Event-Based Performance Perturbation: A Case Study

Allen D. Malony*

Center for Supercomputing Research and Development
University of Illinois
Urbana, Illinois 61801

Abstract

Determining the performance behavior of parallel computations requires some form of intrusive tracing measurement. The greater the need for detailed performance data, the more intrusion the measurement will cause. Recovering actual execution performance from perturbed performance measurements using event-based perturbation analysis is the topic of this paper. We show that the measurement and subsequent analysis of synchronization operations (particularly, **advance** and **await**) can produce, in practice, accurate approximations to actual performance behavior. We use as testcases three Lawrence Livermore loops that execute as parallel DOACROSS loops on an Alliant FX/80. The results of our experiments suggest that a systematic application of performance perturbation analysis techniques will allow more detailed, accurate instrumentation than traditionally believed possible.

1 Introduction

Many advances in modern science have been achieved through the systematic application of the scientific method. However, experimental scientists have long understood the relationship between the need to observe finer levels of operational behavior (e.g., to test the limits of a theoretical framework) and the technological capabilities of the measurement tools to deliver accurate observations. Clearly, instrumentation and phenomenon must be commensurate to maintain instrumentation perturbations at acceptable levels and to achieve reliable observations.

The problems of uncertainty and instrumentation perturbation in computer system performance analysis are no less profound than in other experimental sciences. The terms "Heisenberg Uncertainty" [15] and "probe effect" [6] have been used to describe the error introduced in the performance measurement due to a monitor's intrusion on computer system behavior. With the exception of passive hardware performance monitors, performance experiments rely on software instrumentation for performance data capture. Such instrumentation mandates a delicate balance between volume and accuracy. Excessive instrumentation perturbs the measured system; limited instrumentation reduces measurement detail — system behavior must be inferred from insufficient data. Simply put, performance instrumentation manifests an *Instrumentation Uncertainty Principle*:

- Instrumentation perturbs the system state.
- Execution phenomena and instrumentation are coupled logically.
- Volume and accuracy are antithetical.

The primary source of instrumentation perturbations is execution of additional instructions. However, ancillary perturbations can result from disabled compiler optimizations and additional operating system overhead. These perturbations manifest themselves in several ways: execution slowdown, changes in memory reference patterns, and even register interlock stalls [19].

From a performance evaluation perspective, instrumentation perturbations must be balanced against the need for detailed performance data. Regrettably, there have been no formal models of performance perturbation that would permit quantitative evaluation given instrumentation costs, measured event frequency, and desired instrumentation detail. Given the lack of models and the potential dangers of excessive instrumentation, detailed performance measurements, mainly in the

*Supported in part by the National Science Foundation under Grants No. NSF MIP-88-07775 and No. NSF ASC 84-04556, and the NASA Ames Research Center Grant No. NCC-2-559.

form of software event traces, often are rejected for fear of corrupting the data (i.e., a small volume of accurate, though incomplete, instrumentation data is preferred). We hypothesize that this restriction is, in many cases, unduly pessimistic.

Our approach to understanding the problem of performance perturbation involves both the creation of perturbation models and the testing of those models through empirical studies [18]. The perturbation models we developed are based on timing and event analysis. Time-based perturbation models attempt to recover accurate timing of trace events from knowledge of instrumentation overhead, assuming event independence. Event-based perturbation models focus on removing the effects of instrumentation on the ordering of events in parallel execution. For both time-based and event-based perturbation analysis, we have conducted a series of instrumentation experiments to determine the validity of the models in an actual execution context [18, 19]. The results of these experiments suggest that a systematic application of performance perturbation analysis techniques will allow more detailed, accurate instrumentation than traditionally believed possible.

This paper reports the results from an empirical study of event-based perturbation analysis as applied to parallel loops with execution dependencies. In particular, we demonstrate the utility of perturbation analysis for three parallel loops from the Lawrence Livermore Loops these three loops execute as DOACROSS loops on an Alliant FX/80. We briefly describe the instrumentation approach we used for our experiments and the goals of performance perturbation analysis in §2. In §3, we present our results from time-based analysis and discuss the basic limitations of this approach for parallel computations with dependent execution. In §4, we consider conservative perturbation analysis approaches for programs with DOACROSS loops using advance/await synchronization. The results from the Livermore loop experiments using event-based perturbation analysis are presented in §5. In §6, we give concluding remarks.

2 Instrumentation and Perturbation Analysis

Models to capture and remove perturbations due to instrumentation must be based on a particular instrumentation approach. Because tracing is the most general form of instrumentation, allowing both static and dynamic analysis, we derive perturbation models for trace instrumentation. Given an understanding of possible performance instrumentation perturbations and measures of *in vitro* trace instrumentation costs in an execution environment, our goal for perturbation analysis is to recover the "true" trace of events from an measured trace as they would have been generated during an execution without instrumentation. There are two phases in this perturbation analysis:

- **Execution Timing Analysis** – Given the measured costs of instrumentation, adjust the trace event times to remove these perturbations.
- **Event Trace Analysis** – Given instrumentation perturbations that can reorder trace events, adjust the event sequence based on knowledge of event dependencies, maintaining causality.

In both phases, models are needed that describe observed behavior as a perturbation of true behavior. For timing analysis, one must approximate true times of event occurrence, either for each trace event or for the total execution time. That is, the timing model must describe how the perturbations affect measured execution times. Event analysis models are more difficult; program or system semantic information is needed to determine if the relative event order is incorrect and, if so, generate a better approximation to the true order. Before discussing the timing and event models, we begin with a formal description of our instrumentation approach.

