Trace-Based Parallel Performance Overhead Compensation

Felix Wolf[1], Allen D. Malony[2], Sameer Shende[2], and Alan Morris[2]

[1] Innovative Computing Laboratory,
University of Tennessee
fwolf@cs.utk.edu

[2] Department of Computer and Information Science,
University of Oregon
{malony,morris,sameer}@cs.uoregon.edu

Abstract. Tracing parallel programs to observe their performance introduces intrusion as the result of trace measurement overhead. If post-mortem trace analysis does not compensate for the overhead, the intrusion will lead to errors in the performance results. We show that measurement overhead can be accounted for during trace analysis and intrusion modeled and removed. Algorithms developed in our earlier work [5] are reimplemented in a more robust and modern tool, KOJAK [12], allowing them to be applied in large-scale parallel programs. The ability to reduce trace measurement error is demonstrated for a Monte-Carlo simulation based on a master/worker scheme. As an additional result, we visualize how local perturbation propagates across process boundaries and alters the behavioral characteristics of non-local processes.

Keywords: Performance measurement, analysis, parallel computing, tracing, message passing, overhead compensation.

1 Introduction

Trace-based measurement is used to observe the performance of a parallel program when one wants to see the interoperation of multiple threads or processes of execution, as it is recorded in a time-sequence trace of events. Any performance measurement, tracing included, will introduce *overhead* during program execution due to extra code being executed and hardware resources (processor, memory, network) consumed. When performance overhead affects the program execution, we speak of *performance (measurement) intrusion*. Performance intrusion, no matter how small, can result in *performance perturbation* [6] where the program's measured performance behavior is "different" from its unmeasured performance. Whereas performance perturbation is difficult to assess, performance intrusion can be quantified by several metrics, the most important of which is dilation in program execution time. This type of intrusion is often reported as a percentage slowdown of total execution time, but the intrusion effects themselves will be distributed throughout the performance results. In the case of tracing, we will also see performance error due to intrusion (i.e., performance perturbation) in the timings of the interdependent events between the processes.

Of course, we cannot compare the measured parallel execution with the "real" parallel execution to determine the intrusion error because we do not have any information about what the execution would be like without instrumentation. All we know is that there is measurement overhead included in the trace data and that this overhead may have intruded on the parallel performance in such a way as to cause misleading performance effects. For tracing, the overhead introduced with each event measurement includes the creation of an event record and writing it to a trace buffer. If we can determine the overhead size, it may be possible to subtract this overhead for each event individually and generate a second "overhead-free" trace file. We must be careful in doing so not to violate the *happened-before* relation [1] that exists between interdependent process events.

While performance intrusion can alter program execution and, thus, perceived performance, parallel performance tools rarely attempt to adjust the performance measurements to compensate for the intrusion. Recently, we have shown overhead compensation is possible to do in parallel profiling [7, 8]. However, profiling summarizes performance data and, thus, loses the performance detail captured in traces. Also, because overhead compensation must be done online, certain forms of performance perturbation cannot be resolved during profiling. With trace-based overhead compensation, we have the opportunity of preserving performance detail while dealing with more complex intrusion effects. In our earlier work, we designed the performance models necessary for trace perturbation analysis [5, 10]. However, our implementation of these were for research purposes only. In this paper, we update our algorithms and build them into a robust trace measurement and analysis system.

Section 2 provides a brief background on performance intrusion and perturbation analysis. Our algorithms for overhead compensation are presented in Section 3. Section 4 outlines the implementation of these algorithms in the trace analysis tool. We demonstrate the techniques on a set of validation experiments. Section 5 describes the experimentation environment, the testcases, and the trace analysis results. Conclusions and future work are given in Section 6.

2 Tracing, Intrusion, and Perturbation

Events are actions that occur during program execution. Typical events include *interval events* that are characterized by a pair of actions marking *entry* and *exit* points, and *atomic events* that occur at a single place or time. Tools insert measurement code to track the performance of a parallel program as made visible by the instrumented events. Tracing collects event records in an *event trace*. Each record describes an event, when it occurred, and any associated data. From this information, we can see the patterns of execution behavior that contribute to the performance. The measurement intrusion in event traces displays itself as alterations of event timing and order. The goal of overhead compensation in trace analysis is to remove the time intrusion due to measurement overhead and fix its effects on event ordering, in hopes of recovering the actual performance behavior.

