Compensation of Measurement Overhead in Parallel Performance Profiling

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

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

[2] John von Neumann Institute for Computing (NIC)
Forschungszentrum Jülich, Germany
Department of Computer Science
RWTH Aachen University, 52056 Aachen, Germany
f.wolf@fz-juelich.de

Abstract. Performance profiling generates measurement overhead during parallel program execution. Measurement overhead, in turn, introduces intrusion in a program's runtime performance behavior. Intrusion can be mitigated by controlling instrumentation degree, allowing a tradeoff of accuracy for detail. Alternatively, the accuracy in profile results can be improved by reducing the intrusion error due to measurement overhead. Models for compensation of measurement overhead in parallel performance profiling are described. An approach based on rational reconstruction is used to understand properties of compensation solutions for different parallel scenarios. From this analysis, a general algorithm for on-the-fly overhead assessment and compensation is derived.

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

1 Introduction

To observe the performance of a parallel program, the general technique of *performance profiling* is often used. Performance profiling Knuth (1971) can be implemented by direct (*in vivo* instrumentation of the program with measurement code (e.g., see Fahringer and Seragiotto (2002); Mucci; Reed et al. (1998); Rose (2001); Shende and Malony (2006)), or by sampling (*ex vivo*) the program periodically (via interrupts or passive monitoring) to assign performance metrics to code regions identified by the sampled program counter (e.g., see Graham et al. (1982); IBM; Janssen; Laboratories (1979); Mellor-Crummey et al. (2002). The first technique is commonly referred to as *measurement-based profiling* (or simply *measured profiling*) and is an *active* technique. The program is instrumented to observe specific events of interest, such as the entry and exit of a routine. 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 the program. The "events" observed in this case are considered to be either those events inferred from the program "state" at the time of sampling (e.g., the current routine on the call stack) or those events that triggered the sampling action (e.g., overflow of a hardware counter) or a combination of both.

In both statistical and measured profiling, performance measurements are made during program execution. There is an *overhead* associated with performance measurement since extra code is 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* Malony (1991b) 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 different 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 profile results.

Advocates of statistical profiling argue against measured profiling in regards to time overhead and its impact on intrusion. The insertion of measurement code into the program introduces direct overhead for *every* event occurrence at the point in the program's execution when and where the event occurs. In contrast, statistical profiling measures the the program's performance only at the time sampling, whenever and wherever that occurs. Hence, it is argued, statistical profiling can result in less overhead and, consequently, less intrusion than measured profiling. Measured profiling advocates retort that, while this may be true in some cases, there are certain classes of common events and execution actions that cannot be observed by a statistical approach alone or at all. Even Unix gprof-style profiling Graham et al. (1982), typically implemented via program counter sampling, requires direct code measurement to keep track of the number of times an event (i.e., routine entry) occurs during execution.

Any performance profiling technique, short of completely passive monitoring, will encounter measurement overhead and will also have limitations on what performance phenomena can and cannot be observed Malony (1991b). Until there is a systematic basis for judging the validity and verifiability of differing profiling techniques, it is more productive to focus on those challenges that a profiling method faces to improve the accuracy of its measurement. In this regard, we pose the question whether it is possible to compensate for measurement overhead in performance profiling. What we mean by this is to quantify measurement overhead and remove the overhead from profile calculations. (It is important to note we are not suggesting that by doing so we are "correcting" necessarily the effects of overhead on intrusion and perturbation.) Because performance overhead occurs in both measured and statistical profiling, overhead compensation is an important topic of study.

In Malony and Shende (2004), we presented overhead compensation techniques that were implemented in the TAU performance system Shende and Malony (2006) and demonstrated with the NAS parallel benchmarks Bailey et al. (1995) for both flat and callpath profile analysis. While our results showed improvement in NAS profiling accuracy, as measured by the error in total execution time compared to a non-instrumented run, the compensation models were deficient for parallel execution due to their inability to account for interprocess interactions and dependencies. The contribution of this paper is the modeling of performance overhead compensation in parallel profiling and the design of on-the-fly algorithms that would be implemented in practical profiling tools, such as in the TAU system.

Section §2 briefly describes the basic models from Malony and Shende (2004) and show how they fail on a simple master-worker example. We discuss the difficult issues that arise with overhead interdependency in parallel execution. In Section §3 we follow a strategy to model parallel overhead compensation for message-based parallel programs based on the notion of *rational reconstruction* of compensation solutions for specific parallel case studies. From the rationally reconstructed models we generate, we derive an on-the-fly algorithm for overhead analysis and compensation and argue for its general application. The simple master-worker example is then reconsidered in Section §4 in respect to the application of the algorithm. Conclusions and future work are given in Section §5.

2 Basic Models for Overhead Compensation

In our earlier work Malony and Shende (2004), we developed techniques for quantifying the overhead of performance profile measurements and correcting the profiling results to compensate for the measurement error (i.e., overhead) introduced. This work was done for two types of profiles: flat profiles and profiles of routine calling paths. The techniques were implemented in the TAU profiling system and demonstrated on the NAS parallel benchmarks. However, the models we developed were based on a local perspective of how measurement overhead impacted the program's execution. Profiling measurements are, typically, performed for each program thread of execution. (Here we use the term "thread" in a general sense. Shared memory threads and distributed memory processes equally apply.) By a local perspective we mean one that only regards the overhead impact on the process (thread) where the profile measurement was made and overhead was incurred.

Consider a message passing parallel program composed of multiple processes. Most profiling tools would produce a separate profile for each process, showing how time was spent in its measured events. Because the profile measurements are made locally to a process (either through direct instrumentation or sampling), it is reasonable, as a first step, to compensate for measurement overhead in the process-local profiles only. Our original models did just that. They accounted for the measurement overhead generated during TAU profiling for

each program process (thread) and all its measured events, and then removed the overhead from the inclusive and exclusive performance results calculated during online profiling analysis. The compensation algorithm "corrected" the measurement error in the process profiles in the sense that the local overhead was not included in the local profile results.

Indeed, the models we developed 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 due to measurement overhead seen on different processes will be interdependent. We use the term "intrusion" specifically here to point out that although measurement overhead occurs locally, its intrusion can have non-local effects.

Suppose our message passing program implements a master-worker computation and consider the following execution scenario. The 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 and terminate. The master finishes once all the worker results are received. Figure 1(a) depicts the above program execution with profiling enabled. The arrows show the master-to-worker and worker-to-master message communication and the local measurement overheads are shown by rectangles. The small triangles indicate when the worker messages are received by the master and the large triangle marks where the master terminates. Worker termination is depicted by the vertical line. Since no overhead compensation is performed in Figure 1(a), the total execution (worker and master) time is delayed. As shown, the overhead for the master process is assumed to be negligible, since for most of the time it is waiting for the workers to report their results.

Figure 1(b) portrays the effects of our original overhead compensation algorithms. Each worker is slowed down due to measurement overhead with the last worker to report seeing a 30% slowdown approximately. If we assume the local profile compensation works well and accurately approximates the workers "actual" performance, all the worker overhead will be removed. This is depicted by the overhead bunched up at the end of each worker's measured timeline. As a result, the messages returning results from the workers would have been sent earlier (dashed arrows) and the workers would have finished earlier, as if no measurements had been made. 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. Unfortunately, the master knows nothing of the worker overheads and, thus, our "local" compensation algorithms cannot account for them. The master's profile will still reflect the master finishing at the same time point (white large triangle), even though its "actual" termination point is much earlier (grey large triangle).

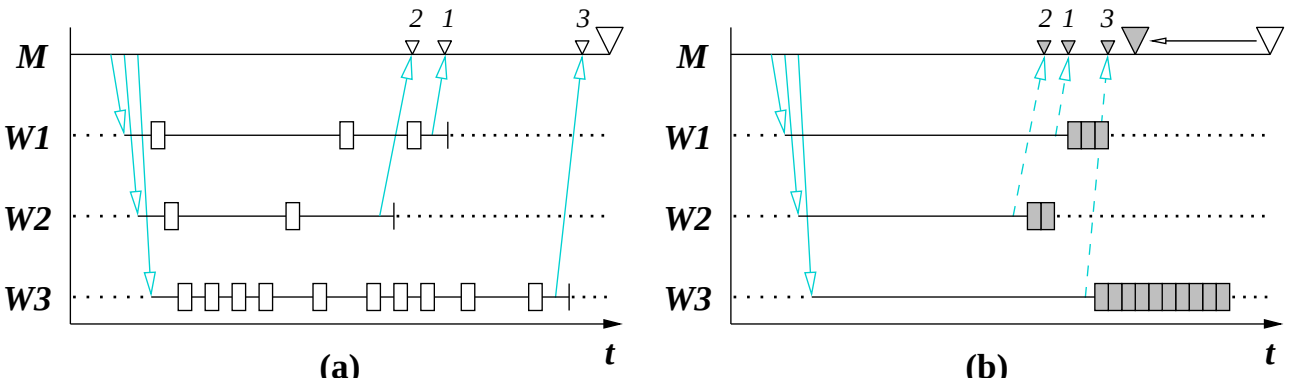


Fig. 1. Parallel Execution Measurement Scenario.

Parallel overhead compensation is a more complex problem to solve. This is not entirely unexpected, given our past research on performance perturbation analysis Malony (1991a); Malony and Reed (1991); Malony et al. (1992). However, in contrast with that work, we do not want to resort to post-mortem 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 very restricted form in Cray's MPP Apprentice system Williams et al. (1994). Certainly, we can learn from techniques for trace-based perturbation analysis Sarukkai

and Malony (1993), but because we must perform overhead compensation on-the-fly, the utility of these trace-based algorithms will be constrained to deterministic parallel execution only, for the same reasons discussed in Malony (1991b); Sarukkai and Malony (1993).

