Models for Performance Perturbation Analysis

Allen D. Malony[†]

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

Daniel A. Reed[‡]

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

Abstract

When performance measurements are made of program operation actual execution behavior can be perturbed. In general, the degree of perturbation depends on the intrusiveness and frequency of the instrumentation. If the perturbation effects of the instrumentation cannot be quantified by a perturbation model (and subsequently removed during perturbation analysis), detailed performance measurements could be inaccurate. Developing models of time and event perturbations that can recover actual execution performance from perturbed performance measurements is the topic of this paper. Time-based models can accurately capture execution time perturbations for sequential computations and concurrent computations with simple fork-join behavior. However, the performance of parallel computations generally depends on the relative ordering of dependent events and the assignment of computational resources. Event-based models must be used to quantify instrumentation perturbation in parallel performance measurements. The measurement and subsequent analysis of synchronization operations (e.g., barrier, semaphore, and advance/await synchronization) can produce accurate approximations to actual performance behavior. Unfortunately, event-based models are limited in their ability to fully capture perturbation effects in nondeterministic executions.

1 Introduction

As the complexity of computer systems increase, there is a greater need for performance data to understand the interactions between different performance factors. However, detailed performance measurement is not without cost. In particular, the overheads associated with the execution of instrumentation points in a program can perturb its actual operational behavior and, consequently, can introduce errors in the observed performance data. For sequential programs, the primary perturbations are in execution time. For parallel computations, instrumentation can perturb event ordering and resource usage, in addition to execution time.

From a performance evaluation perspective, instrumentation perturbations must be balanced against the need for detailed performance data. Excessive instrumentation perturbs the measured system; limited instrumentation reduces measurement detail — system behavior must be inferred from insufficient 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 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).

This paper describes several performance perturbation models that can be used to recover actual performance behavior from perturbed performance measurements. The perturbation models we developed are based on time and event analysis [9]. Time-based perturbation models attempt to recover accurate timing of trace events from knowledge of instrumentation overhead, assuming event independence. Event-based per-

[†] 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. Current at the Department of Computer and Information Science, University of Oregon, Eugene, Oregon 97403.

[‡] Supported in part by the National Science Foundation under grants NSF CCR-86-57696, NSF CCR-87-06653 and NSF CDA-87-22836 and by the National Aeronautics and Space Administration under NASA Contract Number NAG-1-613.

turbation models focus on removing the effects of instrumentation on the ordering of events in parallel execution. Although more robust than time-based models, event-based models require more intrusive measurements to capture data regarding the execution of synchronization operations. All the models we discuss in the paper have been implemented in perturbation analysis software and tested against a series of instrumentation experiments to determine their validity [9, 11, 10]. 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.

In §2, we describe our instrumentation approach and define metrics to quantify instrumentation perturbations. In §3, time-based perturbation models are discussed for both total execution time and per event time approximations. Because time-based models assume event independence during parallel execution, they will fail when there are execution ordering dependencies. Models for event-based perturbation analysis, described in §4, use measurements of synchronization operations to resolve perturbations in these cases. However, perturbation models based on performance measurements alone cannot quantify all perturbation errors. Ultimately, the analysis must include information about the execution context. In conclusion, we discuss the limits of measurement-based perturbation models and the directions for future work.

2 Instrumentation Approach

Performance perturbation models 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. Before discussing the time-based and event-based perturbation models, we begin with a formal description of our instrumentation approach.

2.1 Event Traces

A computer system's operation can be regarded as a sequence of *actions* representing some significant physical or logical activity performed. For measurements purposes, the execution of an action generates an *event* — an encoded instance of an action. Events are dynamic and each instance of an action defines a separate event. A performance measurement can be viewed as the collection of a (possibly infinite) set of events. Each event in the measurement indicates the action type (or identity), the time when the action occurred, information about where the action occurred, and any additional data that further defines the action state.

For our purposes, the actions to be observed are the execution of program statements and we will use instrumentation to capture these actions in an event trace. 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). Each instrumentation point captures an event identifying the execution of the corresponding statement.

