Frequency-based Overhead Compensation in HPC Application Traces

Alef Farah*, Lucas Mello Schnorr*†, Jean-Marc Vincent†
*Institute of Informatics, Federal University of Rio Grande do Sul – UFRGS
†Univ. Grenoble-Alpes, France

***Abstract*—Application tracing is widely used for performance analysis of parallel applications. Tracing perturbs the measured system, mainly because of the execution of additional instructions. The perturbations may be very small by themselves, but they accumulate along the application execution, altering its behavior, which is undesirable for performance analysis. This phenomenon is known as the probe effect, and can be reduced by registering less information, although the trade-off is usually undesirable. Alternatively, one can compensate for it by reducing an estimate of the registering overhead from the event timestamps. In this work we characterize the overhead caused by a tracing tool for MPI applications and compensate for it using a novel approach in which the event registering frequency is taken into consideration. Early results comparing the total execution time with that of the uninstrumented application show an encouraging improvement over traditional compensation methods, which do not consider event frequency.**

I. Introduction

Tracing comprises the register of significant events from an application execution, such as subroutine calls and change in variable values. The recorded data is used for future analysis of the application behavior, in a post-mortem fashion [1]. Unlike profiling techniques [1], every event selected for recording is registered individually, uniquely identified by type, timestamp and additional data according to its type. Tracing is generally used for performance analysis of parallel and distributed applications, in which the chronological order of events is important for identifying multiple performance bottlenecks [1].

Event registering can be done either via software or hardware, with passive counters [2]. Software-based tracing is much more common because of its flexibility. The simplest method is source code instrumentation, in which logging routines are inserted into the code, either manually [3] or automatically [4]. Perturbation caused by these methods can be both direct, by adding overhead to the point at which they are added, and indirect, by allowing or inhibiting compiler optimizations. These perturbations accumulate over the application run, which might register millions of events, so even the quickest routines may add significant overhead to the final result. This can lead to traces that unfaithfully represent the application run harming the performance analysis.

One way to avoid indirect perturbations is to instrument compiled code [5]. However, this can still affect hardware optimizations such as out of order execution and caching. Direct perturbations, on the other hand, cannot be avoided without the usage of a non-intrusive measurement technique such as passive hardware counters. One can, alternatively, minimize them. Either by reducing the amount of information recorded or the set of events to be logged, either way trading low intrusion for less information, which is not desirable. Another approach is to compensate for direct perturbations by adjusting the event timestamps to consider the overhead added by the logging routines. This doesn't require less information to be registered, and is the focus of this work.

In order to compensate, the overhead must first be measured. The usual approach is to isolate the logging routine and take enough measurements of its execution time to obtain a statistically significant expected value [6], [7], [8]. Obtaining this value, which is obviously architecture dependent, is not as trivial as it might seem. Some tools [9], [4], [10] try to minimize the logging overhead by using very fast routines, leading to high volatility in their execution times. Due to their small size, any interference, such as task scheduling by the operating system, induce high relative error. Attention to this error is of utmost importance: an incorrect measurement for the overhead might lead to over or undercompensation, possibly shifting the compensated trace even further away from the uninstrumented (real) execution it is trying to approximate.

We observed that the logging routine execution time is a function of the frequency with which it is called, i.e. of the trace event registering frequency. Previous authors [6], [7], [5], [11], [12] did not take this into consideration, which might lead to a bad approximation of the uninstrumented run. We employ two metrics to compare our method, which considers event frequency, with traditional ones which do not. We use the distance from the uninstrumented application execution time and a space/time comparison. Early results comparing total execution times show improvement from traditional methods, though finer grain comparisons are still inconclusive.

The rest of this paper is organized as follows. Section II presents state of the art. Section III outlines the perturbation and compensation models we are proposing. Results and discussion are presented in Section IV. Conclusions and future work are shown in Section V.

II. Related Work

Wolf et. al. [8] define a model for compensating event timestamps in message passing applications. They estimate the logging routine execution time by doing repetitive calls to it, measuring the execution times and calculating the average. De Kergommeaux et. al. [12] re-introduce the same models with slightly different – but semantically equivalent –

formulas. They also provide a methodology to deal with clock synchronization. More relevant to our work, they discuss the impossibility of compensating non-deterministic applications (see Section III). Kranzlmuller et. al. [11] use SKaMPI, a tool for benchmarking MPI implementations, to measure monitoring overhead of tracing applications that use PMPI (a library interposing interface for tracing MPI programs). The usage of SKaMPI standardizes the measurement process. It is one of the first tools to consider the standard error when benchmarking, and is thus variability-aware.