At a minimum, algorithms for on-the-fly overhead compensation in parallel profiling must utilize measurement infrastructure that conveys overhead information between processes at runtime. It is important to note this is not required for trace-based perturbation analysis and is what make compensation in profiling a unique problem. Techniques similar to those used in PHOTON Vetter (2002) and CCIFT Bronevetsky et al. (2003a,b) to embed overhead information in MPI messages may aid in the development of such measurement infrastructure. Photon extends the MPI header in the underlying MPICH implementation to transmit additional information. In contrast, the MPI wrapper layer in the CCIFT application level checkpointing software allows this information to piggyback on each message.

However, we first need the understand how local measurement overhead affects global performance intrusion so that we can construct compensation models and use those models to develop online algorithms.

3 Models of Parallel Overhead Compensation

To address the problem of overhead compensation in parallel execution, we must develop models that describe the effect of measurement overhead on execution intrusion. From these models we can gain insight in how the profiling overheads can then be compensated. However, unlike sequential computation, the models must identify and describe aspects of parallel interaction that may cause different intrusion behavior and, thus, lead to different methods for compensation. From the discussion above, we know that the methods will involve the communication of information between parallel threads of execution at the time of their interaction. To be more focused in our discussion, we will consider parallel compensation in message passing computation. The parallel overhead compensation models we present below allow for information about execution delay to be passed between processes during message communication. The goal is to determine exactly what information needs to be shared and how this information is to be used in compensation analysis. The modeling methodology we develop extends to shared memory parallel computing, but the case for shared memory will not be presented here.

The approach we follow below constructs an understanding of the parallel compensation problem from first principles. We begin with only two processes and then consider three processes. From this in-depth study, our hope is to gain modeling and analyses understanding that can extend to the general case. We will follow a strategy of "rational reconstruction" where we take scenarios measurement cases and reconstruct an "actual" execution as if the measurement overhead were not present. From what we learn, we derive a model that works for that case and look for consistent properties across the models to formulate a general algorithm.

3.1 Two Process Parallel Models

The simplest parallel computation involves only two processes which exchange messages during execution. Measurement-based profiling will introduce overhead and intrusion local to each process that carries between the processes as they interact. To model the intrusion and determine what information must be shared for overhead compensation, we consider the following two-process scenarios:

One send	Process P1 sends one message to process P2
Two sends	P1 sends two messages to P2
Handshake	P1 sends one message to P2, then P2 sends one message to P1
General	General message send and receive

For each scenario, we enumerate all possible cases for overhead relations between the processes (what is called the "measured execution" model) and for each case derive a representation of the execution with the overhead removed (what is called the "approximated execution" model). We determine the overhead-free approximation using a *rational reconstruction* of the "actual" event timings with the measurement overhead removed.

Both models are presented in diagrammatic form. In addition, we present expressions that relate the overhead, waiting, and timing parameters from the measured execution to those "corrected" parameters in the

approximated execution. It is important to keep in mind that the goal is to learn from the rational reconstruction of the approximated execution how profile compensation is to be done in the other scenarios, especially the general case.

Scenario: One Send To begin, consider a single message sent between two processes, P1 and P2. Figure 2 shows the two possible cases, distinguishing which process has accumulated more overhead up until the time of the message communication. Execution time advances from left to right and shown on the timelines are send events (S) and receive events (Rb, receive begin; Re, receive end). The overhead on P1 is $o1$ and the overhead on P2 is $o2$. The overhead is shown as a blocked region immediately before the S or Rb events to easily see its size in the figure, but it is actually spread out across the preceding timeline where profiled events occur. Also designated is the waiting time (w) between Rb and Re, assuming waiting time can be measured by the profiling system.

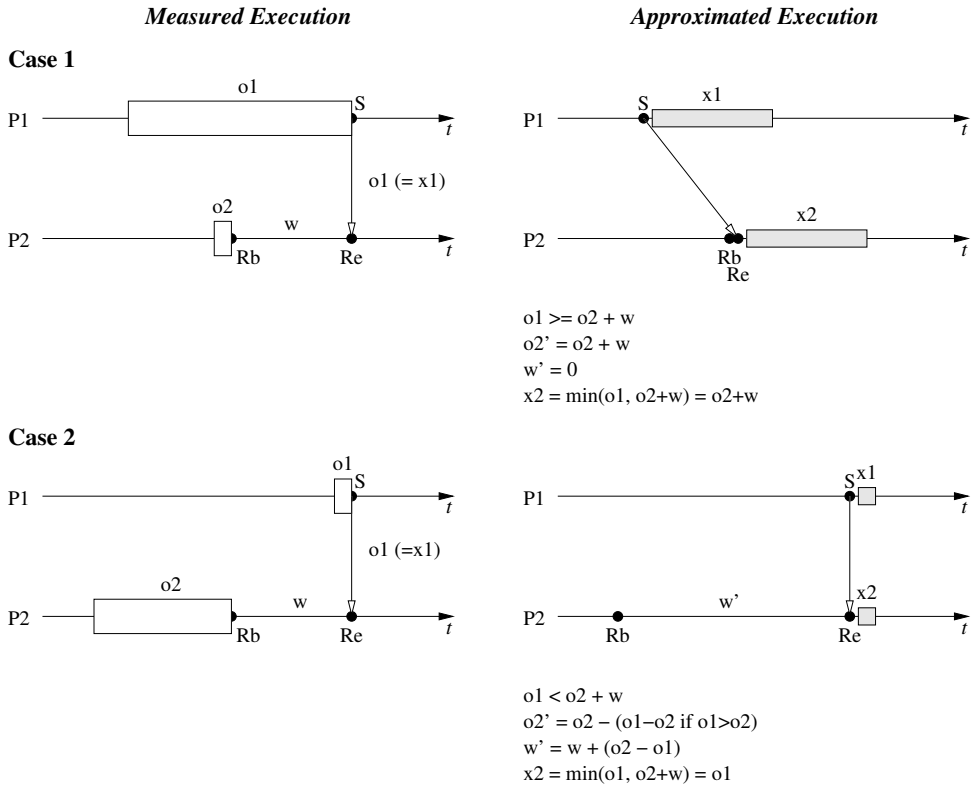


Fig. 2. Two-Process, One-Send – Models and Analysis

Case 1 occurs when P1's overhead is greater than or equal to P2's overhead plus the waiting time ($o1 \geq o2 + w$). A rational reconstruction of the approximated execution determines that P2 would not have waited for the message (i.e., S would occur earlier than Rb). Hence, the approximated waiting time (designated as w') should be zero, as seen in the approximated execution timeline. Of course, the problem is that P2 has already waited in the measured execution for the message to be received. In order for P2 to know P1's message would have arrived earlier, P1 must communicate this information. Clearly, the information is exactly the value $o1$, P1's overhead. This is indicated in the figure by tagging the message communication arrow with this value.

With P1's overhead information, P2 can determine what to do about the waiting time. The waiting time has already been measured and must be correctly accounted. If the approximated waiting is adjusted to zero,

where should the elapsed time represented by w go? If the profiling overhead is to be correctly compensated, the measured waiting time must be attributed to P2's approximated overhead ($o2' = o2 + w$)! This is interesting because it shows how, even in a very simple case, the naive overhead compensation can lead to errors without conveyance of delay information between sender and receiver. It is also important to note that *Rb* cannot be moved back any further in the approximated execution. This suggests that the only correction we can *ever* make in the receiver is in respect to waiting time.

The overhead value sent by P1 with the message conveys to P2 the information "this message was delayed being sent by $o1$ amount of time" or "this message would have been sent $o1$ time units earlier." We contend that this is exactly the information needed by P2 to correctly adjust its profiling metrics (i.e., compensate for overhead in parallel execution). We refer to the value sent by P1 as *delay* and will assign the designator x to represent its modeling and analysis that follows. For instance, P1's delay is given by $x1$. In this case, $x1 = o1$, but it is not always true that delay will be equal to accumulated overhead, as we will see. Now an interesting question arises. How much earlier would future events on process 2 occur in the approximated execution after the message from P1 has been received? In general, each process will maintain a delay value (xi for process Pi) for it to include in its next send message to tell the receiving process how much earlier the message would have been sent. In the approximated execution, for denotational purposes, we show the $x1$ and $x2$ values for P1 and P2 as shaded regions after the last events, S and Re, respectively. We also show an expression for the calculation of $x2$ for this case.

Moving on to the second case, the overhead and waiting time in P2 is greater than what P1 reports (i.e., $o1 < o2 + w$). Rationally, this means that S happens after *Rb* in the approximated execution. What is the effect on w', the approximated waiting time? Interesting, w' can increase or decrease, depending on the relation of $o1$ to $o2$. (Remember, $o1$ is the same as $x1$ in these cases.) However, the occurrence of *Re* is certainly dependent on S and, thus, $x2$ will be entirely determined by (and, in fact, equal to) $x1$.

If this was all the two processes did, the models and analysis expressions would be all we need. Unfortunately, life is not so simple. Let us add in a bit more complexity.

Scenario: Two Sends To gain a better understanding of this concept of a process carrying forward a *delay* quantity (x) indicating how much earlier the next future send event would occur, we look at the scenario of sending two messages. This is shown in Figure 3. There are four cases to consider. In each case, we show the analysis of the first message following the modeling methods demonstrated above. Then we consider the second message. Notice the inclusion of additional profiling overheads in the two processes after the first message ($o1a$ and $o2a$) as well as a second waiting term (y) in P2.