We define a *logical event trace*, τ, to be 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. It is the logical event trace that we are trying to capture in the performance measurement. We use the notation τ_m to denote a *measured event trace*. The measured event trace represents the program's measured performance. Because a program can have both sequential and concurrent components, we define the sequential event trace, τ^s (τ^s_m), as the sub-sequence of events $e_p, e_q, \ldots, e_r$ generated in sequential mode. Similarly, the concurrent event trace for processor i, τ^i (τ^i_m), is the sub-sequence of events $e^i_p, e^i_q, \ldots, e^i_r$ executed in concurrent mode on processor i.

2.2 Definitions and Metrics

The measured trace, τ_m, reflects a perturbation of τ in execution time and, possibly, event order. The following definitions are used to quantify the perturbations of trace instrumentation.

The total execution time of a sequential program P is

$$T^s(P) = \sum_{e_i \in \tau^s} T(S_{eid_i}) \ ,$$

where $T(S_{eid_i})$ is the actual execution time of statement S_{eid_i}. The measured program execution time of an instrumentation of P is

$$T^s_m(\mathcal{I}(P)) = \sum_{e_i \in \tau^s_m} [T(S_{eid_i}) \oplus T(I_{eid_i})] \ ,$$

where $T(I_{eid_i})$ is the direct execution time overhead of the instrumentation point I_{eid_i}. The coupling of execution times for program statements and instrumentation, represented by $\oplus$, denotes perturbations not included in individual instrumentation and statement timings.

For concurrent execution time calculations, one must determine the critical path during concurrent computation. Let $\tau^s = e_p, e_q, \ldots, e_r$ represent the logical trace of sequential events and $\tau^{cp} = e_s, e_t, \ldots, e_u$ the logical trace of concurrent events along the critical path,

respectively. The total actual execution time of a concurrent program P is

$$T^c(P) = \sum_{e_i \in \tau^s} T(S_{eid_i}) + \sum_{e_j \in \tau^{cp}} T(S_{eid_j}) \ .$$

Similarly, the measured program execution time of an instrumentation of P is:

$$\begin{aligned} T_m^c(\mathcal{I}(P)) &= \sum_{e_i \in \tau_m^s} [T(S_{eid_i}) \oplus T(I_{eid_i})] \\ &+ \sum_{e_j \in \tau_m^{cp}} [T(S_{eid_j}) \oplus T(I_{eid_j})] \ . \end{aligned}$$

where $\tau_m^{cp} = e_x, e_y, \ldots, e_z$ represents the critical path of concurrent events in the instrumented program. Unfortunately, the concurrent event sequence identified as the critical path in $T^c(P)$, τ^{cp}, may differ from that for $T_m^c(\mathcal{I}(P))$, τ_m^{cp}.

From the above definitions, one can define a series of instrumentation perturbation metrics. We will use $T(P)$ and $T_m(\mathcal{I}(P))$ to represent actual and measured execution times for both sequential and concurrent timing measurements. The simplest metric, absolute error in measured execution time, is defined in the standard way. The absolute error, AE, is

$$AE = T_m(\mathcal{I}(P)) - T(P) \ . \qquad (1)$$

The perturbation in execution time caused directly by the execution of instrumentation instructions is called *direct perturbation*. The direct perturbation, DP, is defined as

$$DP = DP^s + DP^c \ ;$$

its sequential and concurrent components, DP^s and DP^c, respectively, are

$$DP^s = \sum_{e_i \in \tau_m^s} T(I_{eid_i})$$

and

$$DP^c = \sum_{e_j \in \tau_m^{cp}} T(I_{eid_j}) \ .$$

Although (1) estimates the instrumentation perturbation, it does not estimate actual execution time from trace data. The *approximate* execution time, $T_a(P)$, is the difference between the measured execution time and direct perturbation,

$$T_a(P) = T_m(\mathcal{I}(P)) - DP \ .$$

That is, $T_a(P)$ is the approximated execution time after applying a perturbation analysis model that includes only direct perturbations.