For discussion purposes, let us consider a message passing program. If we look at the impact of overhead on events local to a process, there is a time dilation (slowing

down) of when events occur, resulting in later event time stamps compared to an actual event trace. Because the events occur locally, this dilation can be directly determined and corrected. However, every message communication links the process event streams and the evolution of process times become dependent at these points. The result of measurement overhead affects the interdependent ordering and timing of these events compared to actual. The parallel execution semantics, as reflected in the message communication operations and how the message data is used, determines process dependencies and message event ordering relationships, but only partially. Non-deterministic execution allows for alternative message event orderings, different from observed. It is important to understand that the only information we have about process interdependencies are the message communication events and when they occur in the *measured* execution. If our trace analysis does not have enough information to determine if a different (reconstructed) event order is valid, it must enforce the *same* ordering of message communication events in the "approximated" execution as in the measured execution.

In contrast with our techniques for parallel profile overhead compensation, trace overhead analysis can be thought of as a trace-driven replay of the execution where we can apply both event and timing models to correct intrusion effects. There is also opportunity in trace analysis to utilize measurements of interprocess events to improve the accuracy of the approximated execution, such as with computed message communication times. For the work reported here, it is important to remember that the goal of tracing is to observe detailed temporal performance. Thus, we hope that the overhead analysis will result in more accurate performance characterization both as it has to do with overall performance (e.g., more accurate total execution times) as well as local performance details (e.g., waiting times for individual message communications).

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 [4] quantifies the overhead of MPI monitors using the benchmarking suite SKaMPI, and Fagot's work [2] assesses systematically the overhead of parallel program tracing. The work by Hollingsworth and Miller [3] demonstrates the use of measurement cost models, both predicted and observed, to control intrusion at runtime via measurement throttling and instrumentation disabling. Their work primarily deals with profiling-based measurement.

3 Overhead Compensation Algorithms

The trace of a parallel program's execution provides time-sequenced information about the events that occurred and when they occurred. To compensate for the overhead during measurement, we want to characterize the amount of overhead (O) for each event measurement, and subtract that overhead from the event timings. For events that are *local* to a process (we will call these *independent events*), this overhead compensation can be done directly. For events that are involved in dependent execution, we must take care not to violate happened-before time order relationships [1].

The algorithms we present below are based on our earlier work [5, 10], as targeted here to MPI message passing parallel programs. We make several assumptions in these algorithms:

– Only `MPI_Send()` and `MPI_Recv()` are used for point-to-point communication.
– `MPI_Send()` is always non-blocking.
– Only *n-to-n* and *1-to-n* collective operations are considered.
– The per-event measurement overhead (O) is constant.
– The buffer copy time is a function of message size $C(size(msg))$.

Clearly, the values C and O are platform specific and must be measured. For C, we run experiments with different message sizes and build a table of per byte copy times to be queried during analysis. For O, we run an experiment where N trace events are generated immediately following each other. The amount of time consumed is then divided by N to give O.

Based on the assumptions above, all dependent execution is due to message communication. Point-to-point (P2P) communication involves only the sender and receiver. The dependencies in collective communication are more interesting. While it is possible to logically reduce collective communication to P2P communication, doing so may restrict the analysis from applying what is known about the collective execution semantics.

3.1 Independent Events

Independent events are events that do not directly dependent on communication. In other words, in our environment, they are not communication events. Consider the ith event on a process, $event^i$. We use the notation $event^i_m$ to denote the *measured time stamp* of the event and $event^i_a$ to denote the *approximated time stamp* of the event after trace analysis. The trace analysis moves forward in the trace for each process, computing the approximated time stamp of the next event. Thus, at any point in time during the analysis, we look to see which immediate next event on each process can be processed next.

Let us assume $event^{i-1}_a$ has been computed and the next event, $event^i$ is not a communication event. The trace analysis can determine $event^i_a$ by:

$$event^i_a \;=\; event^{i-1}_a \;+\; (event^i_m - event^{i-1}_m) \;-\; O \qquad (1)$$

Effectively, we keep the execution time between the two events, $event^i_m - event^{i-1}_m$, and subtract the overhead. To this value we then add the approximated time stamp of the predecessor event, $event^{i-1}_a$.