Consider Case 1. The first message is approximated exactly as in Case 1 of the single send model. P1's second send event is easily approximated by subtracting the accumulated overheads. P1's delay is exactly this accumulated overhead ($o1 + o1a$) and this information should be sent to P2, as shown. P2's second receive begins a known amount of time after the first receive completes. However, to correctly compensate for profiling overhead, P2 must be able to determine the relative timing of the second *Rb* event and the second S event.

With the $x2$ delay calculation from the first receive, P2 has all the information it needs. By comparing the value $o1 + o1a$ sent with the second message to $x2$ plus any additional overhead and waiting time on P2 ($o2a + y$), we can determine if the send event or the receive event occurred earlier in the approximated execution. The second message analysis introduces two new variables ($O1$ and $O2$), to bring out the similarities in the expressions for the *One Send* models. As seen, P2 should never wait in its approximated execution. Thus, its immediate future events would occur earlier by an amount based solely on its accumulated overhead and the waiting time it erroneously incurred. This analysis is successfully captured with the expressions shown. Interestingly, these expressions have a strong similarity to the *One Send* cases.

Does this similarity continue to hold for Case 2? Here, we have the alternate first message case (i.e., *Rb* occurs before S in the approximated execution), and the one send analysis determines the value of $x2$. However, our rational reconstruction leads to second message equations that are exactly the same as in Case 1. That is, after the effects of the first message have been approximated, we see that P2 would not have waited for the second message, just as in Case 1.

The similarity in the form of the equations suggest that it may be possible to handle the overhead compensation on a message-by-message basis. We argue that the x delay values maintained by a process are the key.

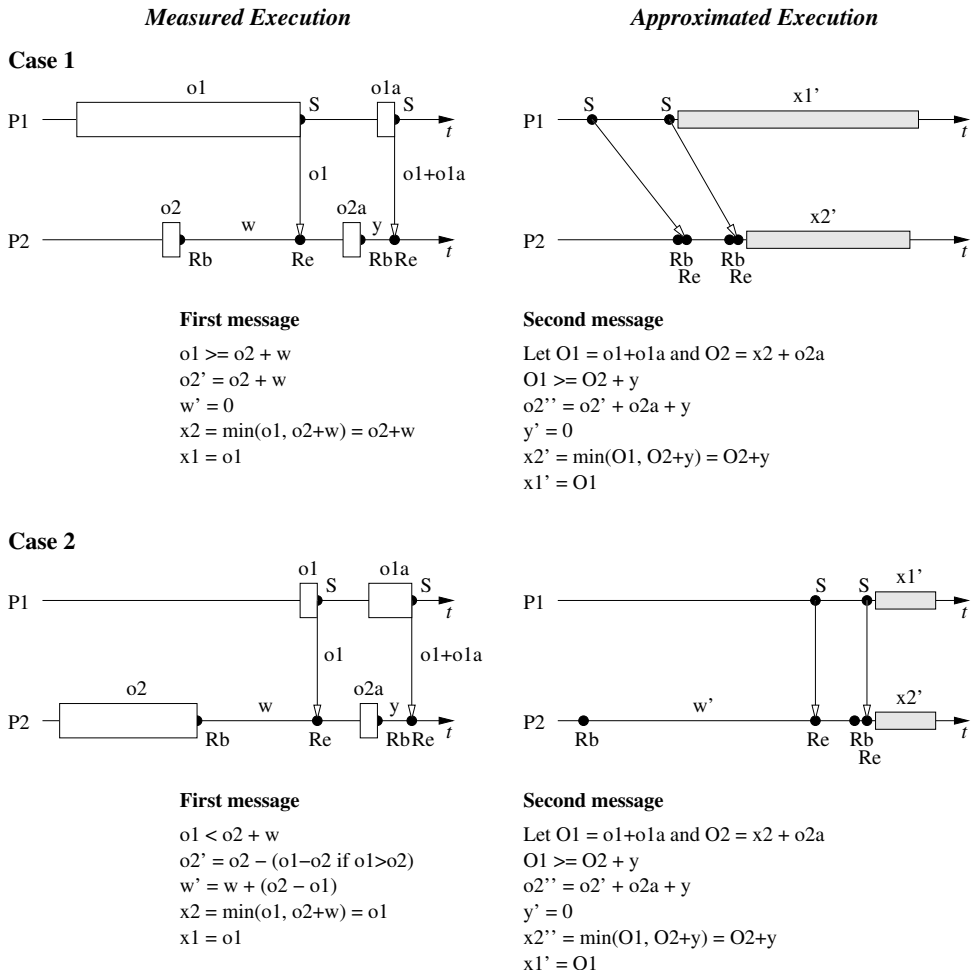


Fig. 3. Two-Process, Two-Send – Models and Analysis (Case: 1, 2)

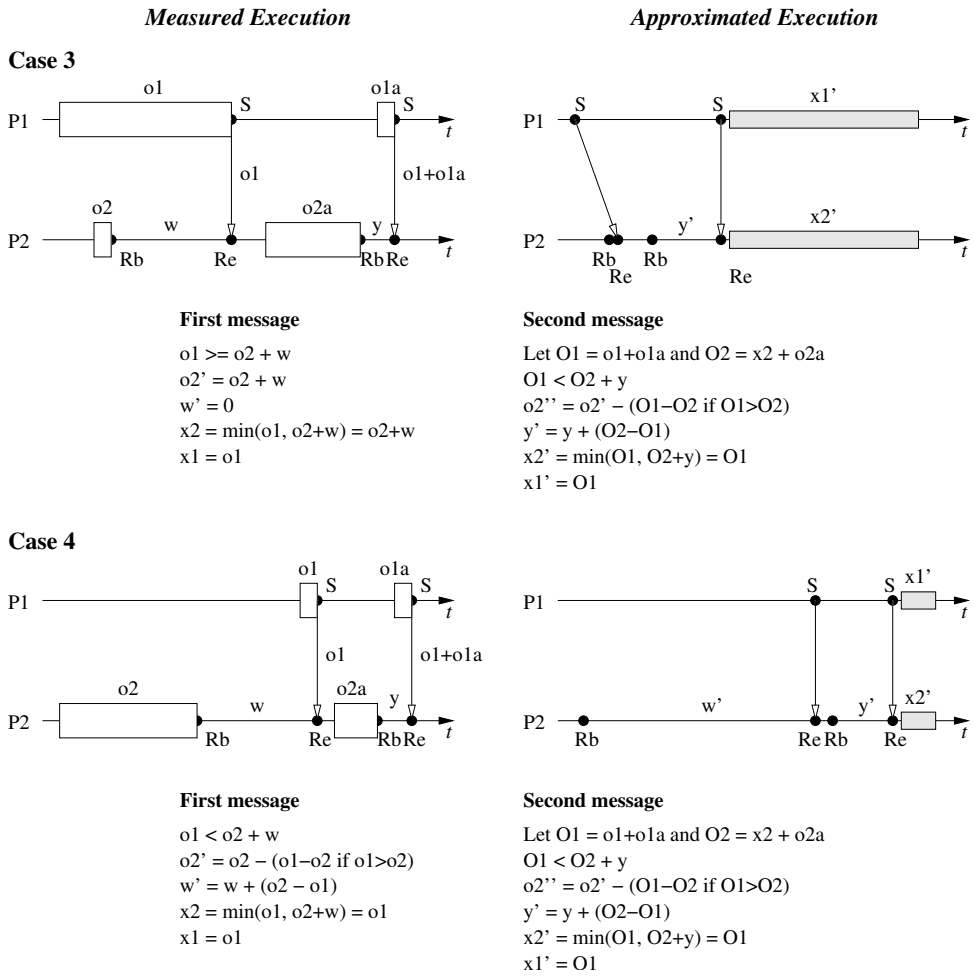


Fig. 4. Two-Process, Two-Send – Models and Analysis (Case: 3, 4)

This is evidenced in the last two cases for the *Two Sends* scenario shown in Figure 4. These two cases differ from the first two in respect to the outcome of the conditional test between $O1$ and $O2 + y$. Here we consider two situations where waiting will occur in the second message approximation. Following the expression pattern for processing this case in the *One Send* scenario, the equations result in a correct updating of the overhead and waiting values. In addition, the values of $x1'$ and $x2'$ are consistently calculated.

Scenario: Handshake So far, we have looked only at cases of one process, P1, sending messages to another process, P2. Here, the execution dependency is only one way, with P2 dependent on arriving P1 messages. While the overhead analysis handles the two scenarios above, it does not address cases of process interdependencies. The simplest example of such a scenario is a two message "handshake" where P1 sends a message to P2 and P2 then sends a message to P1. When we add in overhead and waiting variables, four cases result. These are presented below. Specifically, what are we looking for in this scenario is insight on what value P2 should send back to P1 for overhead compensation analysis.

Figure 5 shows the first two cases. The handling of the first message sent from P1 to P2 is consistent with that of Cases 1 and 2 of the *Two Sends* scenario. After the first message is processed, the calculated value of $x2$ reflects a delay in P2 caused by the combined effects of overhead intrusion on P1 and P2. Thus, when P2 sends the message to P1, it should indicate that its send was delayed by $x2$ plus any additional overheads incurred (given by $o2a$) since the last receive. If this is done, we claim the second message received on P1 can then be processed in same manner as the second message received in the *Two Sends* scenario.

This is exactly what we see. The approach validates the rational reconstruction of the approximated execution. Furthermore, we see a high degree of similarity in the equations for processing the second message with those for handling a single message, just the terms are slightly different, rewritten as in the *Two Sends* scenario and for the inverse situation of P1 receiving and P2 sending. These observations are born out in Cases 3 and 4, shown in Figure 6 for completeness.