3 Time-Based Models

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. Perturbation models then must describe observed (measured) behavior as a perturbation of true behavior. Perturbations are manifest in event execution times and event ordering. For timing analysis, perturbation models must approximate true times of event occurrence, either for each trace event or for the total execution time. Event analysis is more difficult; program or system semantic information is needed to determine if the relative, observed event order is incorrect and, if so, generate a better approximation to the actual order. In both cases, perturbations models must use the execution information contained in a measured trace, τ_m, to resolve the instrumentation perturbations that occurred during the measurement and to approximate actual performance behavior.

We consider only perturbation models for timing analysis. The two types of models we develop, time-based and event-based, differ in their assumptions about program execution. Time-based perturbation models assume independence among threads of execution and, therefore, account only for the execution time overhead of the instrumentation when constructing performance approximations from the measured traces. Event-based models use measurements of synchronization operations to account for dependent execution, and maintain ordering relationships during perturbation analysis. The time-based models used for approximating total execution time and per event times are discussed below. The event-based models are described in §4.

3.1 Total Execution Time Analysis

Time-based models capture the effects of instrumentation perturbation when the time and order events occur is execution independent. For sequential execution, the execution states form a total order, and event times are affected only by instrumentation overhead. For concurrent execution scenarios, mainly those involving simple fork-join behavior and no inter-thread dependencies, time-based perturbation models also apply. We begin with the sequential execution case.

3.1.1 Sequential Total Time Model

During sequential execution, the principal perturbation is direct — execution of additional instrumentation instructions.[1] Furthermore, instrumentation does

1. This does not means that indirect sources of perturbations do not exist. Rather, the execution time overhead is known to

not perturb the total order of program events. Thus, our sequential perturbation model assumes that *all* perturbations are direct (i.e., $AE = DP$) and that the cost for instrumentation is decoupled from statement execution. Simply put, the model approximates actual execution time by the difference between measured execution time and all direct instrumentation costs. More formally, the model's assumptions imply the following:

1. The actual cost $T(I_{eid_i})$ for each instrumentation point I_i is approximated by a constant α.[2]
2. $DP = \sum_{e_i \in \tau^s} T(I_{eid_i}) = \alpha N$, where N is the number of instrumentation points.
3. $T_a(P) = T_m(\mathcal{I}(P)) - DP = T_m(\mathcal{I}(P)) - \alpha N$.

As the approximate equality above suggests, the accuracy of our assumption depends on the interaction of instrumentation perturbations and statement execution. With source code instrumentation, compiler register optimizations can invalidate the assumption of a constant instrumentation perturbation. The desired approximation in this case would be based on a non-uniform perturbation model (i.e., one that considered the effects of each individual instrumentation instance). However, the number of different cases to consider can become large and requires an analysis of the differences in the code generated with and without instrumentation. In light of these complications, we assume that a reasonably stable instrumentation overhead can be obtained and that the constant overhead estimate is valid.

3.1.2 Concurrent Total Time Model

During concurrent execution, multiple threads of control may simultaneously reach trace instrumentation points. Intuitively, a critical path analysis would identify the set of instrumentation points needed to compute total execution time [15]. Unfortunately, different parallel perturbations can make this difficult. Events can be reordered, and the critical path identified from the instrumentation trace may not be the critical path in the real code.

Without resorting to the event analysis models of §4, we can assume that events are *not* reordered and that the concurrent thread with the longest execution time (after direct perturbations have been removed) is the critical path. If most threads execute similar instruction streams (i.e., there is little data dependent code), this assumption is accurate.[3] Like the sequential execution time model, our base assumption implies the following:

1. The actual cost $T(I_{eid_i})$ for each instrumentation point I_i is approximated by a constant α.
2. $DP = DP^s + DP^c_{max} = DP^s + \alpha N_{max}$, where
   - $(T_m(\mathcal{I}(P_{max})) - \alpha|\tau^{max}|) \geq (T_m(\mathcal{I}(P_i)) - \alpha|\tau^i|) \;\; \forall i \;\; 0 \leq i \leq p$,
   - p is the number of processors,
   - $T_m(\mathcal{I}(P_i))$ is the measured concurrent execution time on processor i,
   - $N_{max} = |\tau^{max}|$ is the number of instrumentation events in trace τ^{max}.
3. $T_a(P) = T_m(\mathcal{I}(P)) - DP^c = T_m(\mathcal{I}(P)) - DP^s - \alpha N_{max}$.