Given a program P composed of a sequence of statements $S_1, S_2, \ldots, S_n$ and a set of instrumentation points $I_1, I_2, \ldots, I_n$, an instrumentation of P is defined as

$$\mathcal{I}(P) = I_1, S_1, I_2, S_2, \ldots, I_n, S_n \ ,$$

where some I_j may be null (i.e., no instrumentation). Thus, we define instrumentation on a statement basis, where an event represents the execution of a statement.

A *logical event trace*, τ, is a time-ordered sequence of events $e_1, \ldots, e_m$ where each e_i is of the form $\{t(e_i), eid_i\}$, eid_i is the event identifier for the i^{th} event representing the statement S_{eid_i} in the program, and $t(e_i)$ is the time when the event occurred. The logical event trace represents the program's actual performance. If the program is instrumented, we use the notation τ_m to denote a *measured event trace.* The measured event trace represents the program's measured performance. However, τ_m also reflects a perturbation of τ in execution time and, possibly, event order. Perturbation models attempt to use execution information contained in τ_m to resolve the instrumentation perturbations that occurred during the measurement and to approximate actual performance behavior.

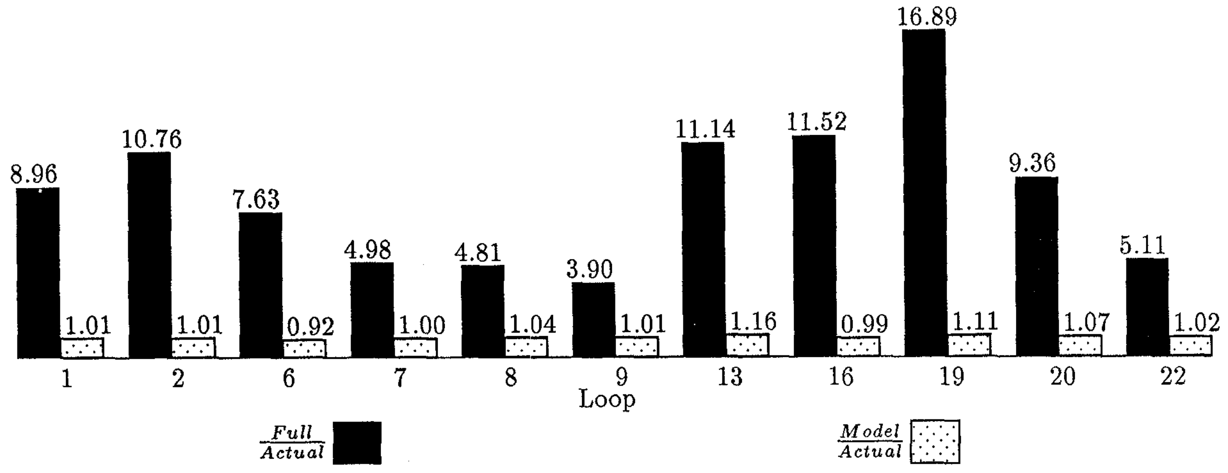

Figure 1: Sequential Loop Execution: Measured and Approximated Ratios

3 Time-Based Analysis

In previous work [19], we developed time-based perturbation models that permit the removal of perturbations from application program traces. The models assumed independence among threads of execution and, therefore, accounted only for the execution time overhead of the instrumentation when constructing performance approximations from the measured traces. We applied the models in the performance analysis of scalar, vector, and concurrent executions of the Lawrence Livermore Loops [20] on the Alliant FX/80 multiprocessor [24]. To test the robustness of the models, a full, statement-level instrumentation of the loops was performed — every source statement execution was captured by an event in an execution trace.

Surprisingly, the relatively simple models were able to approximate many of the Livermore loop execution times to within fifteen percent from full trace instrumentations, where measured execution time perturbations exceeded four orders of magnitude. As an example, Figure 1 shows the perturbation analysis results for sequential loop execution of some of the Livermore loops. In the figure, the black bars represent the ratio of measured loop execution time (with full instrumentation) to the actual loop execution time. The dotted bars represent the ratio of the approximated execution time (using the time-based models) to the actual loop execution time. Even though slowdowns exceeded sixteen times, as in the case of loop 19, approximated times were extremely accurate, relative to the measured error. Not only did the models perform well when approximating total execution time, but the accuracy of individual event timings were equally impressive.

The time-based perturbation models accurately capture the effects of instrumentation perturbation when the time and order events occur is execution independent. This is true for sequential execution because the execution states of sequential programs form a total order, and event times are affected only by instrumentation overhead. As a result, our timing model approximations for the Livermore loops in sequential and vector modes were extremely accurate. Even for some concurrent execution scenarios, typically those with simple fork-join behavior and no inter-thread dependencies, the time-based perturbation models were good.

However, in general, concurrent execution involves data dependent behavior. The states of parallel programs inherently form a partial order that must be followed during execution. If dependency control is spread across threads of execution, instrumentation can perturb the timing relationships of events. Direct applications of time-based perturbation models will fail because they do not capture these inter-thread event dependencies. This was the case for loops 3 (Inner Product), 4 (Banded Linear Equations), and 17 (Implicit, Conditional Computation). From our full instrumentation experiments, Table 1 gives the ratio of the measured and approximated execution times to the actual execution times using time-based perturbation analysis for the three loops.

For Livermore loops 3 and 4, our execution time model over-estimates the instrumentation perturbation (i.e., the model's estimate of total concurrent execution time is too low). Both loops contain a small critical section for the update of a shared variable that synchronizes the loop iterations. Without instrumentation, most processors are blocked on entry to this critical section. Adding instrumentation increases the total computation in each concurrent iteration and reduces the probability of blocking at the critical section. In consequence, the processors spend less time waiting when

Livermore Loop	Execution Ratio	
	$\frac{Measured}{Actual}$	$\frac{Approximated}{Actual}$
3	2.48	0.37
4	2.64	0.57
17	9.97	8.31

Table 1: Loop Execution Time Ratios: Time-Based Analysis

the code is instrumented. Removing only the instrumentation overhead without considering the effects on blocking probability results in an under-approximation of the total execution time.