As further validation of the equations for overhead, waiting, and delay processing, we would expect to see the execution interdependency due to the message handshake result in a synchronization of sorts between the two processes. This is apparent in the fact that the two delay values ($x1'$ and $x2'$), when added to last events (Re on P1 and S on P2), end at the same time.

General Scenario The goal of the two process models is to enumerate the possible cases arising from send/receive message communication. From these cases, we can rationally reconstruct the approximated execution to determine how overhead, waiting, and delay times are to be adjusted. From this reconstruction, we can then derive expressions for overhead analysis and correction. The outcome of the study is that we found a high degree of similarity in the analysis results. This leads us to propose a general scenario for two processes that can be applied to all scenario analyses.

The general scenario considers an arbitrary message send on one process and corresponding message receive on the other process. Thus, this is a generalization of the *One Send* scenario. Figure 7 shows the two cases. Similar to the *One Send* scenario, we only have two cases to consider. In contrast, we use the delay values $x1$ and $x2$ instead of the $o1$ and $o2$ overheads in the analysis.

The importance of the general scenario is the case analysis showing how the delay values are updated and what information is shared between processes during message communication. (Keep in mind that we are arbitrarily designating P1 as the sender and P2 as the receiver. The analysis also applies when P1 is the receiver and P2 the sender, with appropriate reversals of notation in the expressions.) Notice that the overhead values $o1$ (not shown) and $o2$ are accumulated overheads. The $o2$ value is updated here to account for waiting time processing, but whenever any new measurement overhead occurs on P1 or P2, the accumulated overheads $o1$ and $o2$ must be updated accordingly. Similarly, any new measurement overhead must also be added to the delay values $x1$ or $x2$.

The conclusion of the two process modeling is that we can handle the parallel overhead compensation for ALL two-process scenarios by applying the general analysis described above on a message-by-message analysis, maintaining the overhead and delay values as the online analysis proceeds.

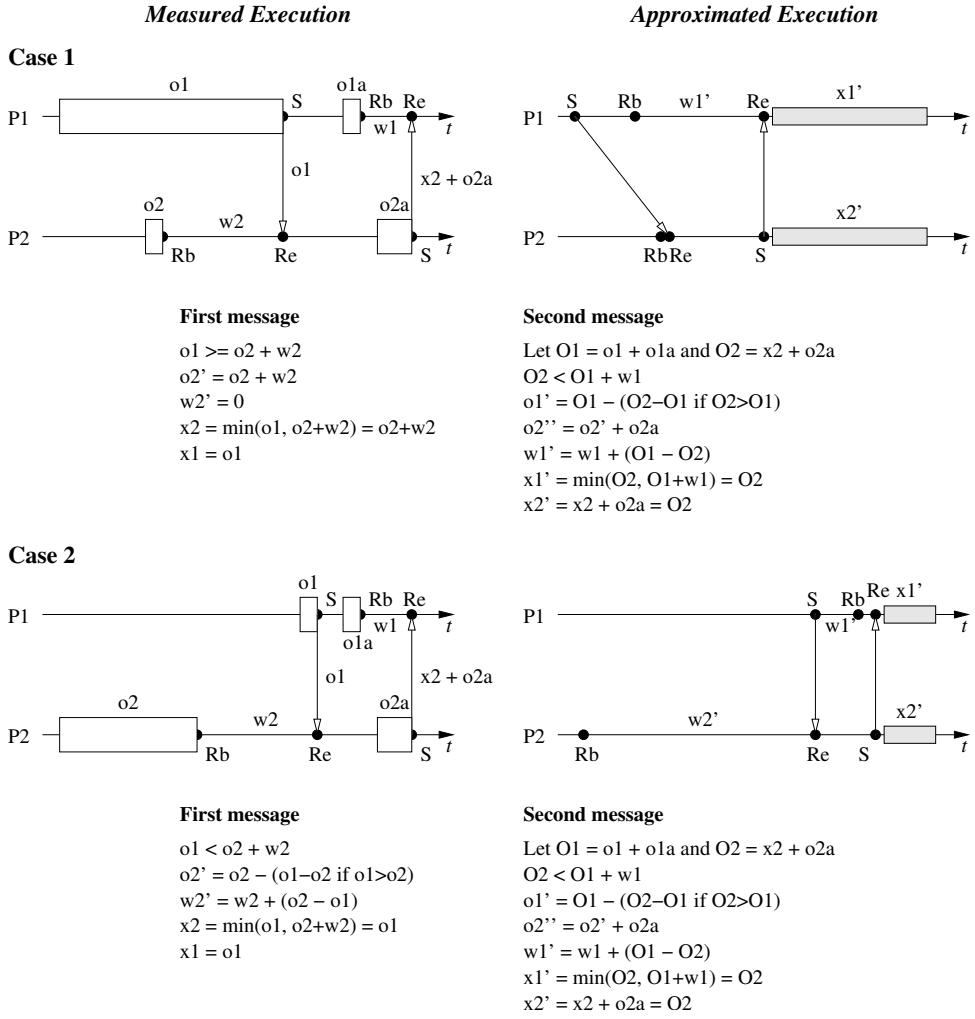
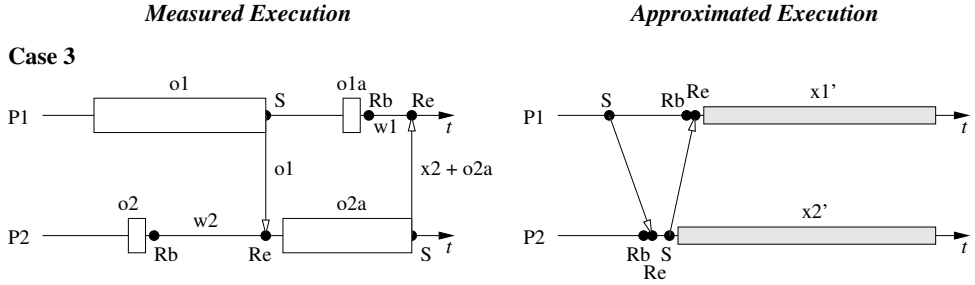


Fig. 5. Two-Process, Handshake – Models and Analysis (Case: 1, 2)

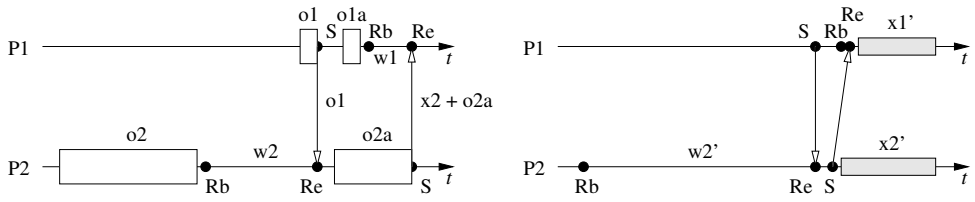


First message

o1 >= o2 + w2
o2' = o2 + w2
w2' = 0
x2 = min(o1, o2+w2) = o2+w2
x1 = o1

Second message

Let O1 = o1 + o1a and O2 = x2 + o2a
O2 >= O1 + w1
w1' = 0
o2'' = o2' + o2a
o1' = O1 + w1
x1' = min(O2, O1+w1) = O1 + w1
x2' = x2 + o2a = O2



First message

o1 < o2 + w2
o2' = o2 − (o1−o2 if o1>o2)
w2' = w2 + (o2 − o1)
x2 = min(o1, o2+w2) = o1
x1 = o1

Second message

Let O1 = o1 + o1a and O2 = x2 + o2a
O2 >= O1 + w1
w1' = 0
o2'' = o2' + o2a
o1' = O1 + w1
x1' = min(O2, O1+w1) = O1 + w1
x2' = x2 + o2a = O2

Fig. 6. Two-Process, Handshake – Models and Analysis (Case: 3, 4)

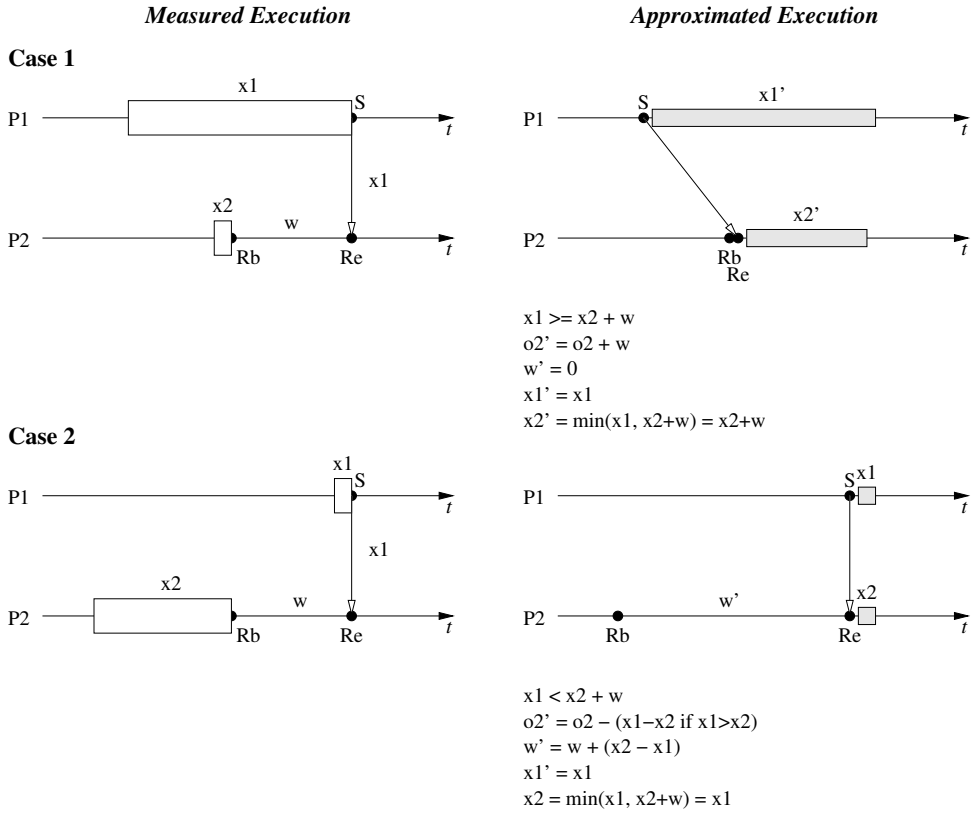


Fig. 7. Two-Process, General – Models and Analysis

3.2 Three Process Parallel Models

The question at this point is whether that conclusion applies to three or more processes. That is, can the general two-process analysis be applied on a message-by-message basis to all send/receive messages between any two processes in a multi-process computation and, more importantly, give the desired overhead compensation result? In this section, we look at two scenarios with three processes to get a sense of the answer. These scenarios are:

Pipeline	Process P1 sends a message to P2, then P2 sends to P3
Two Receive	Process P1 and P3 sends a message each to process P2

We then argue that these two scenarios are enough to elucidate all similar cases regardless of the number of processes. Again, we follow a rational reconstruction approach to determine approximated executions and then derive expressions for updating overhead, waiting time, and delay variables to match the reconstructed executions.

Scenario: Pipeline The *Pipeline* scenario is chosen to investigate the effects of delay propagation. The main issue is whether the calculated delay value from the processing of the first message is sufficient to correctly adjust the variables when the second message is processed. Figure 8 shows the two cases based on the relationship of the adjusted delay value ($x2'$) sent by P2 to P3, and the overhead and waiting time on P3. (Like our earlier models, we assume here for simplicity that this is the first message P3 receives. Clearly, in this case, $x3 = o3$.)

The interesting outcome of the models is that the analysis and update of P3's variables during the processing of the second message is effectively independent of the first message. Of course, the $x2'$ delay value sent from P2 is derivative of the effects of the first message and $x1$, but the expressions are invariant compared to those when P2 sends a message to P3 without first receiving a message from P1 (i.e., the *Two-Process, General* scenario). This conclusion extends to cases where there is an arbitrary number of processes in the pipeline.

Scenario: Two Receive Unfortunately, it is not hard to find a scenario that raises issues in our ability to correct overhead intrusion under a different set of receive assumptions. These issues are brought on by the effect of intrusion on message sequencing. The *Two Receive* scenario exposes the problem. Here one process, P2, receives messages from two other processes. There are four cases to consider depending on the relatives sizes of overheads and waiting times. Figure 9 show the first case. We return to looking only at the first messages being sent and received on each process, and consider the initial overheads (not the delays values) in the analysis.

The first case is similar in many ways to the other scenarios. We show a two-part approximated execution, with part one (top) showing the state after the first message is processed and part two (bottom) showing the result after the second message is processed. The analysis follows the approach we used before, with new waiting values (w' and y') being calculated and P2's delay value ($x2$) updated. In this case, no waiting time would have occurred. Otherwise, nothing particularly strange stands out in the approximated result.

This is also true of the second case shown in Figure 10. The conditional expression in the analysis of the first message is the same as in the first case, but the opposite condition is found for the second message. Waiting occurs on the second message and will be correctly adjusted based on the analysis expressions we developed before. Case 3, shown in Figure 11, also offers no surprises. Here we see the opposite condition for the first message with the same condition for the second message of Case 2.

What would be a surprising result? If the overhead analysis resulted in a reordering of send events in time, between the measured execution and the approximated execution, then there would be concerns of performance perturbation. In Figure 12, we see the send events changing order in time in the approximated execution, with P3's send taking place before P1's send. As with the other cases, our analysis reflects a message-by-message processing algorithm.

In the rational reconstruction, we assume the message communication is explicit and pairs a particular sender and receiver. Under this assumption, the order of messages received by P2 must be maintained in the approximated execution. In this case, is the time reordering of send messages in Figure 12 a problem? In fact, no. It is certainly possible that a process (P2) will first receive a message from a process (P1) sent after another process (P3) sends a message to the receiving process. This just reflects the strict order of P2 receives. However,