The concurrent perturbation model chooses as the critical path the sequential execution path plus the execution path along the concurrent thread that has the greatest accumulated execution time after the direct perturbation has been removed.

3.2 Event Timing Analysis

Although total execution time approximations can serve as partial validation of the time-based models, ultimately we want to recover the actual order and timing of trace events. However, even given careful analysis and a predictive event timing model, one cannot directly determine the accuracy of the predicted event times. Instead, one must infer the stability of an event timing model by comparing its trace predictions with varying levels of trace instrumentation. As with execution time models, we begin with the simpler, sequential case.

3.2.1 Sequential Event Time Model

Each trace event identifies a unique spatial and temporal state (i.e., a code location at a specified time). In a sequential trace, each event is perturbed by the instrumentation for all previous events. Thus, we can iteratively calculate each event time, given the perturbations of previous events.

For a trace, τ^s, of sequential events $e_1, \ldots, e_m$ where each e_i is of the form $\{t_m(e_i), eid_i\}$, the model approximates the actual time of event e_i by $t_a(e_i)$,

$$t_a(e_i) = t_m(e_i) - (i-1)\alpha \;, \qquad (2)$$

where α is the mean time for each trace instrumentation point and $t_m(e_i)$ is the measured time of occurrence of e_i from a trace of an instrumented execution.

3.2.2 Concurrent Event Time Model

Approximating event times for concurrent traces is more difficult than for sequential traces. The perturbation of each event depends on the perturbation of all events on the critical path to the event. In the worst case, a complete characterization of the execution dependencies between concurrent threads of execution is required. To simplify analysis, we assume that events

---

occur with every instrumentation execution, where the indirect perturbations are less likely and deterministic.

2. The approximate to $T(I_{eid_i})$, α, is given by the mean instrumentation time overhead.

3. If not, an event analysis model is needed. However, in [9, 11], we show that timing analysis alone can yield significant insight in many practical cases.

on separate concurrent threads are independent and that the program contains only a single level of fork-join concurrency.[4] With the event-based models, these assumptions can be relaxed.

Given a trace, τ^i, of concurrent events $e^i_1, \ldots, e^i_n$ for each concurrent thread i, and a trace, τ^s, of sequential events $e^s_1, \ldots, e^s_m$,[5] we approximate the actual time of a concurrent event e^i_k as follows.

1. If e^i_k is the first concurrent event after a sequential event e^s_j in the time ordered trace, then

$$t_a(e^i_k) = t_m(e^i_k) - t_m(e^s_j) + t_a(e^s_j) \ .$$

   We use the measured and approximated times of the last sequential event occurrence as the *time basis* for computing the execution time of the first concurrent event of a concurrent phase of computation.

2. If e^i_k immediately follows a concurrent event in the trace on thread i, then

$$t_a(e^i_k) = t_m(e^i_k) - t_m(e^s_j) + t_a(e^s_j) - \alpha c^i_k \ ,$$

   where c^i_k is the number of events in concurrent thread i after the last sequential event e^s_j in the trace. Along a sequence of concurrent events, we use the last sequential event as the time basis for approximating the time of occurrence of e^i_k, but the direct perturbation along thread i also is removed.

We approximate the actual time of a sequential event e^s_k as follows.

1. If e^s_k is the first sequential event in the trace after the last concurrent event from a concurrent phase of computation, then

$$t_a(e^s_k) = t_m(e^s_k) - t_m(e^i_j) + t_a(e^i_j) \ ,$$

   where $t_a(e^i_j) > t_a(e^m_n)$ for all n and m such that e^j_i and e^m_n appear before e^s_k in the trace. It is here that we determine the critical concurrent path in the instrumented execution. The concurrent event appearing before e^s_k in the trace with the greatest approximated timestamp is used as the time basis to approximate the sequential event occurrence.

2. If e^s_k follows a sequential event in the trace, then

$$t_a(e^s_k) = t_m(e^s_k) - t_m(e^i_j) + t_a(e^i_j) - \alpha c^s_k \ ,$$

   where c^s_k is the number of events that have occurred in sequential mode since the last approximated concurrent event e^i_j in the trace (or the beginning of the trace). Along a sequence of sequential events, we again use the last approximated concurrent event as the time basis for approximating the sequential event occurrence. Additionally, we remove the direct sequential perturbation.

