. Our preference is to use the profiles reporting minimums. The reason is that these runs are likely to have less artifacts in the execution (i.e., anomalies not directly attributed to the program) and, thus, represent "best case" performance. On the other hand, an argument can be made to take the average profile values, since artifacts may be related to the instrumentation. We report both values in our results below. However, it is important to note that calculating average profiles may not be reliable for programs that do not behave in a deterministic manner.

Following the experimental methodology above, we tested overhead compensation on all NAS parallel benchmark applications [1]. As the application codes vary in their structure and number of events, we expected differences in the effectiveness of compensation. We ran the ten experiments for each application sequentially and on 16 processors. Problems sizes were chosen mainly to achieve runtimes of reasonable durations. The parallel system used in our study was a Dell Linux cluster[2]. In the following sections, we report on six of the NAS benchmarks: SP, BT, LU, CG, IS, and FT.

4.2 Sequential Experiments

Table 1 shows the total sequential execution time of "main" in microseconds from the different profiles for the different applications. The minimum and mean values are reported. We also calculate the percentage error (using minimum and mean values) in approximating the *MO* time for "main." The dataset size (A or W) used in the experiments is indicated.

An important observation is that the TAU measurement overhead per event is already very small, on the order of 500 nanoseconds for flat profiling on a 2.8 GHz Pentium Xeon processor. This can be easily seen in the TAU profile results (not shown) where the overhead estimation is given as an event in the profile. Of course, the slowdown seen in the *PA* and *CA* runs depends on the benchmark and the number of events instrumented and generated during execution. Because more events are created for callpath profiling, we expect to see more slowdown for the *CA* runs.

The results show that overhead compensation is better at approximating the total execution time, both for flat profiles and for callpath profiles. This is generally true for all of the NAS benchmarks we tested. In the case of IS-A, the flat profile compensation (*PA-comp*) shows remarkable improvement, from a 193% error in the *PA* measurement to within 2.1% of the "main" execution time. The improvements in compensated callpath profiles for SP-W to less than 1% error are also impressive.

To be clear, we are instrumenting *every* routine in the program as well as every depth of callpath. If, as a result, we instrument a small routine that gets

[2] Hardware: 16 dual-processor 2.8 GHz Intel® Pentium 4 Xeon™ CPUs, 512 KB cache, 4 GB memory per node, gigabit ethernet interconnect, hyperthreading enabled. OS: Red Hat Linux 2.4.20-20.8smp kernel.

Table 1. Overhead Compensation Results for NAS Benchmarks on Linux Cluster - Sequential

Experiment	*MO*	*PA*	*PA-comp*	*CA*	*CA-comp*
	µsecs	*µsecs*	*µsecs*	*µsecs*	*µsecs*
SP-A **min**	387588657	397602281	392833924	405226516	399405895
mean	388540699	398360423	394245841	407233889	401650317
%error (min:mean)		2.5 : 2.5	1.3 : 1.4	4.5 : 4.8	3.0 : 3.3
SP-W **min**	65427051	67942093	66404006	71812623	65517453
mean	66178471	69254426	67104562	73659688	66687843
%error (min:mean)		3.8 : 4.6	1.4 : 1.3	9.7 : 11.3	0.1 : 0.7
BT-A **min**	522765488	549063282	542479898	553178345	532736660
mean	524248915	552617635	545409236	555959945	536680190
%error (min:mean)		4.6 : 5.2	3.4 : 3.8	5.8 : 6.0	1.9 : 2.3
LU-W **min**	297366632	300993317	302786082	306287598	303405699
mean	299395075	302941264	305796049	307849925	306172285
%error (min:mean)		1.4 : 3.3	0.0 : -0.6	10.2 : 8.9	3.4 : 2.6
CG-A **min**	5368659	5733951	5740469	6824800	6536302
mean	5560969	5758157	5764569	6916842	6628535
%error (min:mean)		6.8 : 3.5	6.9 : 3.6	27.1 : 24.3	21.7 : 19.1
IS-A **min**	5967910	17540614	6094620	35457776	2632054
mean	5987002	17667114	6215288	36008102	4441510
%error (min:mean)		193.9 : 195.0	2.1 : 3.8	494.1 : 501.4	-55.8 : -25.8
FT-A **min**	24593893	25418103	25296244	29104159	28754736
mean	25215853	25549141	25557557	29470907	28918045
%error (min:mean)		3.3 : 1.3	2.8 : 1.3	18.3 : 16.9	16.9 : 14.6

called many times, overheads can accumulate significantly. For callpath profiling with instrumentation including a small event, overheads will be effectively multiplied by the number of callpaths containing the small routine. This is what is happening in IS-A. Flat profile compensation can deal with the error, but callpath compensation cannot. It is interesting that the reason can be attributed to the small differences in overhead unit estimation, ranging in this case from 957 nanoseconds (minimum) to 1045 (maximum). This seemingly minor 90 nanoseconds difference is enough in IS-A callpath profiling to cause major compensation errors. Certainly, the proper course of action is to remove the small routine from instrumentation.