All previously described related work exclude event frequency and, except for the SKaMPI effort, they do not mention measurement variability in the compensation process. As far as our knowledge goes there hasn't been recent development in the field of overhead compensation in application traces. Thus, our work also shines a new light on the topic considering the latest processor architectures. Unlike previous authors we acknowledge the fact that measuring overhead is (largely) dependent on event frequency, and define a measuring routine which takes event frequency into account.

III. Perturbation and Compensation Models

As an implementation of our model, described in the subsections below, we characterized and measured the overhead of Akypuera [9], a tool developed by the authors for tracing OpenMPI [13] applications. It uses source code instrumentation via the PMPI interface. We also designed and implemented a compensation tool to automatically adjust the timestamps of execution traces generated by Akypuera prior to performance analysis.

A. Measurement of the Intrusion Overhead

We isolate the logging routine to measure the tracing overhead considering how frequent such routine is called. From now on, we identify the logging routine simply as `log`, in order to simplify the explanation. We measure its runtime with a user-defined call frequency f, equal to the event call frequency in the application trace. To do so the user inputs for how long he wants to measure `log` under f. After a warmup to avoid spurious observations, we do $t \times f$ calls to `log` and a function to sleep $1/f$ units of time, which we'll call `sleep`. If the `log` execution was instantaneous and `sleep` runtime was exactly the amount of time it is asked to sleep (i.e., if it had zero overhead), then the calls would take exactly t units of time. Thus, to get the mean execution time of `log` we reduce t from the measured execution time, and also reduce the overhead of `sleep` under f, which is measured in the exact same fashion. We repeat this process according to a user definition, obtaining n means which later on are used to obtain an estimation of the expected value for the overhead of `log` under f. Algorithm 1 details this routine. Notice that `sleeping` holds the time spent sleeping and also the overhead of both `sleep` and of the timer.

Ideally the `sleep` function should mimic the system load during the execution of the traced application. Currently we use a function which suspends the execution of the current thread (namely POSIX `nanosleep`), which is an imperfection of our tool. However, our methodology allows system load to be taken into account by using a more appropriate function instead. We also observe that in our implementation we assume an uniform event frequency throughout the application run. We therefore assume the application behavior is regular. Finally, we currently assume the same event frequency on every process of a parallel application.

Algorithm 1 The benchmarking routine

```
measurements ← ∅ ; period ← 1/f ; iters ← t × f
for i = 0; i <iters; i++ do                          ▷ Warmup
    sleep(period)
end for
sleeping ← 0
for i = 0; i <replications; i++ do
    start_timer()
    for j = 0; j <iters; j++ do
        sleep(period)
    end for
    end_timer()
    sleeping += elapsed_time
end for
sleeping = sleeping / replications
for i = 0; i <replications; i++ do
    start_timer()
    for j = 0; j <iters; j++ do
        log()
        sleep(period)
    end for
    end_timer()
    measurement_i ← (elapsed_time − sleeping) / iters
end for
```

This might seem overzealous, but as aforementioned the calls to `log` are usually very fast. In our case, they are smaller than the overhead of `sleep`. We observed that, for very high frequencies, the variability is very large and we often end up with negative values for execution times after reducing the overhead of `sleep`. There are two strategies to handle high measurement variability: taking more replications or considering the standard error. For the former, we recommend the user do a large number of replications, i.e., such that he obtains a (positive) estimation of the expected value that ceases to change with an increase in the amount of replications. The replications have to be used both when measuring the overhead of `sleep` as well as when measuring the overhead of `log`. For the later, set the number of replications that gives a sufficiently low value for the standard error of the mean [11]. The second strategy can also be implemented in an online fashion, during the execution of Algorithm 1.

B. Compensation Strategies using the Measured Overhead

After estimating the overhead as described in Section III-A, we subtract such estimation from the timestamps of recorded events. For events that are local to a process, i.e. independent

of events from other processes, this compensation is done directly. For non-local events, i.e. events that depend on events from other processes, we must compensate based also on the timestamp of any dependent event in order to respect and maintain the causality between them.

Given $event_m^i$, the ith measured timestamp of an event on a certain process, $event_a^i$, the approximated (compensated) timestamp of that event is given by Equation 1, where O is the overhead estimate.

$$event_a^i = event_a^{i-1} + (event_m^i - event_m^{i-1}) - O \quad (1)$$

Compensating non-local events is not as straightforward, and depends on the concurrency model. We adopt the formulas for message passing applications defined by Malony [8]. As noted by Vincent [12], compensation in the face of non-determinism may change the program behavior because it might break the observed causal relationships among processes. In the case of MPI applications, non-determinism is present when using `MPI_ANY_SOURCE` to receive a message from any process instead of a specific one. In this case, our tracing tool register which process actually sent the message in *that* execution, instead of registering `MPI_ANY_SOURCE`. Thus, the application might be non-deterministic, but the traced execution is not, and we can compensate it normally.