4. Multiple phases of sequential and concurrent computation, and hierarchical fork-joins are allowed.

5. A trace from a parallel execution is a sequence of sequential and concurrent sub-traces.

4 Event-Based Models

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.

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 re-assignment of computational resources, 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 [3, 4, 14], perturbation analysis must recover the actual run-time performance behavior from a perturbed performance measurement.

If performance instrumentation is designed correctly [5], an un-instrumented parallel execution that satisfies Lamport's *sequential consistency* criterion[6] [7, 8] implies that the performance measurement will be *non-interfering* and *safe* [5]. 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[7] will remain unchanged [12]. Thus, perturbation analysis begins with a total ordering of measured events consistent with the *happened before* relation [6] 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 (to be discussed below) [9] 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

6. 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.

7. Helmbold and Bryan refer to the set of feasible executions defined by the partial order of program events as the *partially ordered set* [5].

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. 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.

4.1 Synchronization Models

The approach we use for event-based perturbation analysis is to identify the synchronization operations used in a program to enforce execution dependencies and instrument the operations to capture event timing and ordering information. Although the instrumentation can perturb the measurements, knowledge of synchronization semantics, together with the measured data, can be used to prevent execution ordering violations in the approximations made by event-based analysis. The perturbation models used then are synchronization models, describing both the semantics of synchronization operations and the time approximation techniques for measured synchronization events.

There are many possible forms of synchronization operations that could be analyzed [1]; we consider three: barrier, semaphore, and advance/await synchronization. These were chosen because they represent a cross-section of synchronization alternatives. Our goals in analyzing these three forms of synchronization are, first, to determine what measurements must be made to apply perturbation analysis, and, second, to understand the approximation capabilities of the perturbation analysis techniques. We begin with barrier synchronization.

4.2 Barrier Perturbation Model

Simply, a *barrier* is a piece of code used to synchronize multiple threads of execution at a single point in time. Each thread participating in a barrier synchronization will first enter the barrier, wait for all the other threads to arrive, and then exit the barrier with all other threads. The unique feature of the barrier is that all threads will block until the last thread enters, establishing a point in the computation where the states of all threads participating in the barrier synchronization are known. This point occurs when the barrier synchronization has been satisfied and all threads are allowed to proceed.

4.2.1 Barrier Instrumentation

The performance instrumentation of a barrier should allow one to determine the sequence of thread arrivals at the barrier, the waiting time of each thread, and the time the threads exit the barrier [2]. This analysis requires capture of two barrier events for each thread: *enter* and *exit*. The *enter* event is recorded immediately after a thread enters the barrier and the *exit* event is recorded immediately before the thread exits the barrier. From the standpoint of perturbation analysis, we are primarily interested in the *exit* events — these establish a time basis for all following events on each thread; see below. However, the *enter* events are also important for barrier performance analysis.

Consider Figure 1 which shows four threads synchronizing at a barrier b. When a thread reaches the barrier, it executes instrumentation corresponding to barrier entry (indicated by light shading) before executing the barrier code. After the last thread reaches the barrier and the threads have synchronized, they all execute instrumentation code corresponding to barrier exit (indicated by darker shading) before continuing along their separate execution paths.

The barrier events are recorded in a trace for each thread. To uniquely identify a particular barrier, we assume the *entry* and *exit* events recorded are additionally typed with this information. The ability to distinguish different barriers is required by the perturbation analysis.

The measured timeline shows an example of how the threads might have executed at the barrier. Diagrammatically, we represent the barrier's synchronization code executed after the last thread reaches the barrier by $\square$. The enter and exit barrier instrumentation and the waiting time on each thread are shown. The timeline reflects only the measured execution behavior. It is clearly possible that instrumentation on each thread prior to the barrier can affect the order that threads arrive. Thus, the timing relationships shown between the *entry* and *exit* events may not be representative of actual barrier behavior.