For Livermore loop 17, our model under-estimates the instrumentation perturbation. The reason is the same as for loops 3 and 4 — the loop contains a critical section. However, the critical section is large and includes tracing code when instrumented. The additional instrumentation code increases the probability of contention, and the critical section becomes a larger fraction of the total execution time in the instrumented code. The additional waiting time associated with increased blocking is not considered by the time-based perturbation analysis, resulting in an over-approximation of total execution time.

4 Event-Based Analysis

The performance of parallel computations often depends on the relative ordering of dependent events across multiple threads of execution. Each parallel thread can be viewed as making transitions between phases of independent and dependent execution. That is, either the thread can proceed with a computation independent of the activities in the other threads, or the thread is dependent on some conditions that must be satisfied before proceeding. In cases where a thread is waiting for a synchronization action, overall performance is reduced. However, parallel performance is also dependent on the efficient allocation and utilization of resources, mainly processor, memory, and network resources. Scheduling, load balancing, data partitioning, and communication overhead are among the many performance issues that must be addressed.

4.1 Likely Executions and Conservative Approximations

The fundamental problem with making detailed measurements of parallel computations is not performance degradation, as it is with sequential computations, but rather the perturbation of the set of "likely" event orderings, resulting in the re-mapping of event occurrence to threads of execution, the reassignment of computational resources [16], and changes in the behavior of resource use. Unlike parallel debugging approaches that attempt to detect data races in parallel programs by applying an event-based, partial order theory of "feasible" program execution [5, 23, 25], perturbation analysis must recover the actual run-time performance behavior from a perturbed performance measurement.

If performance instrumentation is designed correctly, an un-instrumented parallel execution that satisfies Lamport's *sequential consistency* criterion[1] [13] implies that the performance measurement will be *non-interfering* and *safe* [10]. If the performance measurements involve only the detection and recording of event occurrence (i.e. tracing), the partial order relationships will be unaffected and the set of feasible executions[2] will remain unchanged [21, 23]. This consequence will also be true in the less restricted condition of *weak ordering* [2]. Thus, perturbation analysis begins with a total ordering of measured events consistent with the *happened before* relation [12] defined by the original partial order execution. To this total order, we can apply time-based perturbation analysis to thread events that occurred during independent execution to remove the instrumentation overhead. Similarly, event-based perturbation analysis [18] (see below) can be applied to the synchronization operations (e.g., barriers, semaphores, advance/await) that implement the dependency relationships. As long as the total ordering of dependent events present in the measured execution is maintained during the analysis, the approximated execution also will be a feasible execution. We will call such an approximated execution a *conservative approximation.*

However, the important question is not whether the conservative approximation is a feasible execution, but whether it is a likely execution. The set of likely executions is the subset of the feasible executions that are most probable. Computing the likelihood distribution of feasible executions is an extremely difficult problem, requiring a model of time and concurrent execution. Analytical queueing models have been used to predict the performance of parallel computations [8, 9], including general, parallel execution structures with precedence constraints [17, 22, 27, 28] and synchronization [1]. However, these approaches typically model exe-

[1] A parallel execution is sequentially consistent if the result is the same as if the operations were executed in some sequential order obtained by arbitrarily interleaving the thread execution streams.

[2] Helmbold and Bryan refer to the set of feasible executions defined by the partial order of program events as the *partially ordered set* [10].

cution time behavior stochastically, limiting their use in practice. Other approaches attempt to model time dependent behavior in concurrent software. Lane [14] proposes the *event dependency tree* model that includes time semantics but lacks a broad set of synchronization operations to make it a practical approach. Haase [7] and Shaw [26], on the other hand, each define a *time logic* based on program statements for reasoning about timing properties in programs. Shaw's work is more robust, considers a larger class of program constructs and uses interval arithmetic in the logic specification, but it only superficially treats timing issues in dependent concurrent execution. Shaw comments that an approach to concurrent timing analysis must consider the specific context within which synchronization statements are used, including their overheads and interactions.

The inability to predict likely executions makes it difficult to bound the error of conservative approximations. Furthermore, no intrusive performance measurements can possibly allow event-based perturbation analysis to determine the proper assignment and use of resources in the approximated execution. To improve the "accuracy" of the conservative approximation, additional information must be provided to the perturbation analysis process that describes certain behavioral properties of the computation (e.g., data dependency information and loop scheduling algorithms). The perturbation analysis can use this information to make more "liberal" approximations. Although the liberal approximations might be more accurate than conservative ones, in the sense that they are closer to likely executions, it is still difficult to show error bounds without a more formal timing model.

4.2 Advance/Await Synchronization

In [18], we give an extensive discussion of methods for conservative perturbation analysis of synchronization operations found in concurrent programs. Because we are interested in testing event-based analysis in a real execution context, we limit the discussion here to advance/await synchronization as found in DOACROSS loops — the Livermore loops with data dependencies execute in concurrent mode on the Alliant FX/80 as DOACROSS loops using the advance/await synchronization hardware of that machine.