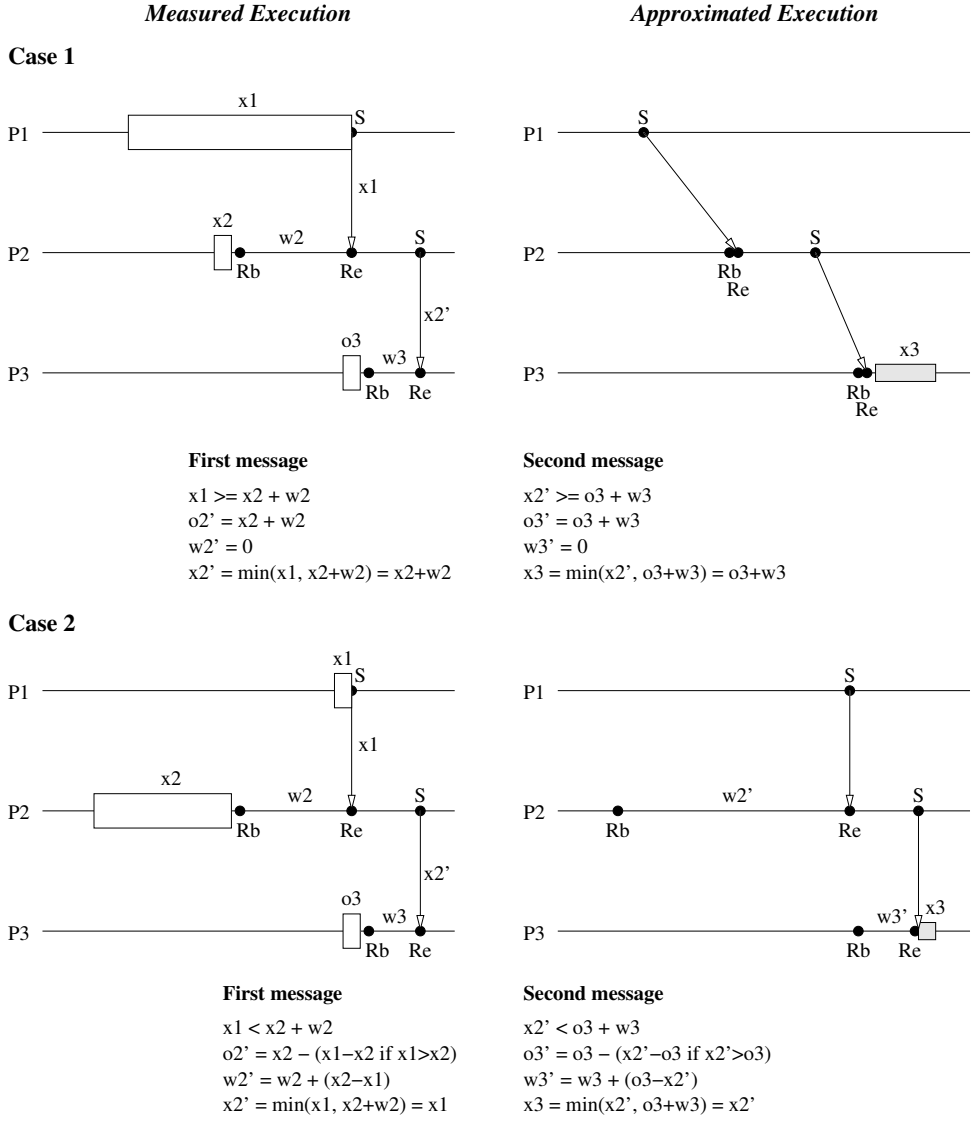


Fig. 8. Three-Process, Pipeline – Models and Analysis

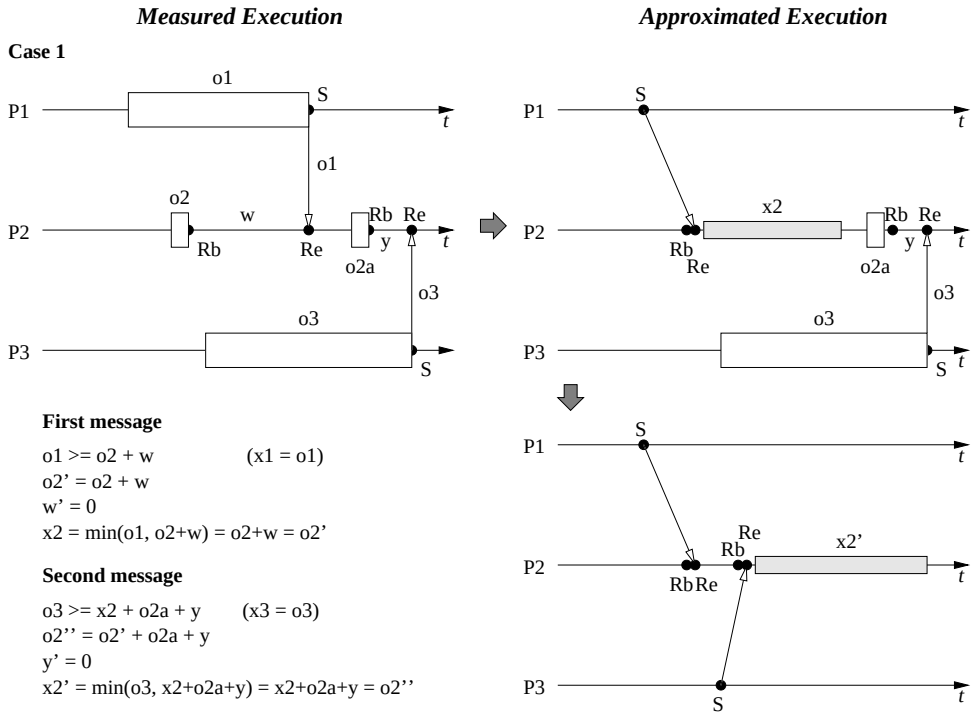


Fig. 9. Three-Process, Two Receive – Models and Analysis

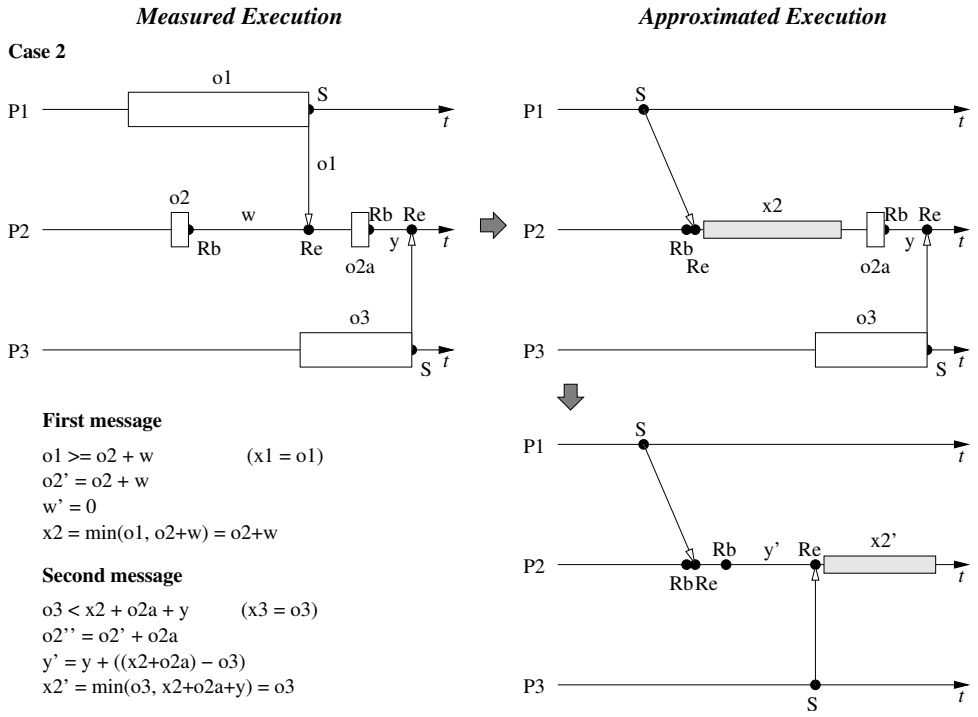


Fig. 10. Three-Process, Two Receive – Models and Analysis (Case: 2)

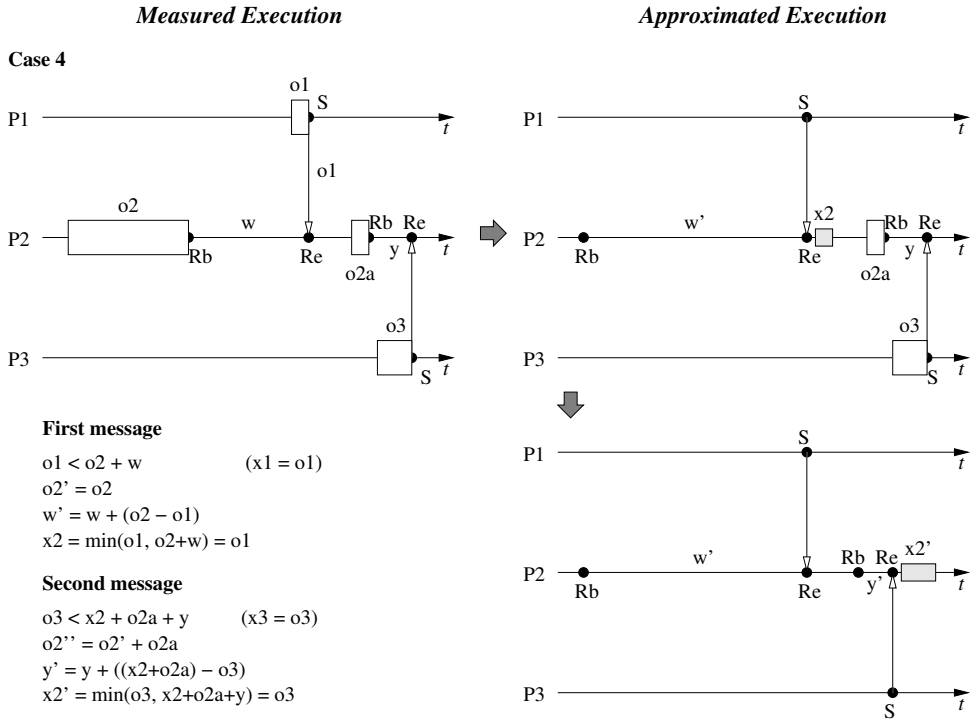


Fig. 11. Three-Process, Two Receive – Models and Analysis (Case: 3)

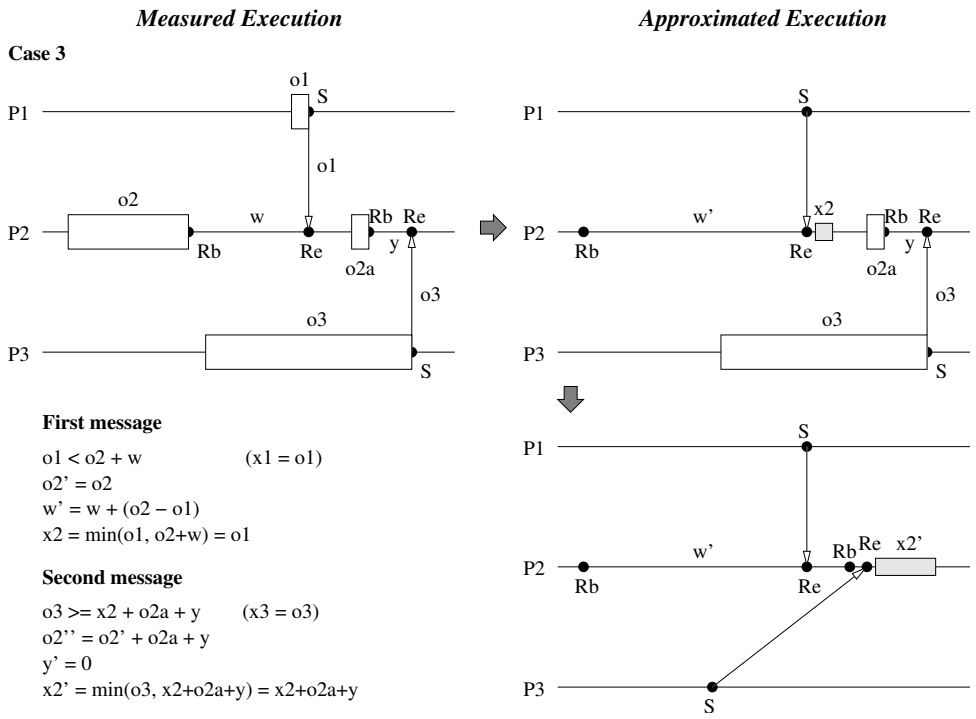


Fig. 12. Three-Process, Two Receive – Models and Analysis (Case: 4)

if we consider receive operations that can match any send, the send reordering exposes a problem with overhead compensation, since the message from P3 would be received first in the "real" execution.

The application of our overhead compensation models to programs using receive operations that can match any send message produces profile analysis results constrained to message orderings as observed in the measured execution. These message orderings are affected by intrusion and, thus, may not be the message orderings that occur in the absence of measurement. However, while it is possible to detect reordering occurrences (i.e., measured versus approximated orderings), it is not possible to correct for reordering during online overhead analysis and compensation. Why? There are two reasons. First, our analysis is unable to determine if it is correct to associate a receive event with a different send event. That is, the analysis does not know why type of receive is being performed, one that is for a specific sender or one that can accept any sender. Second, even if we know the type of receive operation, it is not possible to know whether changing receive order will affect future receive events. Therefore, the models must, in general, enforce message receive ordering. That said, other techniques may be brought to bear on the problem. For instance, the analysis could relax the constraint, allowing measured order to be locally altered, and then apply algorithms developed by Kranzmüller Kranzlmüller (2000) to look for future message event violations.

3.3 Summary

Our above modeling and analysis of measurement overhead in parallel message passing programs has produced four important outcomes. First, the rational reconstructions of the measurement scenarios and the analysis of the approximated executions has resulted in a robust procedure for message-by-message overhead compensation analysis in parallel profiling. It updates correctly waiting times associated with message processing and calculates per process values that capture online the amount a process has been effectively delayed due to measurement overhead and its effects. From this overhead compensation foundation, the parallel profiling operations used to update inclusive and exclusive performance can be applied. Second, this analysis requires ALL send messages to be augmented with the delay value of the sender process at the time the message is sent. This information is necessary for the receiving process to apply the analysis procedures. Third, the general two-process analysis covers the general multi-process analysis case with the understanding of the issues concerning compensation effects on message order. Fourth, approximation models based on receive type can result in more accurate overhead handling and profile results, but the accuracy gains are anticipated to be minor compared to the processing complexity involved.