4.3 Parallel Experiments

Table 2 reports the results for parallel execution of the six NAS benchmarks on the Linux Xeon cluster. All of the applications execute as SPMD programs using MPI message passing for parallelization across 16 processors. For evaluation purposes, we compare minimum "main" values for each process to those for the *MO* run. Each process will complete its execution separately, resulting in different "main" execution times. We show the range of minimum values (labeled "high"

Table 2. Overhead Compensation Results for NAS Benchmarks on Linux Cluster - Parallel

Experiment	*MO*	*PA*	*PA-comp*	*CA*	*CA-comp*
	µsecs	*µsecs*	*µsecs*	*µsecs*	*µsecs*
SP-A **min (high)**	67369049	67519758	67758618	72968801	73416350
min (low)	64346890	64834412	64963104	67047549	67124742
%error (mean:high)		0.6 : 0.2	0.8 : 0.5	4.3 : 8.3	4.3 : 8.9
SP-W **min (high)**	13874506	14217942	14257427	15336991	13985473
min (low)	11306714	11602819	11628739	12539279	11064565
%error (mean:high)		2.5 : 2.4	2.5 : 2.7	9.9 : 10.5	-1.5 : 0.7
BT-A **min (high)**	76799427	77454300	77839767	85876074	85835820
min (low)	74182308	74696115	74937243	78018235	77721303
%error (mean:high)		0.6 : 0.8	1.0 : 1.3	5.5 : 11.8	5.4 : 11.7
LU-A **min (high)**	36966517	37783314	37629343	52540729	52395303
min (low)	34399415	35194131	35099696	43787261	43176436
%error (mean:high)		2.2 : 2.2	1.8 : 1.7	27.5 : 42.1	25.7 : 41.7
CG-A **min (high)**	4353851	4612676	4525479	8677331	8291439
min (low)	1848843	2076113	1950485	4252990	3691704
%error (mean:high)		7.4 : 5.9	3.8 : 3.9	84.1 : 99.3	65.6 : 90.4
IS-A **min (high)**	5420444	5973752	5836727	8301860	5585069
min (low)	2772617	3490618	3080709	5789329	1634756
%error (mean:high)		17.9 : 10.2	11.1 : 7.6	76.9 : 53.1	1.4 : 3.0
FT-A **min (high)**	8085574	8195461	8088853	9620210	9366497
min (low)	5422766	5518819	5485972	6021030	6029058
%error (mean:high)		0.8 : 1.3	-0.1 : 0.0	8.9 : 18.9	8.6 : 15.8

and "low" in the table), the mean error over this range (comparing process-by-process with the minimum results from the *MO* run), and the error of the mean and "high" values (effectively execution time of Node 0's "main").

Overall, the results show that compensation techniques improve performance estimates, except in a few cases where the differences are negligible. This is encouraging. However, we also notice that the results display a variety of interesting characteristics, including differences from the sequential results.

For instance, the *PA* values for SP-A, BT-A, and FT-A are practically equivalent to "main" only results, yet the sequential profiles show slowdowns. The *PA-comp* values are within less than 1% in these cases. We believe this suggests that the instrumentation intrusion is being effectively reduced due to parallelization, resulting in fewer events being measured on each process. We tend to characterize the SP-W flat profile experiments in the same way, since the errors are reasonably small and the minimum ranges are tight.

Other benchmarks show differences in their range of minimum "main" execution times. IS-A is one of these. It also has the greatest error for flat profile compensation. Compared to the sequential case, there is a significant reduction in *PA* error (193.9% to 10.2%) due to intrusion reduction, but the compensated values are off by 11.1% on average per process and 7.6% for Node 0's "main"

time (compared to 2.1% minimum error in the sequential case). This suggests a possible correlation of greater range in benchmark execution time with poorer compensation, although it does not explain why.