4.2.1 Operation

The advance/await form of synchronization is a special case of the general semaphore. Each **await** operation synchronizes with a unique **advance** operation. Each **advance/await** operation pair can be thought of as operating on a unique semaphore. A general advance/await synchronization variable, A, stores the history of **advance** operations. The semantics of the **advance** and **await** operations are shown below:[3]

advance(A, i):	mark in A that i was advanced
await(A, i):	if (i has not been advanced in A) wait until i has been advanced

4.2.2 Instrumentation

To correctly identify which **advance** and **await** operations should be paired during perturbation analysis, *advance* and *await* events must be recorded with a unique value identifying the pair. Typically, advance/await synchronization is used to control the execution of loops with iteration dependencies [3]. The unique identifier in this case might be the loop iteration index. In general, the instrumentation for capturing the *advance* and *await* events must generate the unique identifier itself. From the semantics of the **advance** and **await** primitives above, this unique identifier could be the argument i.

Actually, two *await* events are recorded: one to identify the beginning of the **await** operation, $await_B$, and another to identify the end of the **await** operation, $await_E$, after the **advance** operation has occurred. If the **advance** operation occurs before the corresponding **await** operation, the $await_E$ event will be recorded immediately after $await_B$, separated in time only by the instrumentation overhead and the **await** code processing. The *advance* event is recorded after the **advance** operation completes.

4.2.3 Perturbation Analysis

Because of the strict synchronization enforced by the **advance** and **await** operations, a partial ordering of these actions can be determined directly from the measured *advance* and *await* events. We can easily identify the advance/await pairing from the instrumentation measurement and, hence, the ordering relationship of pairs of these events across the threads of execution. Independent of how **advance** and **await** operations are assigned to threads, this partial ordering must be maintained in the approximated execution.

Perturbation analysis is a constructive process. For each thread of execution, it attempts to resolve the approximate time of occurrence, t_a, for each successive event. The approximate time of an event x, $t_a(x)$, cannot be determined until the approximate time of all other events upon which x depends have been resolved;

[3]Our definition of the advance and await operations is more general than what is often described. However, the perturbation analysis still applies in the more restrictive case.

x is always *execution dependent* [29] on the last preceding event occurring on the same thread.[4] Of these events, one will serve as a *time basis* when calculating $t_a(x)$.

When approximating the time of *advance* events, we only need to know the approximate time of the preceding event on the same thread. If we let u represent the preceding event, then

$$t_a(advance) = t_a(u) + t_m(advance) - t_m(u) - \alpha,$$

where $t_m(advance)$ represents the measured time of the *advance* event, $t_m(u)$ the measured time of u, and α the overhead of the **advance** operation instrumentation. Similarly, the approximate time of the event preceding an $await_B$ event, v, serves as the time basis for calculating $t_a(await_B)$,

$$t_a(await_B) = t_a(v) + t_m(await_B) - t_m(v) - \beta,$$

where β is the instrumentation overhead at the beginning **await** operation.

However, approximating the time of an $await_E$ event requires knowing $t_a(advance)$ and $t_a(await_B)$. If $t_a(advance) \leq t_a(await_B)$, we assume no waiting occurs in the approximate execution and

$$t_a(await_E) = t_a(await_B) + s_{nowait},$$

where s_{nowait} represents the synchronization overhead of the **await** operation when no waiting occurs. If $t_a(advance) > t_a(await_B)$, waiting will occur in the approximated execution, and $t_a(advance)$ will serve as the time basis for approximating the time of occurrence of $await_E$. That is,

$$t_a(await_E) = t_a(advance) + s_{wait},$$

where s_{wait} represents the synchronization overhead of the **await** operation when waiting results, after the **advance** operation occurs. The overheads s_{nowait} and s_{wait} are empirically determined and are input to the perturbation analysis.

By applying the above formulae, we can resolve all time approximations for *advance* and *await* events appearing in the measured trace. However, because instrumentation intrusion can cause a perturbation of the relative ordering of **advance** and **await** operation from the actual to the measured execution, this relative ordering could also be different in the approximation. This can cause synchronization waiting, which occurred in the measured execution due to instrumentation intrusion, to be removed during perturbation analysis. It is also possible that synchronization waiting manifests itself in the approximation when it did not occur in the measurement. Figure 2 shows these two cases, with the *advance*, $await_B$, and $await_E$ events marked. Note that the instrumentation overheads have been removed in the approximation.

One interesting aspect of advance/await synchronization is that the pair of execution threads synchronizing can change dynamically. Although conservative, event-based perturbation analysis can keep the partial order consistent with a feasible execution, the concurrent work constrained by the **advance** and **await** operations might be scheduled differently in the actual execution than what is observed from the measured events — a condition that conservative analysis cannot detect or resolve. The perturbation analysis does not know *a priori* that work reassignment to threads is allowed. The use of external execution information to reassign the work bounded by *advance* and *await* events during perturbation analysis can lead to significant differences in approximated execution behavior [18].

[4] If an event y occurs before an event x on the same thread of execution, then x is execution dependent on y, but only in the context of the particular execution.

4.3 The DOACROSS Loop Model

A DOACROSS loop [3] contains data dependencies [11] between iterations that that must be enforced to maintain correct execution order. The notion of a *data dependence distance* [29], d, is used to quantify the dependencies within an iteration of a DOACROSS loop. If iteration $i + d$ is dependent on iteration i, the distance of the data dependence is d. For our purposes, we will focus on *constant-distance* dependencies [29] (i.e., dependencies with constant data dependence distances).

Unlike DOALL loops [4], the execution of DOACROSS loops is not entirely nondeterministic— the execution is constrained by the synchronization points within each iteration and by their partially ordered execution. Although errors can occur in conservative event-based perturbation analysis because of lack of knowledge about correct resource assignment, requiring some form of scheduling simulation to improve approximation accuracy, the dependent execution can restrict the set of possible run-time behaviors.