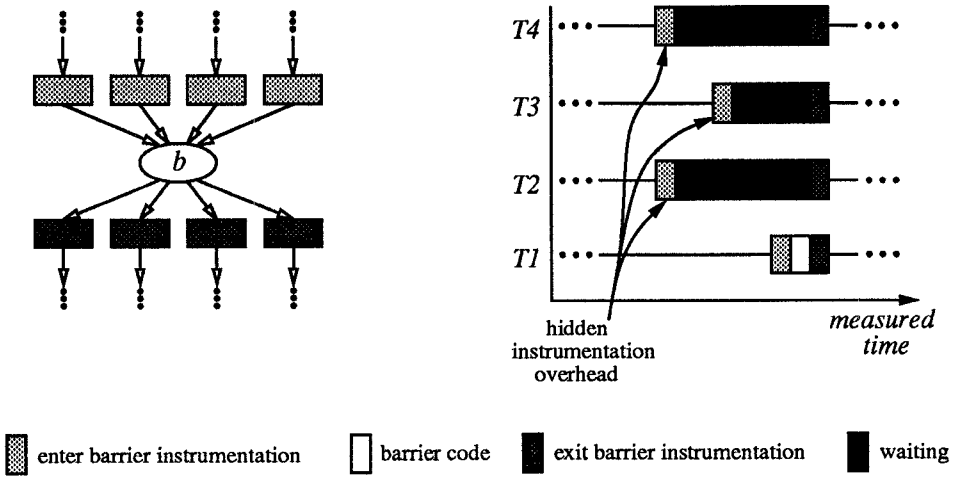


Figure 1: Barrier Synchronization of Four Threads with Instrumentation

4.2.2 Barrier Approximation

The perturbation analysis of the instrumented barrier is based solely on the *exit* events. The *enter* events are used to approximate certain performance characteristics of the barrier, but it is the *exit* events that are used to resolve the time approximations of future events. Figure 2 shows a possible timeline of the approximated barrier execution. We assume the *enter* events for each thread, whose approximated time of occurrence is represented by the white arrows, have been previously resolved by perturbation analysis. The approximated time the first thread enters the barrier is denoted by $t_a(enter_f)$ and the approximated time the last thread enters the barrier is denoted by $t_a(enter_l)$.

The perturbation analysis problem is to approximate the barrier exit behavior. By assumption, all threads are known to be present at the barrier at approximated time $t_a(enter_l)$. The approximated execution time of the barrier code after this point is given by

$$t_m(exit_f) - t_m(enter_l) - \alpha,$$

where $t_m(exit_f)$ is the measured time the first thread exits the barrier, and $t_m(enter_l)$ is the measured time the last thread enters the barrier. The approximated time the first thread exits the barrier is

$$t_a(exit_f) = t_a(enter_l) + t_m(exit_f) - t_m(enter_l) - \alpha.$$

Because we use $t_m(exit_f)$ in computing $t_a(exit_f)$, the thread first to exit the barrier in the measured execution is the same one that exits first in the approximated execution. The approximated time the jth thread exits the barrier after the first is given by

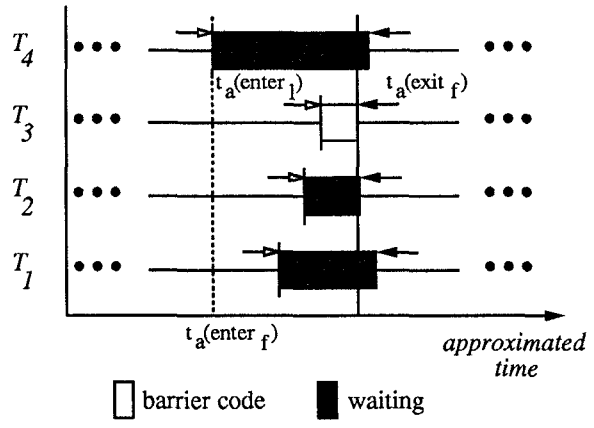
$$t_a(exit_j) = t_a(exit_f) + t_m(exit_j) - t_m(exit_f) - \alpha,$$

where the event $exit_j$ is obviously the *exit* event of the jth thread. The ordering relationships of barrier exit in the measured execution are maintained in the approximated execution with $t_a(exit_f)$ serving as the time basis for all other *exit* event approximations.