3.2 Dependent Events

What happens if the event is a communicaton event? Here is where things get interesting. Let us focus on P2P communication first. There are six events to consider: *enter.send*, *send*, *exit.send*, *enter.recv*, *recv*, and *exit.recv*. Among these, only *recv* directly depends on communication. Often instrumentation of message communication is done using an interposition library, such as in the MPI profiling interface, PMPI [9]. Here, the time stamps of the *send* and *enter.send* events will likely differ only by a very small amount. The same is true for *recv* and *exit.recv*.

There are two cases in the measured execution to consider for a particular P2P communication:

(m.1) $enter.recv_m \leq exit.send_m$
(m.2) $enter.recv_m > exit.send_m$

It should be understood that the send and receive events are taking place on two different processes. Condition (m.1) means that there is a temporal overlap between the `MPI_Send()` and the `MPI_Recv()` operation. Condition (m.2), in contrast, means that there is a gap between the two.

(m.1) Communication time can be measured. If $enter.recv_m$ occurs before $exit.send_m$, we assume that the receiver can begin processing the message as soon as it is delivered. As a result, we can calculate the actual communication time from the measured trace:

$$Comm_m = recv_m - send_m \tag{2}$$

This is important since the measured communication time is the most accurate representation of communication performance. Two cases result for the approximated execution:

(a.1) $send_a + Comm_m > enter.recv_a$

(a.2) $send_a + Comm_m \leq enter.recv_a$

In the first approximation case, the entry to the receive occurs before the communication completes, meaning that the receiver has to wait. Thus, the approximated receive time can be determined by:

$$recv_a = send_a + Comm_m \tag{3}$$

In the second case, the receive occurs after the message has already been delivered and supposedly is present at the receiving process. All that is left to do is for the receiver to copy the message into the receive buffer:

$$recv_a = enter.recv_a + C(size(msg)) * size(msg) \tag{4}$$

Again, in our experiments, we use a lookup table to determine C for difference message sizes.

(m.2) Communication time cannot be measured. If the receive operation begins after the send operation has finished, we cannot use the trace measurement directly to compute the message communication time. An upper bound approximation on the communication time still comes from the communication measurement:

$$Comm_a^{upper} = recv_m - send_m \tag{5}$$

However, this time may include time a message spends sitting at the receiver before the receive begins. A lower bound approximation effectively assumes that the transmission

time through the communications network is zero. In this case, we need to only account for the message copy time both at the sender and at the receiver:

$$Comm_a^{lower} = 2 * C(size(msg)) * size(msg) \tag{6}$$

Since the start times of the send and the receive operation might be significantly pulled apart in the approximated execution, there is no guarantee that the lower-bound and the upper-bound communication times together give a valid time interval for the approximated receive event, that is, the following condition might be violated:

$$recv_a > enter.recv_a \tag{7}$$

This consistency requirement leads to the stipulation of a minimum communication time that has to be observed in both cases.

$$Comm^{min} = (enter.recv_a - send_a) + (C(size(msg)) * size(msg)) \tag{8}$$

Now, both the lower-bound and the upper bound communication time need to be modified not to fall below the minimum, similar to what we did earlier in case (m.1 / a.2).

$$Comm_a^{lower} = max(2 * C(size(msg)) * size(msg), Comm^{min}) \tag{9}$$

$$Comm_a^{upper} = max(recv_m - send_m, Comm^{min}) \tag{10}$$

Finally, the bounds for the approximated receive time are:

$$send_a + Comm_a^{lower} \leq recv_a \leq send_a + Comm_a^{upper} \tag{11}$$

3.3 Collective Communication

For our work in this paper, we also consider *n-to-n* and *1-to-n* collective communication operations. For any collective communication, we can transform the operation to point-to-point communication and then apply the formulas above to perform overhead compensation. However, it is not so easy to translate collective communication operations into their P2P equivalents. Also, collective communication has additional semantics that must still be enforced when processing the collective events. These semantics can be used to build overhead compensation algorithms specific to collective operations.