CG-A also has a significant difference in its "high" and "low" range, but its *PA-comp* errors are lower than IS-A. However, as more events are profiled with callpath instrumentation, the *CA* and *CA-comp* errors increase significantly. Compared to the sequential *CA* and *CA-comp* runs, we also see a slowdown in execution time compared to the sequential case. This is odd. Why, if we assume the measurement intrusion is being reduced by parallelization, do we see an execution slowdown? Certainly, the number of events is affecting compensation performance, as was the case in the sequential execution, but the increase in execution times beyond the sequential results suggests some kind of intrusion interdependency. In addition, we see the execution time range is widening.

Looking for other examples of widening execution time range with increased number of events, we find additional evidence in the callgraph runs (*CA* and *CA-comp*) for SP-A, BT-A, LU-A, and FT-A. The effect for LU-A is particularly pronounced. Together with the observations above, these findings imply a more insidious problem that may limit the effectiveness of our compensation algorithms. We discuss these problems below.

5 Discussion

The experiments we conducted were stress tests for the overhead compensation algorithms. We profiled all routines in the application source code for flat profiles and we profiled all routine calling paths for callpath profiles[3]. While the results show the overhead compensation strategies implemented in TAU are generally effective, we emphasize the need to have an integrated approach to performance measurement that takes into account the limits of measurement precision and judicious choice of events. It should not be expected that performance phenomena occurring at the same granularity as the measurement overhead can be observed accurately. Such small events should be eliminated from instrumentation consideration, improving measurement accuracy overall. In a similar vein, there is little reason to instrument events of minor importance to the desired performance analysis.

TAU implements a dynamic profiling approach where events to be profiled are created on-the-fly. This is in contrast with static profiling techniques where all events must be known beforehand [7]. Static approaches can be more efficient in the sense that the event identifiers and profile data structures can be allocated *a priori*, but these approaches do not work for usage scenarios where events occur dynamically. While TAU's approach is more general, modeling the overhead is more complicated. For instance, we do not currently track event creation overhead, which can occur at any time. Future TAU releases will include this estimate in the overhead calculation. The good news is that we made significant

[3] We did eliminate a few very small routines: `BINVCRHS`, `MATMUL_SUB`, and `MATVEC_SUB`, from the BT benchmark, and `ICNVRT` from CG.

improvements in the efficiency of TAU's profiling system in the course of this research. Our callpath profiling overheads were improved ten times by switching to a more efficient callpath calculation and profile data lookup mechanism.

Callpath profiling is more sensitive to recovery of accurate performance statistics for two reasons. First, there are more callpath events than in a flat profile and each callpath event is proportionally smaller in size. Second, we only estimate the flat profiling overhead at this time in TAU. The overhead for profiling measurements of callpaths is greater because the callpath must be determined on-the-fly with each event entry and the profiling data structures used must be mapped dynamically at runtime (flat profile data structures are directly linked). Nevertheless, it is encouraging how well compensation works with callpath profiling using less exact (smaller) overhead values. Also, TAU's implementation of callpath profiling allows the depth of callpath to be controlled at execution time. A callpath depth of 0 results in a flat profile. Setting the callpath depth to d results in events for all callpaths of length $\leq d$ being profiled. This callpath depth control can be used to limit intrusion.

The most important result from the research work is the insight gained on overhead compensation in parallel performance profiling. Our present techniques are necessary for compensating measurement intrusion in parallel computations, but they are not sufficient. Depending on the application's parallel execution behavior, it is possible, even likely, that intrusion effects seen on different processes are interdependent. Consider the following scenario. A master process sends work to worker processes and then waits for their results. The worker processes do the work and send their results back to the master. A performance profile measurement is made with overhead compensation analysis. The workers see some percentage intrusion with the last worker to report seeing a 30% slowdown. The compensated profile analysis works well and accurately approximates the workers "actual" performance. The master measurement generates very little overhead because there are few events. However, because the master must wait for the worker results, it will be delayed until the last worker reports. Thus, its execution time will include the last worker's 30% intrusion! Our compensated estimate of the master's execution time will be unable to eliminate this error because it is unaware of the worker's overhead. We believe a very similar situation is occurring in some, if not all, of the parallel experiments reported here.