The barrier perturbation analysis assumes that the overhead for executing the barrier code remains unchanged from the measured to the approximated execution. In general, the execution of the barrier code can take variable time, depending on the barrier's implementation. For instance, if a single lock is used to control access to a counter indicating the number of threads which have entered the barrier, the lock becomes an access "hot spot" [13] in the computation. The performance of barrier code implemented in this manner will depend on the degree of lock contention; in general, the higher the contention, the poorer the performance. Because the measured thread arrival behavior can be different from the actual behavior, there can be errors in the perturbation analysis because of differences in barrier code performance. These errors can be both positive (over-approximations due to poorer barrier code performance in the measured execution) and negative (under-approximations due to poorer barrier code performance during the actual execution). Ideally, the barrier performance would be modeled by the perturbation analysis based on known performance behavior of the barrier implementation. The barrier entry event times for each thread would provide barrier entry timings to the model.

4.3 Semaphore Perturbation Model

The perturbation analysis of barrier synchronization could be easily understood because the barrier creates a global synchronization point. If all threads meet at a barrier synchronization, the perturbation analysis of events after the barrier will not be affected by the errors resulting from the perturbation analysis of events



$$t_a(exit_f) = t_a(enter_l) + t_m(exit_f) - t_m(enter_l) - \alpha$$
$$t_a(exit_j) = t_a(exit_f) + t_m(exit_j) - t_m(exit_f) - \alpha$$

Figure 2: Barrier Performance Approximation

prior to the barrier. In effect, the barrier provides event-based perturbation analysis with a means of partitioning the computation between barrier synchronizations and isolating perturbation analysis errors to individual partitions.

Not all synchronization operations are as well behaved. In particular, the semaphore represents one of the most primitive forms of synchronization, and hence one of the least restrictive. The semaphore establishes a synchronization relationship between only two events and, thus, does not have the global synchronization ramifications of the barrier. Furthermore, the pairing of semaphore synchronization events can change from actual to measured execution.

The basic *semaphore* embodies the most recent history of two operations: **P** and **V**. The **P** operation checks the state of the semaphore to determine whether a dependency has been satisfied. The **V** operation signals that some execution dependency has been satisfied. The semaphore's state indicates only whether the last operation on the semaphore was a **P** or a **V**. If it was a **V**, the next **P** operation will not wait. Otherwise, the next **P** operation will wait. If S is a semaphore whose value can be P or V, the semantics of the **P** and **V** synchronization operations are shown below:

P(S): if (S equals V)
  $S = P$
else
  wait until S equals V
  $S = P$

V(S): $S = V$.

More complex semaphores, such as counting semaphores, maintain additional history of **P** and **V** operations as well as allow multiple threads to wait. From a perturbation analysis perspective, we will consider only the basic semaphore case. All the problems we encounter with this case also appear in the more complex semaphore types.

4.3.1 Semaphore Instrumentation

There are five important events that must be monitored in the semaphore's operation: the V event, the P_s event, the P_w event, the P_p event, and the P_e event. The V event corresponds directly to the **V** operation on the semaphore and is recorded immediately after the **V** operation is performed. The P_s event is recorded immediately before the **P** code is executed and signifies that the semaphore state is about to be tested. If the **P** operation blocks, the P_w event is recorded before the thread begins to wait; otherwise, the P_p event is recorded before the semaphore state is set to P. Finally, the P_e event is recorded immediately before a thread performing a **P** operation leaves the semaphore.

Because multiple semaphores may be active in a program, to correctly identify the **P** and **V** operations on a particular semaphore during perturbation analysis, information uniquely identifying the semaphore must must be recorded with the P and V events.

4.3.2 Semaphore Approximation

The perturbation analysis of a semaphore's instrumentation is concerned not so much with the removal of instrumentation overhead that comes with recording the P and V events, although this is important to achieve timing accuracy, but rather with identifying anomalous

semaphore operation that might occur in the approximated execution as a result of perturbation analysis errors. In fact, this is the main reason for studying the semaphore form of synchronization. Under a limited set of assumptions regarding how a semaphore is used during execution, we looked at several cases where questions arise concerning how event-based perturbation analysis should resolve approximated semaphore behavior [9]. To simplify the discussion, we assume in the following cases that an equal number of **P** and **V** operations are performed on each semaphore. Furthermore, we discuss only the single-P, single-V case.