5 Event-Based Perturbation Analysis of the Livermore Loops

As discussed earlier, not all of the Livermore loops were amenable to timing-based perturbation analysis of concurrent execution. Loops 3, 4, and 17, in particular, show significant errors in the timing model approximations on the Alliant FX/80. All three loops execute

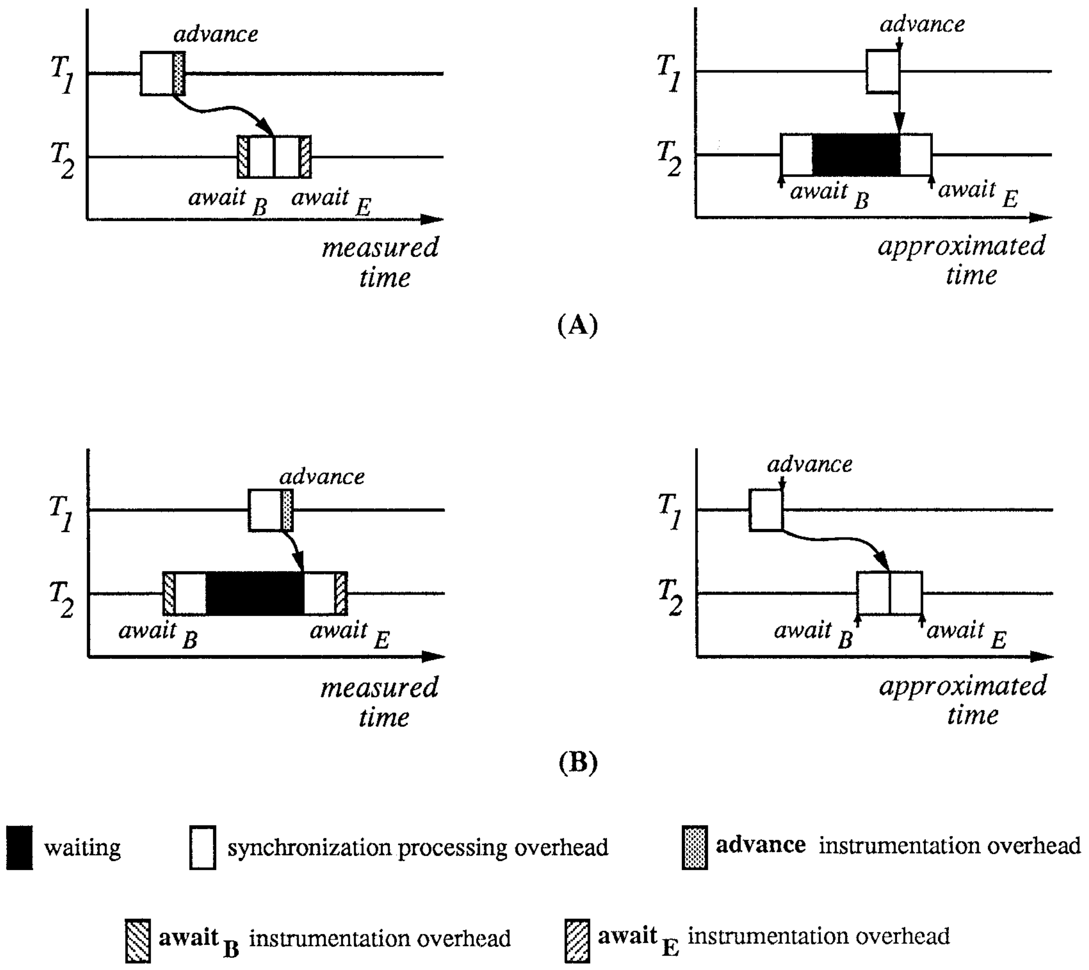
advance
T_1
T_2
$await_B$
$await_E$
measured time
approximated time
(A)
(B)
waiting
synchronization processing overhead
advance instrumentation overhead
$await_B$ instrumentation overhead
$await_E$ instrumentation overhead