Figure 1 depicts the above scenario. Figure 1(a) shows the measured execution with time overhead indicated by rectangles and termination times by triangles. The large triangle marks where the program ends. The overhead for the master is assumed to be negligible. The arrows depict message communication. Figure 1(b) shows the execution with all the overhead bunched up at the end as a way to locate when the messages returning results from the workers (dashed arrows) would have been sent and the workers would have finished (small shaded triangles), if measurements had not been made. Profile analysis would correctly remove the overhead in worker performance profiles under these circumstances. However, the master knows nothing of the worker overheads and, thus, our current compensation algorithms cannot compensate for it. The master

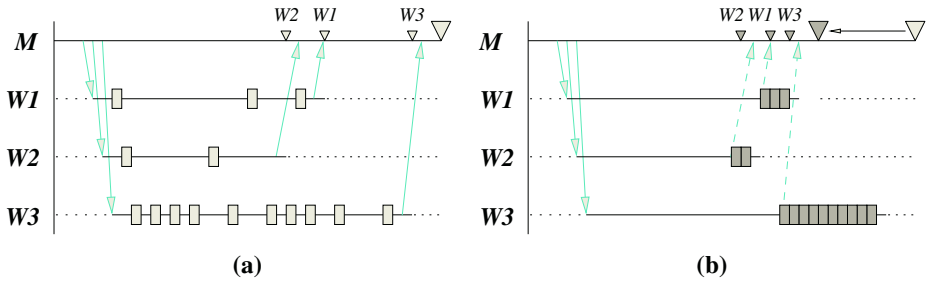

Fig. 1. Parallel Execution Measurement Scenario.

profile will still reflect the master finishing at the same time point, even though its "actual" termination point is much earlier.

Unfortunately, parallel overhead compensation is a more complex problem to solve. This is not entirely unexpected, given our past research on performance perturbation analysis [18–20]. However, in contrast with that work, we do not want to resort to a parallel trace analysis to solve it. The problem of overhead compensation in parallel profiling using only profile measurements (not tracing) has not been addressed before, save in a restricted form in Cray's MPP Apprentice system [29]. We are currently developing algorithms to do on-the-fly compensation analysis based on those used in trace-based perturbation analysis [25], but their utility will be constrained to deterministic parallel execution only, for the same reasons discussed in [17, 25]. Implementation of these algorithms also will require techniques similar to those used in PHOTON [27] and CCIFT [2, 3] to embed overhead information in MPI messages. While Photon extends the MPI header in the underlying MPICH implementation to transmit additional information, the MPI wrapper layer in the CCIFT application level checkpointing software allows this information to piggyback on each message.

6 Conclusion

Measured profiling has proven to be an important tool for performance analysis of scientific applications. We believe it has significant advantages over statistical profiling methods, but there are important issues of intrusion that must be addressed. In this paper, we focus on the removal of measurement overhead from measured profiling statistics. The algorithms we describe for quantifying the overhead and eliminating it on-the-fly have been implemented in the TAU performance system. Testing these algorithms using TAU on the NAS parallel benchmarks shows that they are effective at reducing the error in estimated performance. This is demonstrated for both flat and callpath profiling. In general, the overhead compensation techniques can be applied to any set of performance metrics that can be profiled using TAU.

However, there are still concerns and problems to address. We need to validate the compensation analysis approach on other platforms where different factors

will influence observed overhead. We need to better understand the limits of overhead compensation and its proper use in an integrated instrumentation and measurement strategy. In particular, the problems with overhead compensation analysis in parallel profiling require further study. While the current methods do reduce intrusion error in parallel profiling, they are unable to account for interdependent intrusion effects. We will address this problem in future research.