n-to-n. Consider *n-to-n* collective communication. There are two collective events of interest for each process: *enter* and *exit*. The approximated *enter* time stamp, $enter_a^i$, is determined for each process i based on the algorithm for a independent event. However, $exit_a^i$ is dependent on when the collective synchronization occurs. Let j be the process with the latest measured entry event, $enter_m^j$. Let k be the process with the latest approximated entry event, $enter_a^k$. Let l be the process with the earliest measured exit event, $exit_m^l$. Because *n-to-n* collective operations enforce collective synchronization, we could assume $exit_a$ is computed to be the same for all processes:

$$exit_a^i \;=\; enter_a^k \;+\; (exit_m^l - enter_m^j) \tag{12}$$

Here, $exit_m^l - enter_m^j$ is a measurement (from the trace) of the time to synchronize, once all processes have entered the collective communication. However, we can also measure this synchronization time for each process individually:

$$exit_a^i \;=\; enter_a^k \;+\; (exit_m^i - enter_m^j) \tag{13}$$

1-to-n. In the case of *1-to-n* collective communication, the translation to a P2P equivalent form will work fine for approximation purposes. The one sender (*root*) process has the events *enter.send* and *exit.send*. The multiple receivers each have events *enter.recv* and *exit.recv*. For each send-receiver pair, we translate the events as follows:

$send := enter.send$
$recv := exit.recv$

We compute $size(msg)$ as the amount of data received by receiver from root.

Note that the actual communication time cannot be measured because it is not known which fraction of the send operation was performed on behalf of a particular receiver. Therefore, we essentially give an upper-bound approximation. Plus, as collective communication is often implemented in a tree-like fashion, receivers may be senders as well. We do not take this into consideration.

4 Implementation

We validated our model using a prototype implementation within the KOJAK performance evaluation system [12]. KOJAK is a suite of performance tools that collect and analyze trace data from parallel programs including MPI applications. Event traces are generated automatically at runtime using a combination of source code annotations or compiler-supported instrumentation and hardware counters. The analysis component uses pattern recognition to convert the traces into information about performance bottlenecks relevant to developers.

Before executing the application, it is linked to a tracing library responsible for generating the trace file at runtime. The trace files are written in the EPILOG format. Overhead-intensive activities performed by the library to support trace-file generation, such as offset measurements among local clocks for a later time synchronization or file IO operations to flush the memory buffer upon overflow, are enclosed by special records so that the compensation filter can account for these overheads. After the application has terminated, an off-line analyzer scans the resulting trace files for execution patterns, classifies them by behavior type, and quantifies their impact on the overall performance. The results can be viewed in a GUI that shows performance problems broken down by call path and process or thread (see Figure 2).

At the core of the analyzer is a library called EARL [13] reading EPILOG traces and providing abstractions that simplify the task of detecting patterns in the event stream.

These abstractions include execution state information, such as the progress of collective operations at the time of a given event, and links between related events that allow, for example, the analyzer to find the send event to a given receive event. Another important feature is EARL's ability to access events randomly by their relative position in the trace file.

Because of its convenient model to access and process trace information, EARL was chosen to implement the compensation filter. The filter consists of three parts: (i) a component to measure platform-specific quantities, such as the average per-event overhead and the memory bandwidth needed to compute the buffer copy time, (ii) a queuing system to reorder the events in accordance with the approximated time stamps, and (iii) the actual compensation engine to execute the algorithms described earlier in Section 3. To accommodate cache effects, the memory bandwidth is measured for buffers of varying size. To improve scalability, the queuing systems maintains only a finite window of time stamps, events whose final position within the approximated trace can be determined and whose time stamps are no longer needed to approximate subsequent events are written to the approximated trace.

5 Experimental Results

To illustrate the effectiveness of our strategy, we examine a parallel MPI application that computes the value of π using a Monte-Carlo integration algorithm, which calculates the area under the circle function curve from 0 to 1. The program comprises of a master (or server) task that generates work packets with a set of random numbers. The master task waits for a request from any worker and sends a chunk of randomly generated numbers to it. For each pair of numbers that is given to a particular worker, it finds out whether the pair of Cartesian coordinates represented by the number is inside or outside the circle. Thus, the workers collectively estimate the value of π iteratively until it is within a given error range.