4 Implementation

To test our models of parallel overhead compensation, we built a prototype using the TAU performance system Shende and Malony (2006) and the Message Passing Interface (MPI). Our goal was to produce a widely portable prototype that could be efficiently implemented and easily applied. We chose MPI as the communication substrate due to its wide acceptance in the parallel computing community as the de-facto message communication standard, as well as due to its portable tool support.

4.1 MPI Profiling Support

MPI supports creation of portable profiling and tracing tools using its profiling interface, PMPI. This interface allows a tool to interpose a library between the application and the MPI substrate and intercept one or more MPI calls. MPI provides a name-shifted interface to all its calls. For example, an MPI call such as `MPI_Send()` is also available as `PMPI_Send()`. Both are guaranteed by the MPI standard to provide the same functionality. Furthermore, if a tool defines an `MPI_Send()` call, it takes precedence over the MPI library's `MPI_Send()` call (this is done by using weak bindings for defining the library's calls). The tool can then define one or more MPI bindings and create measurement timers and start and stop them around the name-shifted version of the corresponding MPI call. Every MPI implementation must implement this profiling interface to conform to the MPI standard. This mechanism allows vendors of parallel systems to optimize the implementation of MPI to their target platforms and at the same time expose the hooks for tracking MPI performance to tool builders without providing them access to their proprietary source code.

4.2 Schemes to Piggyback Delay

To transmit the local delays encountered in a process (due to program instrumentation) to other processes, we examined several alternatives. The first scheme modifies the source code of the underlying MPI implementation by extending the header sent along with a message in the communication substrate (Photon Vetter (2002) uses this approach). Unfortunately, it is not portable to all MPI implementations and relies on a specially instrumented communication library. The second scheme sends an additional message containing the delay information for every data message. This scheme only requires changes to the portable MPI wrapper interposition library for the tool. While it is portable to all MPI implementations, it has a performance penalty associated with transmitting an additional message, a penalty not incurred by the first scheme. As a result, the overhead caused by the additional message would require further compensation.

The third scheme copies the contents of the original message and creates a new message with our own header that would include the delay information. This scheme has the portability advantage of the second scheme and avoids the second scheme's transmission of an additional message. However, copying contents of a message could prove to be an expensive operation, especially in the context of large messages that are transmitted in point-to-point communication operations.

We implemented a modification of the third scheme, but instead of building a new message and copying buffers in and out of messages (at the sender and the receiver), we create a new datatype. This new datatype is a structure with two members. The first member is a pointer to the original message buffer comprised of n elements of the datatype passed to the MPI call. The second member is a double precision number that contains the local delay value. Once created, the structure is committed as a new user-defined datatype and MPI is instructed to send or receive one element of the new datatype. Internally, MPI may transmit the new message by composing the message from the two members by using vector read and write calls instead of its scalar counterparts. This efficient transmission of the delay value is portable to all MPI implementations, sends only a single message, and avoids expensive copying of data buffers to construct and extract messages.

4.3 TAU Overhead Compensation Prototype

To test the validity of our parallel profile compensation models, we built the portable prototype within the TAU performance system Shende and Malony (2006). We previously implemented local overhead compensation, and now included the parallel compensation support. TAU computes parallel profile data during execution for each instrumented event. At runtime, TAU maintains an event callstack for each thread of execution. This callstack has performance information for the currently executing event (e.g., a routine entry) and its ancestors. We compute the delay that a process sees locally by first adding the number of completed calls to half the number of entries along the thread's callstack. We assume that an *enter* profile call takes roughly the same time as an *exit* profile call, which is true is most cases. Once we know the total number of timer calls and the total overhead associated with calling the enter and exit methods (see Malony and Shende (2004) for details), their product gives the local timer overhead. We keep track of adjusted wait times in a process, as explained earlier and subtract it from the local overhead to compute the local delay. This delay value is then piggybacked with a message.

Mapping MPI Calls The essence of our parallel overhead compensation scheme is that whenever two processes interact with each other, the receiver is made aware of the sender's delay value, or how much sooner the communication operation would have taken place in the absence of instrumentation. We have discussed above how this scheme operates for synchronous message communication operations using `MPI_Send` and `MPI_Recv`. In this section we explore how other MPI calls can be made aware of remote delays.

Asynchronous Operations When storing or retrieving the piggyback value, we create an auto variable on the stack in our wrapper routines for `MPI_Send` or `MPI_Recv`. Synchronization operations involve loads or stores to this variable. The logic to process the piggyback value when it is received is incorporated in the `MPI_Recv` wrapper routine. Here, we compare the local and remote delays to arrive at how much adjustment needs to be

made to the waiting time. Now let us examine the asynchronous `MPI_Isend` and `MPI_Irecv` calls. When the user issues the `MPI_Isend` call, we compute the local delay and create a global variable where this is stored. The location of this global piggyback variable in the heap memory is used when we create our struct for a new datatype for sending the message.

On the receiving side, a similar arrangement of the piggyback value is used. When the message is finally received, MPI automatically copies the contents of the piggyback value into the heap where this value is to be stored. We also create a map that links the address of the MPI request to the address of this piggyback value. The logic that compares the local and remote delays cannot be incorporated in the `MPI_Irecv` wrapper due to the very nature of the asynchronous operation (the values are not received when the routine executes). Hence, we do not adjust the time spent in `MPI_Irecv` as we did for `MPI_Recv`. Instead, an asynchronous message is visible to the program only after executing the `MPI_Wait`, `MPI_Test`, or variants of these calls (`Waitall`, `Waitsome`, `Testall`, `Testsome`) to wait for or test one or more requests. When a request is satisfied, we examine the map and retrieve the value of the piggyback variable where the remote process' overhead is stored. Then, a comparison of local and remote delays and an adjustment of waiting time is made on the receiving side. When more than one message is received by the process, we need to examine all the remote delays to determine how much time the process would have waited in the absence of instrumentation. We discuss this in more detail next with collective operations.

Collective Operations Consider the class of collective operations supported by MPI. Let us first examine the `MPI_Gather` call where each process in a given communicator provides a single data item to MPI. The process designated with the rank of root gathers all the data in an array. It is important to communicate the local delays from each process to the root process. To do this, we form a message with the piggyback delay value and call a single `MPI_Gather` call. At the receiving end, we receive a single contiguous buffer where the application data and the delay values are put together in a single buffer. We extract the piggyback values out of this buffer and construct the application buffer with the rest. Once we get an array of the delay values from each process we compute the minimum delay value from the group of processes. Since the collective operation cannot complete without the message with the minimum delay, it must adjust its waiting time based on this value. So, the collective operation reduces to the case where the receiver gets a message from one process that has the least delay in the communicator. We can now apply the performance overhead compensation model as described in the previous section.

When broadcasting a message from one task to several, `MPI_Bcast` is modeled based on the two process overhead compensation model (see Malony and Shende (2005)). We create a new datatype, on the root process, that embeds the original message and the local delay value. This message is sent to all other members of the group. Each receiver compares the remote delay with its local delay and makes adjustments to the waiting time and local overhead, as if it had received a single message from the remote task. We use the model described earlier to do this.

To model `MPI_Scatter`, which distributes a distinct message to all members of the group, we create a new datatype that includes the overhead from the root process. This is similar to the `MPI_Gather` operation. After the operation is completed, each receiver examines the remote overhead and treats it as if it had received a single message from the root node, applying our previous scheme for compensating for perturbation.

`MPI_Barrier` requires all tasks to block until all processes invoke this routine. `MPI_Barrier` is implemented as a combination of two operations: `MPI_Gather` and `MPI_Bcast`, sending the local delay from each task to the root task (arbitrarily selected as the process with the least rank in the communicator). This task examines the local delay and compares it with the task with the least delay, adjusts its wait time and then sends the new local delay to all tasks using the MPI_Bcast operation. This mechanism preserves the efficiencies that the underlying MPI substrate may provide in implementing a collective operation. By mapping one MPI routine to another, we exploit those efficiencies.

4.4 Caveats

Overhead compensation for parallel profiling *requires* transmitting delay information with messages. Doing so undoubtedly introduces more overhead in the process, in apparent contradiction to our goals. Our methods

do not adequately account for these overheads, nor is it obvious exactly how they can or should. While the approach described attempts to balance portability and efficiency concerns, its overhead in practice will depend on what the underlying MPI implementation does with datatypes, and it might do different things with different network interfaces. If the technique is deployed in production environments, it will be important to evaluate MPI implementations to determine their overhead effects.

5 Experimental Results

We validate our parallel performance intrusion compensation model using a prototype implemented within the TAU performance system. To illustrate the problem, we examine a parallel MPI application that computes the value of π using the Monte-Carlo integration algorithm. The program calculates the area under the π function curve ($\int_0^1 4/(1+x^2)\ dx$) 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 the chunk of randomly generated numbers to it. For each pair of numbers that is given to a particular worker, it finds out if the pair of cartesian co-ordinates represented by the numbers is below or above the π function curve. Then, collectively, the workers estimate the value of π iteratively until it is within a given error range. This simple example highlights how instrumentation overheads accumulated at the worker tasks are communicated to the master task. We execute the application in four modes: when there is no TAU instrumentation, with instrumentation without any compensation, with local perturbation compensation, and finally, with parallel perturbation compensation. As shown in table 1, these experiments are shown as distinct columns and we show the time spent in the worker and master tasks. We show the minimum times spent in the respective tasks. The timer overhead associated with a TAU timer was 480 nanoseconds on an Intel®Itanium2 Linux machine running at 1.5 GHz. The accuracy of compensation improves when we use high resolution timers, such as those provided by PAPIBrowne et al. (2000).