References

1. D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, M. Yarrow, "The NAS Parallel Benchmarks 2.0," Technical Report NAS-95-020, NASA Ames Research Center, 1995.
2. G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "Collective Operations in an Application-level Fault Tolerant MPI System," *International Conference on Supercomputing (ICS)*, 2003.
3. G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "Automated Application-level Checkpointing of MPI Programs," *Principles and Practice of Parallel Programming (PPoPP)*, 2003.
4. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A Portable Programming Interface for Performance Evaluation on Modern Processors," *International Journal of High Performance Computing Applications*, **14**(3):189–204, Fall 2000.
5. H. Brunst, M. Winkler, W. Nagel, H.-C. Hoppe, "Performance Optimization for Large Scale Computing: The Scalable VAMPIR Approach," In V. Alexandrov, J. Dongarra, B. Juliano, R. Renner, K. Tan, (eds.), *International Conference on Computational Science*, Part II, LNCS 2074, Springer, pp. 751–760, 2001.
6. L. De Rose, "The Hardware Performance Monitor Toolkit," *Euro-Par Conference*, 2001.
7. L. De Rose and F. Wolf "CATCH - A Call-Graph Based Automatic Tool for Capture of Hardware Performance Metrics for MPI and OpenMP Applications," *Euro-Par Conference*, LNCS 2400, Springer, pp. 167–176, 2002.
8. Alain Fagot and Jacques Chassin de Kergommeaux, "Systems Assessment of the Overhead of Tracing Parallel Programs," *Euromicro Workshop on Parallel and Distributed Processing*, pp. 179–186, 1996.
9. T. Fahringer and C. Seragiotto, "Experience with Aksum: A Semi-Automatic Multi-Experiment Performance Analysis Tool for Parallel and Distributed Applications," *Workshop on Performance Analysis and Distributed Computing*, 2002.
10. S. Graham, P. Kessler, and M. McKusick, "gprof: A Call Graph Execution Profiler," *SIGPLAN Symposium on Compiler Construction*, pp. 120–126, June 1982.
11. R. Hall, "Call Path Profiling," *International Conference on Software Engineering*, pp. 296–306, 1992.
12. J. Hollingsworth and B. Miller, "An Adaptive Cost System for Parallel Program Instrumentation," *Euro-Par Conference*, Volume I, pp. 88–97, August 1996.
13. IBM, "Profiling Parallel Programs with Xprofiler," IBM Parallel Environment for AIX: Operation and Use, Volume 2.
14. C. Janssen, "The Visual Profiler," http://aros.ca.sandia.gov/∼cljanss/perf/vprof/.
15. D. Kranzlmüller, R. Reussner, and C. Schaubschläger, "Monitor Overhead Measurement with SKaMPI," *EuroPVM/MPI Conference*, LNCS 1697, pp. 43–50, 1999.

16. D. Knuth, "An Empirical Study of FORTRAN Programs," *Software Practice and Experience*, **1**:105–133, 1971.
17. A. Malony, "Performance Observability," Ph.D. thesis, University of Illinois, Urbana-Champaign, 1991.
18. A. Malony, D. Reed, and H. Wijshoff, "Performance Measurement Intrusion and Perturbation Analysis," *IEEE Transactions on Parallel and Distributed Systems*, **3**(4):433–450, July 1992.
19. A. Malony and D. Reed, "Models for Performance Perturbation Analysis," *ACM/ONR Workshop on Parallel and Distributed Debugging*, pp. 1–12, May 1991.
20. A. Malony "Event Based Performance Perturbation: A Case Study," *Principles and Practices of Parallel Programming (PPoPP)*, pp. 201–212, April 1991.
21. A. Malony, S. Shende, "Performance Technology for Complex Parallel and Distributed Systems," In G. Kotsis, P. Kacsuk (eds.), *Distributed and Parallel Systems, From Instruction Parallelism to Cluster Computing, Third Workshop on Distributed and Parallel Systems (DAPSYS 2000)*, Kluwer, pp. 37–46, 2000.
22. J. Mellor-Crummey, R. Fowler, and G. Marin, "HPCView: A Tool for Top-down Analysis of Node Performance," *Journal of Supercomputing*, **23**:81–104, 2002.
23. P. Mucci, "Dynaprof," http://www.cs.utk.edu/∼mucci/dynaprof
24. D. Reed, L. DeRose, and Y. Zhang, "SvPablo: A Multi-Language Performance Analysis System," *International Conference on Performance Tools*, pp. 352–355, September 1998.
25. S. Sarukkai and A. Malony, "Perturbation Analysis of High-Level Instrumentation for SPMD Programs," *Principles and Practices of Parallel Programming (PPoPP)*, pp. 44–53, May 1993.
26. Unix Programmer's Manual, "prof command," Section 1, Bell Laboratories, Murray Hill, NJ, January 1979.
27. J. Vetter, "Dynamic Statistical Profiling of Communication Activity in Distributed Applications," *ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, ACM, 2002.
28. F. Wolf and B. Mohr, "Automatic Performance Analysis of SMP Cluster Applications," Technical Report IB 2001-05, Research Centre Juelich, 2001.
29. W. Williams, T. Hoel, and D. Pase, "The MPP Apprentice Performance Tool: Delivering the Performance of the Cray T3D," *Programming Environments for Massively Parallel Distributed Systems*, North-Holland, 1994.
30. M. Zagha, B. Larson, S. Turner, and M. Itzkowitz, "Performance Analysis Using the MIPS R10000 Performance Counters," *Supercomputing Conference*, November 1996.