IV. Early Results and Discussion

In this section we present early results comparing the implementation of our method, which considers event frequency to calculate the mean execution time, with traditional methods which are base solely on averages. We traced a set of applications (described in section IV-B) in a shared memory environment (in IV-A) and compared the compensation techniques using several metrics (in IV-B).

A. Hardware/Software Configuration and Benchmarks

Besides using simple "ping pong" applications, we evaluate our model with two real world applications: Ondes3D v1.1, an earthquake simulator [14] and OSU Microbenchmarks v5.2 [15], more specifically the `osu_multi_lat` benchmark.

We use a server from INF/UFRGS with 32GB of memory and two Intel Xeon E5-2630 Sandy Bridge processors with six physical cores each (each core with two processing units), running Ubuntu 14.04.1, Linux 3.16.0-51. Everything was compiled using OpenMPI 1.6.5 implementation of the MPI standard and GCC 4.8, with the default optimization flags from each application. More specifically, all tests were done using the SM (shared memory) BTL (byte transfer layer) of OpenMPI, with the default MCA (Modular Component Architecture) parameters.

B. Results

We first do a visual space/time comparison of the compensated and uncompensated traces. Such fine grained comparison strategy looks for small changes in the compensated version. An example is shown in Figure 1, depicting a space/time view of the original execution trace of a run of Ondes3D. The processes are displayed on the vertical axis, while the runtime is in the horizontal axis. Each rectangle represents an event, and the arrows show message passing among them. The arrow color represents the message size in bytes. Early attempts with this method yielded too dissimilar views for a given time slice, rendering the comparison between the compensated (with our method) and original traces inconclusive. In other words, compensation shifted the event timestamps to such a degree that zooming in the same time slice renders two different regions of the trace, one in the original trace and another in the compensated one, at least at the end of the trace file, where the accumulated overhead is larger. Since we could not determine the fine grain impact of our method, we did not go any further with this metric to try to compare ours with traditional methods. As future work we intend to run Ondes3D on a networked environment and use the simpler metric described below to evaluate our method.

The main problem with the space/time view is the lack of a unique and simple metric that represent how far we are from the original uninstrumented version. Because of this, we consider also a simple metric to compare mean application execution times – that of the uninstrumented application run, that of the instrumented application, and that of the compensated version of that trace using both ours and traditional methods. In this case we say that a method is better than another if the execution time is closer to that of the uninstrumented run. For such comparison, we do thirty executions and compare the mean execution times.

Table I shows the comparison of the mean execution time of the OSU Microbenchmarks application. When considering event frequency, the execution times in the compensated trace are better than when not considering it. Although we observed similar results with other applications as well, we note that the difference in execution time is within the standard error. Furthermore, there was very little difference between the instrumented and uninstrumented execution times (i.e. little intrusion to be compensated in the first place). The measurements fit a somewhat normal distribution, with two modes towards the center.

TABLE I: Comparing the mean execution time of each trace (for a 99.7% confidence interval) of the OSU benchmark.

Execution	Mean	Standard error
Uninstrumented	12.958	0.2805
Instrumented	13.102	0.1766
Traditional	13.058	0.1765
Frequency (our approach)	12.945	0.1765

V. Conclusion and Future Work

In performance analysis we wish for the recorded application behavior to faithfully approximate the real application run. Compensation is one method to do this approximation, and a correct overhead estimation is paramount. In this work