The results in Figure 13 and Table 1 show that local compensation schemes do manage to reduce the overhead in the worker tasks, but they fail in the master. The parallel compensation scheme reduces the overhead properly in both master and worker tasks.

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 Malony and Shende (2004). In this paper, we model how local overheads affect performance delays across the computation and implement these parallel models in the context of MPI message passing and demonstrate that parallel overhead compensation can be effective in practice to improve measurement error. The engineering feats to accomplish the implementation are novel. In particular, the approach to delay piggybacking can be generalized to other problems where additional information must be sent with messages.

It is important to understand that we are not saying that the performance profile we produce with overhead compensation represents the actual performance profile of an uninstrumented execution. The *performance uncertainty principle* Malony (1991b) 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 being measured during profiling. What we are saying in this paper is that the performance profiles produced with our models for performance overhead compensation will be more accurate than performance results produced without compensation.

7 Acknowledgements

This research is supported by the U.S. Department of Energy, Office of Science contracts DE-FG02-05ER25680, DE-FG03-01ER25501 and DE-FG02-03ER25561.

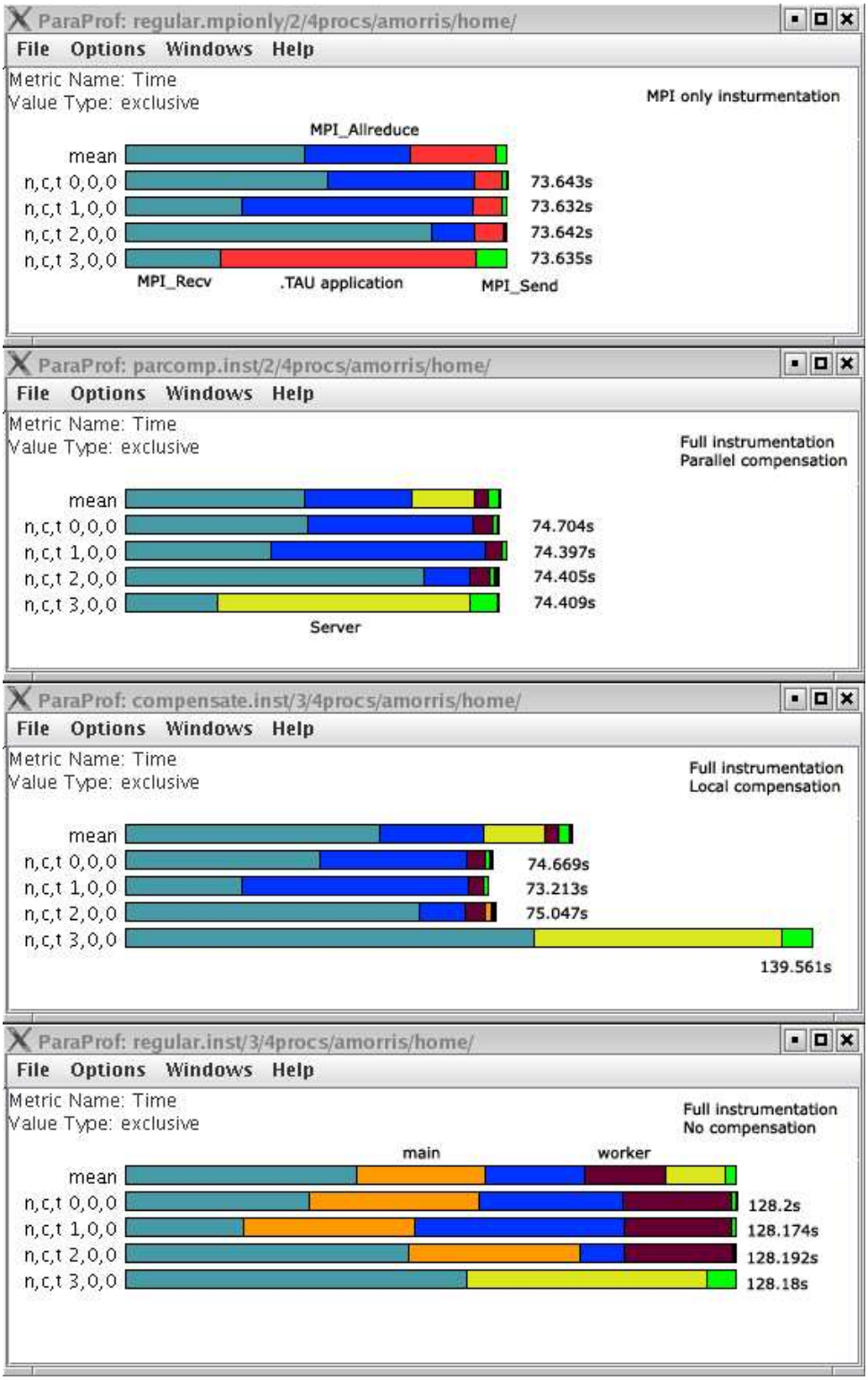


Fig. 13. Parallel overhead compensation in TAU

Bibliography

Bailey, D., Harris, T., Saphir, W., van der Wijngaart, R., Woo, A. and Yarrow, M., 1995. The nas parallel benchmarks 2.0. Technical Report Technical Report NAS-95-020, NASA Ames Research Center.

Bronevetsky, G., Marques, D., Pingali, K. and Stodghill, P., 2003a. Automated application-level checkpointing of mpi programs. In Principles and Practice of Parallel Programming (PPoPP).

Bronevetsky, G., Marques, D., Pingali, K. and Stodghill, P., 2003b. Collective operations in an application-level fault tolerant mpi system. In International Conference on Supercomputing (ICS).

Browne, S., Dongarra, J., Garner, N., Ho, G. and Mucci, P., 2000. A portable programming interface for performance evaluation on modern processors. International Journal of High Performance Computing Applications 14(3):189–204.

Fahringer, T. and Seragiotto, C., 2002. Experience with aksum: A semi-automatic multi-experiment performance analysis tool for parallel and distributed applications. In Workshop on Performance Analysis and Distributed Computing.

Graham, S., Kessler, P. and McKusick, M., 1982. gprof: A call graph execution profiler. In SIGPLAN Symposium on Compiler Construction, pp. 120–126.

IBM. Profiling parallel programs with xprofiler. In IBM Parallel Environment for AIX: Operation and Use, volume Volume 2.

Janssen, C. The visual profiler. URL http://aros.ca.sandia.gov/ cljanss/perf/vprof/.

Knuth, D., 1971. An empirical study of fortran programs. Software Practice and Experience 1:105–133.

Kranzlmüller, D., 2000. Event Graph Analysis for Debugging Massively Parallel Programs. Ph.d., Johanne Kepler Universität Linz.

Laboratories, B., 1979. prof command. In Unix Programmer's Manual, volume Section 1. Murray Hill, NJ.

Malony, A., 1991a. Event based performance perturbation: A case study. In Principles and Practices of Parallel Programming (PPoPP), pp. 201–212.

Malony, A., 1991b. Performance Observability. Ph.d., University of Illinois at Urbana-Champaign.

Malony, A. and Reed, D., 1991. Models for performance perturbation analysis. In ACM/ONR Workshop on Parallel and Distributed Debugging, pp. 1–12.

Malony, A., Reed, D. and Wijshoff, H., 1992. Performance measurement intrusion and perturbation analysis. IEEE Transactions on Parallel and Distributed Systems 3(4):433–450.

Malony, A. and Shende, S., 2004. Overhead compensation in performance profiling. In Euro-Par Conference, volume LNCS 3149, pp. 119–132. Springer.

Malony, A. and Shende, S., 2005. Models for on-the-fly compensation of measurement overhead in parallel performance profiling. In Euro-Par Conference, volume LNCS 3648, pp. 72–82. Springer.

Mellor-Crummey, J., Fowler, R. and Marin, G., 2002. Hpcview: A tool for top-down analysis of node performance. Journal of Supercomputing 23:81–104.

Mucci, P. Dynaprof. URL http://www.cs.utk.edu/ mucci/dynaprof.

Reed, D., Rose, L. D. and Zhang, Y., 1998. Svpablo: A multi-language performance analysis system. In International Conference on Performance Tools, pp. 352–355.

Rose, L. D., 2001. The hardware performance monitor toolkit. In Euro-Par Conference, volume LNCS 2150, pp. 122–131. Springer.

Sarukkai, S. and Malony, A., 1993. Perturbation analysis of high-level instrumentation for spmd programs. In Principles and Practices of Parallel Programming (PPoPP), pp. 44–53.

Shende, S. and Malony, A., 2006. The tau parallel performance system. International Journal of High Performance Computing Applications 20(2):287–331.

Vetter, J., 2002. Dynamic statistical profiling of communication activity in distributed applications. In ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems.

Williams, W., Hoel, T. and Pase, D., 1994. The mpp apprentice performance tool: Delivering the performance of the cray t3d. In Programming Environments for Massively Parallel Distributed Systems. North-Holland.

Task	*No instrumentation*	*No compensation*	*Local compensation*	*Parallel compensation*
Master	73.926	128.179	139.56	73.926
Worker	73.834	128.173	73.212	73.909

Table 1. A comparison of parallel overhead compensation scheme in Monte-carlo integrator