Figure 2: Advance/Await Synchronization: Measurement and Approximation

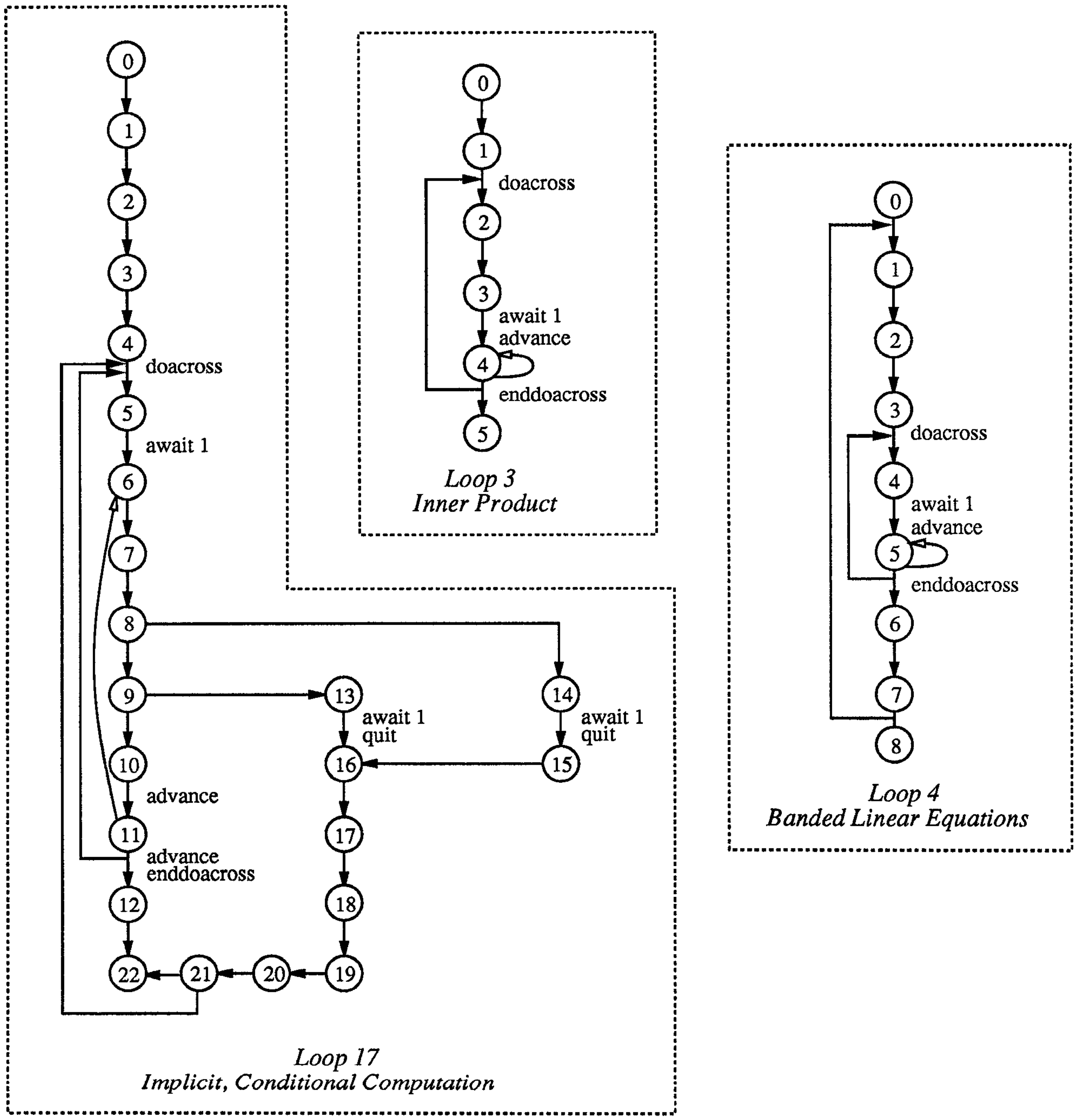

Figure 3: Lawrence Livermore Loops 3, 4, and 17

as DOACROSS loops with advance/await synchronization to prevent concurrent access to a critical section. The concurrent execution behavior of the loops violates the assumptions of the time-based perturbation analysis; namely, the assumptions of event independence and simple fork-join behavior of parallel computations. An event-based approach must be applied instead. To evaluate the effectiveness of our event-based perturbation analysis, in particular the models for advance/await synchronization in DOACROSS loops, we applied an event trace analysis tool to these three Livermore loops.

5.1 Loop Synchronization Structure

Figure 3 shows the structure of the three loops, including the placement of DOACROSS loop begin and end, and the **advance** and **await** synchronization operations. As before, every statement in the loop was instrumented — each node in the graph corresponds to a statement in the loop and, hence, an event in the trace. In addition, the synchronization operations used to satisfy statement dependencies were also instrumented.[5] The statement dependencies are shown by the white arrows. The events following the **advance** and **await** operations are identified as special synchronization events by the analysis tool so that the advance/await semantics can be correctly enforced.[6] The end of the DOACROSS loops are handled as barriers.[7]

5.2 Loop Approximations

Table 2 gives the ratio of measured and approximated execution time to the actual execution time of the loops. The $\frac{Measured}{Actual}$ ratio compares the measured execution time from a full instrumentation for event analysis to the actual time. When compared to the ratios in Table 1, we see a slowdown in measured execution time for event analysis because of the additional synchronization instrumentation overhead.

The $\frac{Approximated}{Actual}$ ratio shows the accuracy of the approximation from event-based perturbation analysis. Applying event-based perturbation analysis to ensure the partial ordering of **advance** and **await** operations in the approximated execution significantly improves the accuracy of the execution time estimations. For loops 3 and 4, the timing model approximated a smaller execution time (see Table 1) than actual because the analysis did not treat the *advance* and *await* events as special, and waiting times that should result because of the critical section in the actual execution are not maintained in the approximation. In the case of loop 3, the actual execution time is 2.7 times that of the timing-based approximation (a -63 percent error). However, event perturbation modeling improves the approximation to be a factor of 0.96 of the actual time (a -4 percent error). The improvements in the loop 4 approximation are similar. The event model achieves a 6 percent error in this case.

Livermore Loop	Execution Ratio	
	$\frac{Measured}{Actual}$	$\frac{Approximated}{Actual}$
3	4.56	0.96
4	3.38	1.06
17	14.08	0.97

Table 2: Loop Execution Time Ratios: Event-Based Analysis

In contrast, the timing-based model approximates an execution time slower than actual for loop 17. Here the instrumentation in the critical section of the loop results in greater waiting during measured execution which cannot be removed by the timing analysis. However, even with a 14 times slowdown in the measured execution, event-based perturbation analysis accurately resolves the timing of *advance* and *await* synchronization events to produce an approximation with only a -3 percent error. The advantage over the timing-based model is apparent — a factor of over 8 in improved accuracy is achieved.

From the results above, there appears to be a violation of the Instrumentation Uncertainty Principle we introduced in §1; namely, performance data volume and accuracy are antithetical. It was necessary to instrument loops 3, 4, and 17 more heavily in order to capture synchronization execution. From the $\frac{Measured}{Actual}$ ratios, this additional instrumentation shows up clearly as increased execution time overhead. However, the additional instrumentation is providing information about the occurrence of synchronization operations during the computation. Although the instrumentation overhead further perturbs the execution time measurements, this additional knowledge of synchronization operation allows more accurate perturbation analysis.