We executed the application in two modes: uninstrumented and instrumented. The instrumentation was applied to all user functions and MPI functions to generate an EPILOG trace file during execution. The number of processors was varied between 2 and 32 on a 4-way Intel Pentium III Xeon 550 MHz Linux cluster with 8 nodes. We ran the application 5 times under each configuration and took the shortest run as our representative. The time stamps in the trace files were initially collected using the RDTSC timer and later off-line synchronized with process 0 using linear interpolation between two offset measurements to compensate for different clock drifts among local clocks. For each measured trace file, two approximated trace files were generated, the first one using lower-bound approximation, the second one using upper-bound approximation. The results are shown in Figure 1.

The main fraction of the instrumentation overhead results from a small function named `get_coords()` that is frequently called by worker processes. Since number of invocations is proportional to the amount of work, the amount of intrusion declines as the number of worker increases. It can be seen that the execution times generated by both approximations come close to the uninstrumented time. When the dilation of execution time introduced by the instrumentation is high, our approximation proves

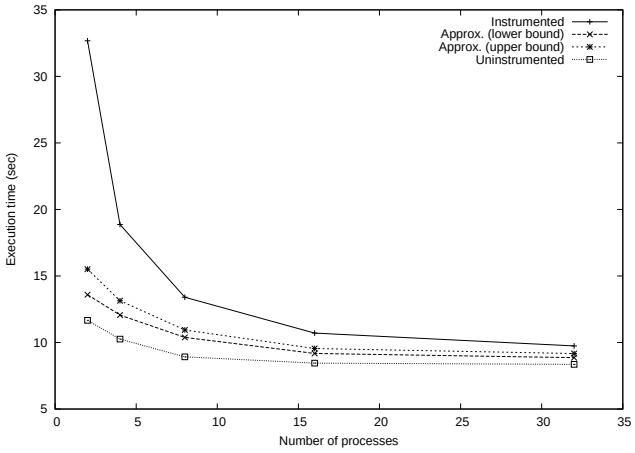


Fig. 1. Measured and approximated execution times for Monte-Carlo application.

to be most effective in terms of the percentage of overhead removed. Unfortunately, however, the lower-bound compensation is still too pessimistic in that it remains consistently above the uninstrumented execution. Reasons for this might be found in inaccurate measurements of platform constants and in simplified assumptions made by the model itself.

In Figure 1, we only investigate execution time *slow down*. The actual strength of our approach, however, is that the approximated event traces allow us to study the perturbation, that is, the qualitative change in program behavior caused by the overhead. Since interprocess communication can propagate instrumentation overhead across different processes, perturbation effects may be observed at a process that actually does not produce significant overhead itself. To examine qualitative perturbation effects in more detail, we applied KOJAK's pattern analyzer EXPERT to the measured and the approximated trace file. The two results have been subtracted using KOJAK'S performance algebra utility [11] to study the overhead composition in detail. The results are shown in Figure 2 displayed in the KOJAK GUI.

All numbers represent percentages of the difference in the execution time between measured and approximated execution. The left pane shows the overall difference broken down by behavior type for the entire program and all processes. It can be seen that the majority (63 %) of the overhead is non-communication (i.e., metric *Execution* expanded). The non-communication overhead has been found to be a direct effect of the tracing library's operation (i.e., per-event overhead and flushing the memory buffer) and is almost exclusively caused by workers. On the other hand, a significant fraction (28 %) of the total overhead is waiting time within the *Late Sender* pattern, which describes a receiver waiting for a message that has not been sent yet. In contrast to our previous finding, this indirect effect of perturbation can be nearly exclusively (80 %) attributed to the server.