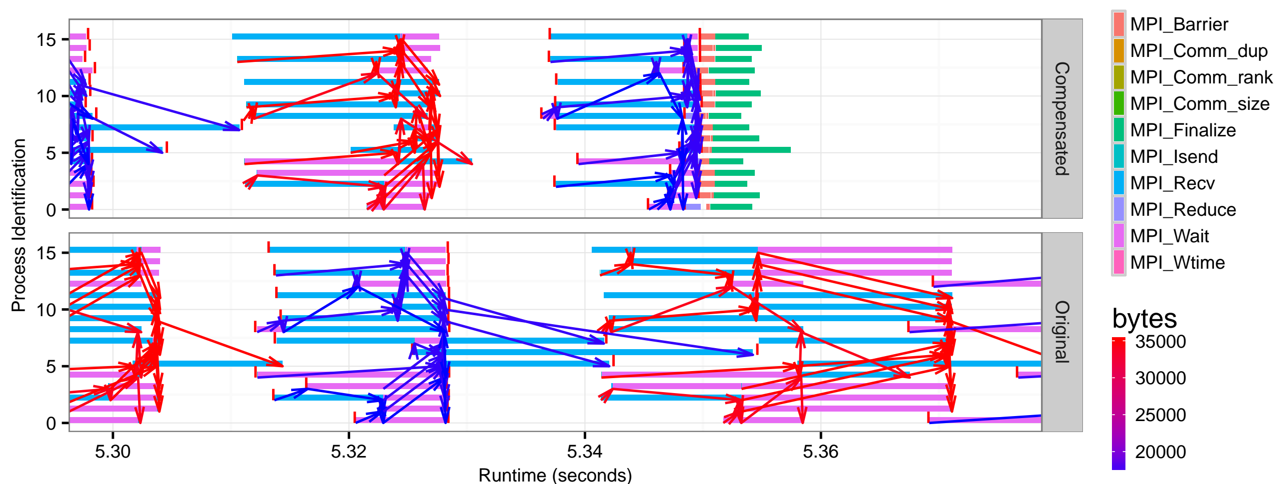

Fig. 1: Space/time comparison of original (bottom) and compensated version (top) for the Ondes3D trace with 16 ranks.

we observed that event frequency is a determinant factor of logging overhead. We also presented a novel approach to deal with this relation, and presented evidence from early experiments suggesting improvement over traditional methods.

Although we did not yet find a suitable way to do a fine grained comparison of our method with previous ones, mean execution time comparisons (as used by previous authors [8]) indicate improvements. Moreover, the mere observation of the dependency between event frequency and execution time should be enough to reduce the error in the compensation.

Future work include tests with applications with larger intrusion, and tests in a networked environment instead of a shared memory one. We also intend to improve our implementation of the method described in this article, for instance considering per process frequencies and comparing this to the current implementation. We also intend to study the validity of this approach when working with irregular applications. At the same time we shall keep searching for comparison metrics to better evaluate the impacts of our work.

Acknowledgements

The results reported in this study were generated in virtue of the agreement between Hewlett Packard Enterprise (HPE) and the Federal University of Rio Grande do Sul (UFRGS), financed by resources in return for the exemption or reduction of the IPI tax, granted by Brazilian Law nº 8248, 1991, and its subsequent updates. This investigation also receives funds from the H2020 program EU and MCTI / RNP-Brazil through HPC4E project with code 689772, the FAPERGS / Inria ExaSE design, universal design CNPq 447311 / 2014-0, and international CNRS / LICIA laboratory.

References

[1] B. de Oliveira Stein, "Depuração e visualização de programas paralelos," in *I Escola Regional de Alto Desempenho*, T. A. Divério and P. O. Navaux, Eds. Porto Alegre: Gráfica da PUCRS, 2001, pp. 151–175.

[2] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *2010 39th International Conference on Parallel Processing Workshops*. IEEE, 2010, pp. 207–216.

[3] G. J. da Silva, L. M. Schnorr, and B. Stein, "Jrastro: A trace agent for debugging multithreaded and distributed java programs," in *Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing*. Los Alamitos: IEEE Computer Society, 2003, pp. 46–54.

[4] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, *Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir*. Springer Berlin Heidelberg, 2012, pp. 79–91.

[5] S. S. Shende and A. D. Malony, "The tau parallel performance system," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[6] A. Fagot and J. C. de Kergommeaux, "Systematic assessment of the overhead of tracing parallel programs," in *Parallel and Distributed Processing, 1996. PDP'96. Proceedings of the Fourth Euromicro Workshop on*. IEEE, 1996, pp. 179–186.

[7] A. D. Malony, D. A. Reed, and H. A. G. Wijshoff, "Performance measurement intrusion and perturbation analysis," *IEEE Transactions on parallel and distributed systems*, vol. 3, no. 4, pp. 433–450, 1992.

[8] F. Wolf, A. D. Malony, S. Shende, and A. Morris, "Trace-based parallel performance overhead compensation," in *International Conference on High Performance Computing and Communications*. Springer, 2005, pp. 617–628.

[9] L. M. Schnorr, "Akypuera," http://github.com/schnorr/akypuera, 2016.

[10] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," in *Proceedings of WoTUG-18: Transputer and occam Developments*, vol. 44. mar, 1995, pp. 17–31.

[11] D. Kranzlmüller, R. Reussner, and C. Schaubschläger, "Monitor overhead measurement with skampi," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 1999, pp. 43–50.

[12] J. C. De Kergommeaux, E. Maillet, and J. Vincent, "Monitoring parallel programs for performance tuning in cluster environments," *Parallel Program Development for Cluster Computing: Methodology, Tools and Integrated Environments" book, P. Kacsuk and JC Cunha eds*, 2001.

[13] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[14] F. Dupros, C. P. Ribeiro, A. Carissimi, and J.-F. Méhaut, "Parallel simulations of seismic wave propagation on numa architectures." in *PARCO*, 2009, pp. 67–74.

[15] T. O. S. University, "Osu micro-benchmarks," http://mvapich.cse.ohio-state.edu/benchmarks, 2016.