[5] It is interesting to note that these synchronization operations were not a part of the original source and, therefore, could not be instrumented at the source level. Instead, these operations were added by the Alliant parallelizing Fortran compiler, requiring the instrumentation to be made to the assembly code produce by the compilation.

[6] As discussed in 4.2.2, the instrumentation for event perturbation analysis stores additional information with synchronization events to correctly determine their relationship. In the case of the loops here, we store the iteration number with every event.

[7] For this barrier synchronization, we applied barrier perturbation models [18] during the analysis.

Processor							
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
4.05 %	8.09 %	4.05 %	2.70 %	4.05 %	5.40 %	2.70 %	4.05 %

Table 3: DOACROSS Waiting Time in Loop 17

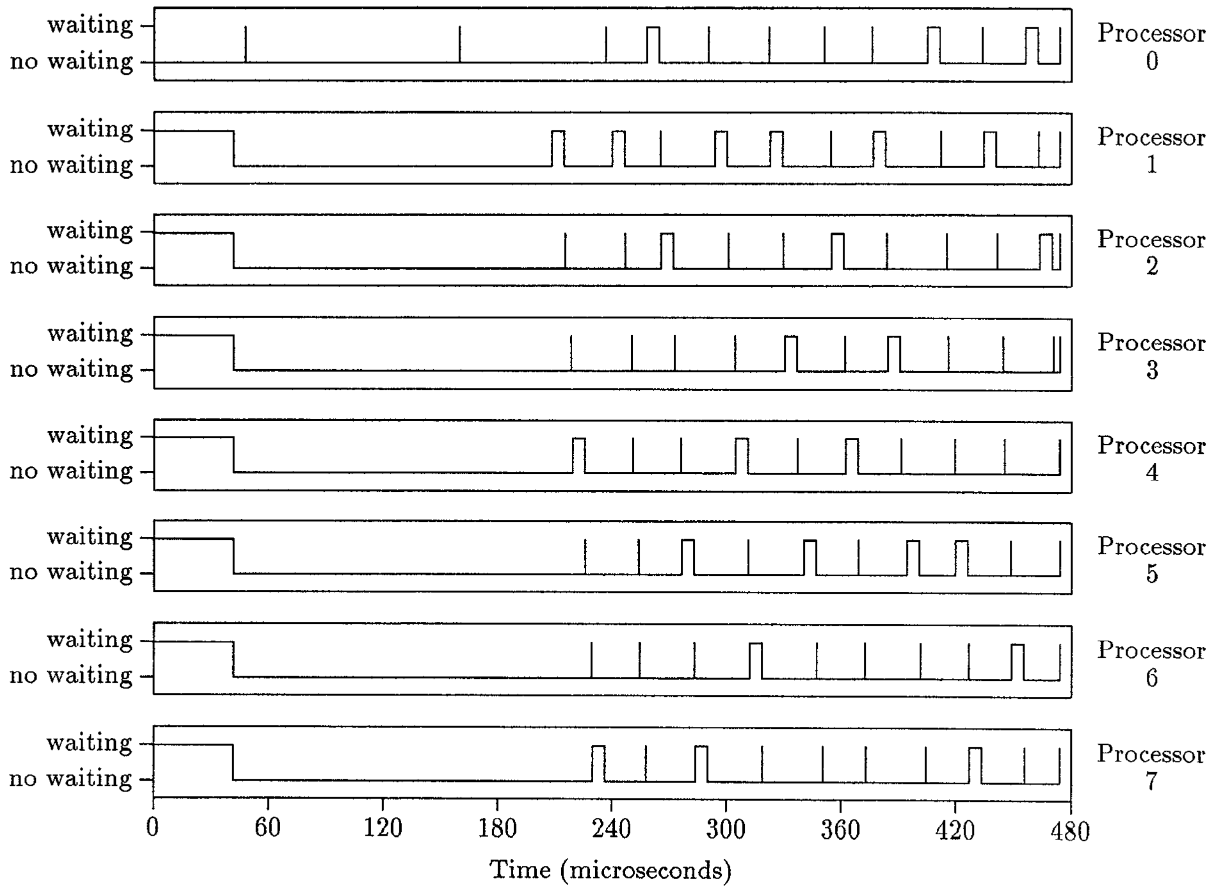

Figure 4: Approximated Waiting Behavior in Livermore Loop 17

5.3 Loop Performance Analysis

In addition to producing total execution time approximations, event-based analysis can also generate statistics about loop execution such as the amount of waiting on each processor and the degree of parallelism across processors. As an example, we computed the percentage of total execution time spent waiting on each processor in loop 17. These results are shown in Table 3. A execution time history graph of waiting time for each processor is shown in Figure 4. (The sequential portions before and after the parallel DOACROSS loop are shown as processor zero active.)

From the waiting information, we computed the level of parallelism in the computation. The average level of parallelism of loop 17, excluding the sequential portions, is 7.5. More insight into parallelism behavior can be gained from a graph of parallelism over time. This graph is shown in Figure 5. All the waiting and parallelism curves presented were generated from the execution approximations of the event-based perturbation model.

6 Conclusion

Achieving the accuracy in the event-based approximations of Livermore loops 3, 4, and 17 from full instrumentations is astonishing. It is even more surprising in light of the extra instrumentation (and hence intrusion) needed to collect synchronization operation data for perturbation analysis; a perceived violation of the instrumentation principle. Although we cannot expect this level of accuracy in all cases, it does suggest that a systematic application of these techniques will increase the confidence of making detailed performance measurements.