Figure 3(a) shows the measured execution of one iteration in our Monte-Carlo example with four processes in a time-line diagram. The finely striped sections represent

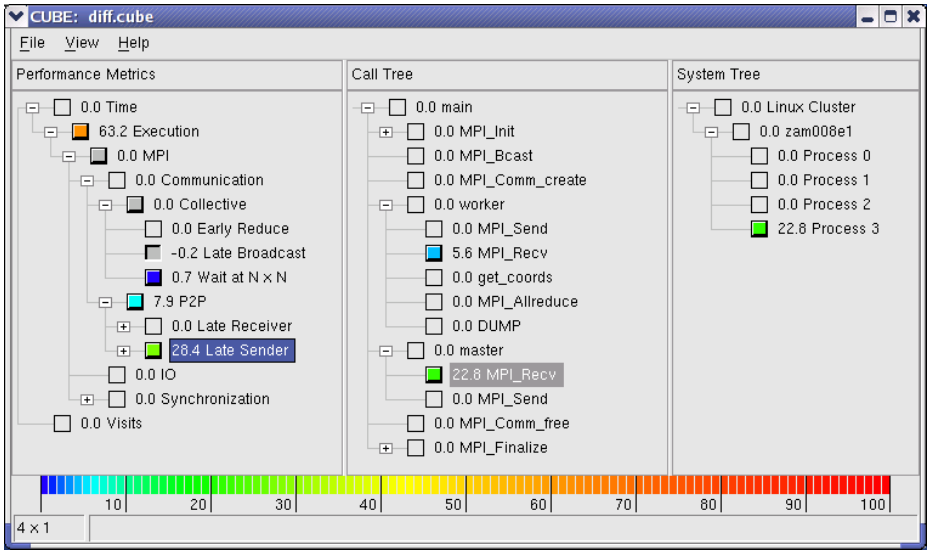


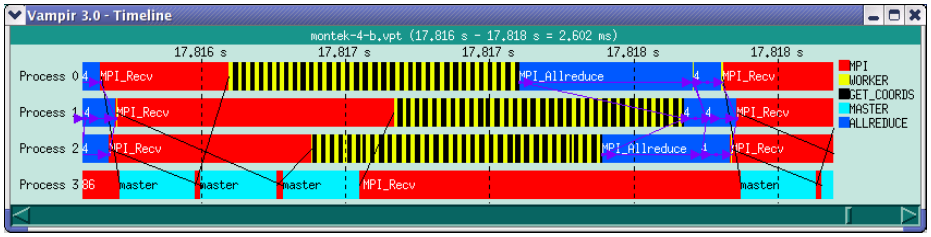
Fig. 2. Composition of overhead for 4 processors.

workers processing a chunk of random numbers. The black stripes indicate calls to `get_coords()`, the function mainly responsible for intrusion. At the bottom, the master process (process 3) performs three send operations - one for each of the three workers, after which it starts waiting in a receive call for response. The wait state lasts until the end of the collective call performed by all workers to determine the progress of the computation.

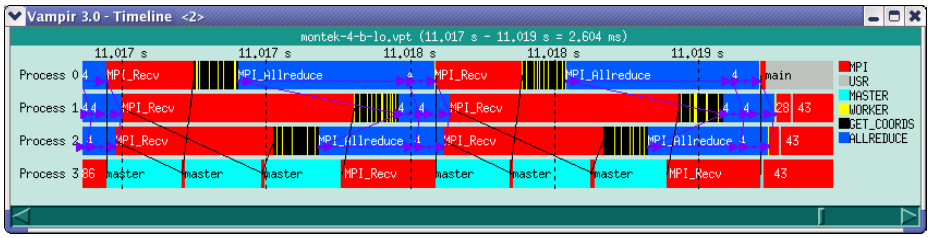
Figure 3(b), in contrast, shows about the same amount of execution time of the lower-bound approximation. The iteration on the left finishes significantly earlier. Most notable is, however, that as a result of the reduced overhead visible on the worker time lines, the wait state found on the master time line became significantly smaller, whereas the preceding send operations remained about the same. Thus, our example demonstrates that local perturbation effects can propagate across process boundaries and significantly distort the performance behavior of non-local processes.

6 Conclusion

Most parallel performance measurement tools ignore the overhead incurred by their use. Tool developers attempt to build the measurement system as efficiently as possible, but do not attempt to quantify the intrusion other than as a percentage slowdown in execution time. Our earlier work on overhead compensation in parallel profiling showed that the intrusion effects on the performance of events local to a process can be corrected [7] and it also modeled how local overheads affected performance delay across the computation [8]. However, parallel profiling only provides performance summary statistics. In order to see execution detail, tracing measurements must be used. This papers concerns the compensation of overhead during trace analysis.



(a) Measured execution.



(b) Approximated execution.