In general, event-based perturbation models must be based on a better understanding of the effects of intrusion in the context of nondeterministic execution. In many cases, the complete range of *feasible* executions will be restricted to a smaller set of *likely* executions due to the computational environment. If instrumentation is added, the set of likely executions can change. Without a formulation of nondeterministic execution in the presence of instrumentation, we must rely on empirical evidence, as demonstrated above, that event-based perturbation analysis is a viable technique.

References

[1] M. Abrams and A. Agrawala. Exact Performance Analysis of Two Distributed Processes with Multiple Synchronization Points. Technical Report TR-1845, University of Maryland, Department of Computer Science, February 1987.

[2] S. Adve and M. Hill. Weak Ordering - A New Definition and Some Implications. In *Proceedings of the Seventeenth International Symposium on Computer Architecture*, pages 2–14, Seattle, Washington, May 1990.

[3] R. Cytron. *Compile-Time Scheduling and Optimization for Asynchronous Machines.* PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, October 1984.

[4] J. Davies. Parallel Loop Constructs for Multiprocessors. Master's thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, May 1981.

[5] C. Fidge. Partial Orders for Parallel Debugging. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pages 183–194. ACM Sigplan/Sigops, May 1988. University of Wisconsin.

[6] J. Gait. A Probe Effect in Concurrent Programs. *Software Practice and Experience*, 16(3), March 1986.

[7] V. Hasse. Real-Time Behavior of Programs. *IEEE Transactions on Software Engineering*, 7(9):454–501, September 1981.

[8] P. Heidelberger and K. Trivedi. Queueing Network Models for Parallel Processing with Asynchronous Tasks. *IEEE Tranactions on Computers*, 31(11):1099–1109, November 1982.

[9] P. Heidelberger and K. Trivedi. Analytic Queueing Models for Programs with Internal Concurrency. *IEEE Tranactions on Computers*, 32(1):73–82, January 1983.

[10] D. Helmbold and D. Bryan. Design of Run Time Monitors for Concurrent Programs. Technical Report CSL-TR-89-395, Stanford University, October 1989.

[11] D. Kuck. *The Structure of Computers and Computations.* Wiley, New York, 1978.

[12] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[13] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Tranactions on Computers*, 28(9):690–691, September 1979.

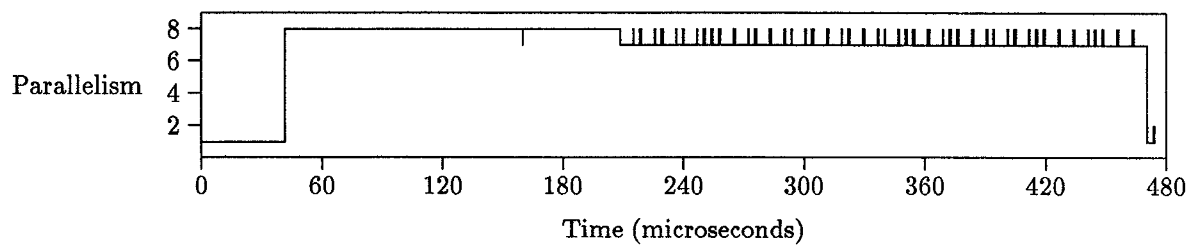

Figure 5: Approximated Parallelism Behavior in Livermore Loop 17

[14] D. Lane. A Model of Time Dependent Behavior in Concurrent Software Systems. Technical Report TR-87-28, University of California, Irvine, Department of Information and Computer Science, November 1987.

[15] C. LeDoux and D. Parker. Saving Traces for Ada Debugging. In *Proceedings of the SIGAda International Ada Conference*, pages 1–12, 1985.

[16] T. Lehr. *Comensating for Perturbation by Software Performance Monitors in Asynchronous Computations*. PhD thesis, Carnegie-Mellon University, Department of Computer Science, April 1990.

[17] V. Mak. Performance Prediction of Concurrent Systems. Technical Report CSL-TR-87-344, Stanford University, December 1987.

[18] A. Malony. *Performance Observability*. PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, September 1990.

[19] A. Malony, D. Reed, and H. Wijshoff. Performance Measurement Intrusion and Perturbation Analysis. Technical Report CSRD-923, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, October 1989.

[20] F. McMahon. The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, December 1986.

[21] J. Mellor-Crummey. *Debugging and Analysis of Large-Scale Parallel Programs*. PhD thesis, University of Rochester, Department of Computer Science, September 1989.

[22] J. Mohan. *Performance of Parallel Programs*. PhD thesis, Carnegie-Mellon University, Department of Computer Science, July 1984.

[23] R. Netzer and B. Miller. On the Complexity of Event Ordering for Shared Memory Parallel Program Executions. Technical Report CS-908, University of Wisconsin at Madison, Department of Computer Science, January 1990.

[24] R. Perron and C. Mundie. The Architecture of the Alliant FX/8 Computer. In *Spring COMPCON '86*, pages 390–393, March 1986.

[25] V. Pratt. Modeling Concurrency with Partial Orders. *International Journal of Parallel Programming*, 15(1), 1986.

[26] A. Shaw. Reasoning About Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, 15(7):875–889, July 1989.

[27] A. Thomasian and P. Bay. Queueing Network Models for Parallel Processing of Task Systems. In *Proceedings of the 1983 International Conference on Parallel Processing*, pages 421–428, August 1983.

[28] A. Thomasian and P. Bay. Analytic Queueing Network Models for Parallel Processing of Task Systems. *IEEE Tranactions on Computers*, 35(12):1045–1054, December 1986.

[29] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, 1982.