Fig. 3. VAMPIR time-line diagrams of measured and approximated execution.

The goal of trace analysis is to detect performance patterns and identify performance problems associated with certain patterns. It is important then that the timing properties of the trace data be as accurate as possible. Overhead can introduce intrusion that alters event timing structure and order, causing trace analysis to report performance problems where none are present in the "actual" execution, or to even mask performance problems that might otherwise appear.

The algorithms we designed and re-engineered in KOJAK can remove measurement overhead from a parallel trace. In doing so, we contend the performance properties [14] captured in the transformed trace data will be more representative of the performance behavior in a uninstrumented execution. The experiments presented here give powerful evidence to this conclusion. The Kojak analysis of the master-worker test case shows clearly the better performance problem identification as a result of overhead compensation. Figure 3 gives visual evidence to the improvement in detailed event time relations.

It is important to understand that we do not claim the compensated trace resulting from trace analysis is exactly the same as the trace of an uninstrumented execution, if that trace could be obtained without measurement overhead. It is even difficult to make quantitative statements about the bounds on analysis error. Indeed, the *performance uncertainty principle* [6] implies that the accuracy of performance data is inversely correlated with the degree of performance instrumentation. Our goal is to improve the tradeoff, that is, to improve the accuracy of the performance data through more intelligent trace analysis. What we are saying in this paper is that the performance results produced by applying our algorithms for trace-based overhead compensation will be more accurate that performance results produced without compensation. In addition,

we are providing this overhead compensation capability in state-of-the-art tracing tools that are being distributed to the parallel computing community.

7 Acknowledgements

This research is supported at the University of Oregon by the U.S. Department of Energy (DOE), Office of Science contract DE-FG02-05ER25680. At the University of Tennessee, this research is supported by the U.S. Department of Energy under Grants DE-FG02-01ER25510 and DE-FC02-01ER25490.

References

1. L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System," *CACM*, **21**(7), pp. 558-565 (July 1978).
2. A. Fagot and J. de Kergommeaux, "Systems Assessment of the Overhead of Tracing Parallel Programs," *Euromicro Workshop on Parallel and Distributed Processing*, pp. 179–186, 1996.
3. J. Hollingsworth and B. Miller, "An Adaptive Cost System for Parallel Program Instrumentation," *Euro-Par Conference*, Volume I, pp. 88–97, August 1996.
4. D. Kranzlmüller, R. Reussner, and C. Schaubschläger, "Monitor Overhead Measurement with SKaMPI," *EuroPVM/MPI Conference*, LNCS 1697, pp. 43–50, 1999.
5. A. Malony, "Event Based Performance Perturbation: A Case Study," *Principles and Practices of Parallel Programming (PPoPP)*, pp. 201–212, April 1991.
6. A. Malony, "Performance Observability," Ph.D. thesis, University of Illinois, Urbana-Champaign, 1991.
7. A. Malony and S. Shende, "Overhead Compensation in Performance Profiling," *Euro-Par Conference*, LNCS 3149, Springer, pp. 119–132, 2004.
8. A. Malony and S. Shende, "Overhead Compensation in Parallel Performance Profiling," *Parallel Processing Letters*, to be pubished, 2005.
9. Message Passing Interface Forum. MPI: A Message Passing Interface Standard, Chapter 8, Profiling Interface, Juni 1995. http://www.mpi-forum.org.
10. 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.
11. F. Song, F. Wolf, N. Bhatia, J. Dongarra, and S. Moore. An Algebra for Cross-Experiment Performance Analysis. In *Proc. of the International Conference on Parallel Processing (ICPP)*, Montreal, Canada, August 2004.
12. F. Wolf and B. Mohr. Automatic performance analysis of hybrid MPI/OpenMP applications. *Journal of Systems Architecture*, 49(10-11):421–439, 2003. Special Issue "Evolutions in parallel distributed and network-based processing".
13. F. Wolf. EARL - API Documentation. Technical Report ICL-UT-04-03, University of Tennessee, Innovative Computing Laboratory, October 2004.
14. F. Wolf and B. Mohr. Specifying Performance Properties of Parallel Applications Using Compund Events. *Parallel and Distributed Computing Practices*, 4(3), September 2001. Special Issue on Monitoring Systems and Tool Interoperability.