Single-P, Single-V

If a semaphore is used by only two threads, every **P** operation will be "satisfied" by a unique **V** operation; success of the ith **P** operation will depend only on the ith **V** operation. We refer to such a semaphore as a *single-P, single-V* semaphore. The P and V events for this semaphore can be matched explicitly by the perturbation analysis based on their order of occurrence. For each ith P-V pair, the time-order relationship of the P and V events must be maintained in the approximated execution.

Figure 3 shows three different approximated executions that can result from the perturbation analysis of a single-P, single-V semaphore. The left graph in each case shows the ordering relationship of the P and V events in the measured execution. The right graph shows the events in the approximated execution. In case A, no waiting is encountered at the semaphore in the measured execution because $t_m(V) < t_m(P_s)$. However, in the approximated timeline, $t_a(P_s) < t_a(V)$ and tread T_2 must wait. We assume events P_s and V have been resolved in the approximated execution and that $t_a(P_p)$ and $t_a(P_e)$ must be determined. There is no information from the measured execution to indicate how long after event V the event P_p occurs in the approximated execution, so we must assume it is immediate.[8] This establishes $t_a(P_p)$ as the time basis for approximating $t_a(P_e)$. Thus,

$$t_a(P_e) = t_a(P_p) + t_m(P_e) - t_m(P_p).$$

The amount of waiting time can be calculated as

$$t_a(P_p) - t_a(P_s) + (t_m(P_p) - t_m(P_s)).$$

In case B, the opposite conditions exist. That is, waiting occurs in the measured execution but not in the approximated execution. Instead of the event P_p indicating the **P** operation succeeded, we are interested in when waiting begins. The P_w event provides a time basis for $t_a(P_e)$ in the approximation. We can calculate from the measured execution how long after event V the event P_e occurs in the approximated execution. From this value, the approximated time of P_e is given by

$$t_a(P_e) = t_a(P_w) + t_m(P_e) - t_m(V).$$

Because the approximated time of the V event is not affected by semaphore waiting, it is possible that the situation in case C may be encountered due to perturbation analysis errors. As shown, two successive V events, V^i and V^{i+1}, are approximated to occur prior to the ith P events. This violates the assumed operation of the semaphore. There are two recourses if this situation occurs. The perturbation analysis could halt, indicating that a violation of semaphore execution semantics has occurred. A less abrupt action logs the violation, adjusts the approximated time of V^{i+1} to be immediately after $t_a(P_e)$ (see figure), and continues the perturbation analysis.

Notice that the situation presented in case C cannot be true of the approximated time of the P events. That is, given two successive P_e events in the trace, P_e^i and P_e^{i+1}, and the ith V event, V^i,

$$t_a(V^i) < t_a(P_e^{i+1}).$$

The approximated time $t_a(P_e^{i+1})$ will always be adjusted by the perturbation analysis to be greater than V^i.

4.4 Advance/Await Perturbation Model

Conservative semaphore perturbation models strictly enforce the P-V event pairing found in the measured execution in the approximation. However, as observed in the perturbation analysis of semaphores involving multiple **P** and/or **V** operations (see [9]), reordering of measured P-V event pairs in liberal approximations (to give better approximations and smaller execution times) fundamentally depends on what is known about the execution behavior between concurrent threads. This knowledge cannot be measured from the execution.

Semaphores can be used for general synchronization operations in nondeterministic programs and, thus, the potential perturbations of event order can be varied. However, the *advance/await* form of synchronization makes explicit the "post" and "wait" actions involved in programs using the **advance** and **await** operations to enforce execution ordering dependencies. Because the synchronization is strictly enforced, a partial ordering of these actions can be determined directly from the measured *advance* and *await* events. Independent of how **advance** and **await** operations are assigned to threads of execution, this partial ordering must be maintained by an advance/await perturbation model (even a liberal one) in the approximated execution.

8. Performance measurements of semaphore operation could be applied here to establish a minimum separation between V and P_p.

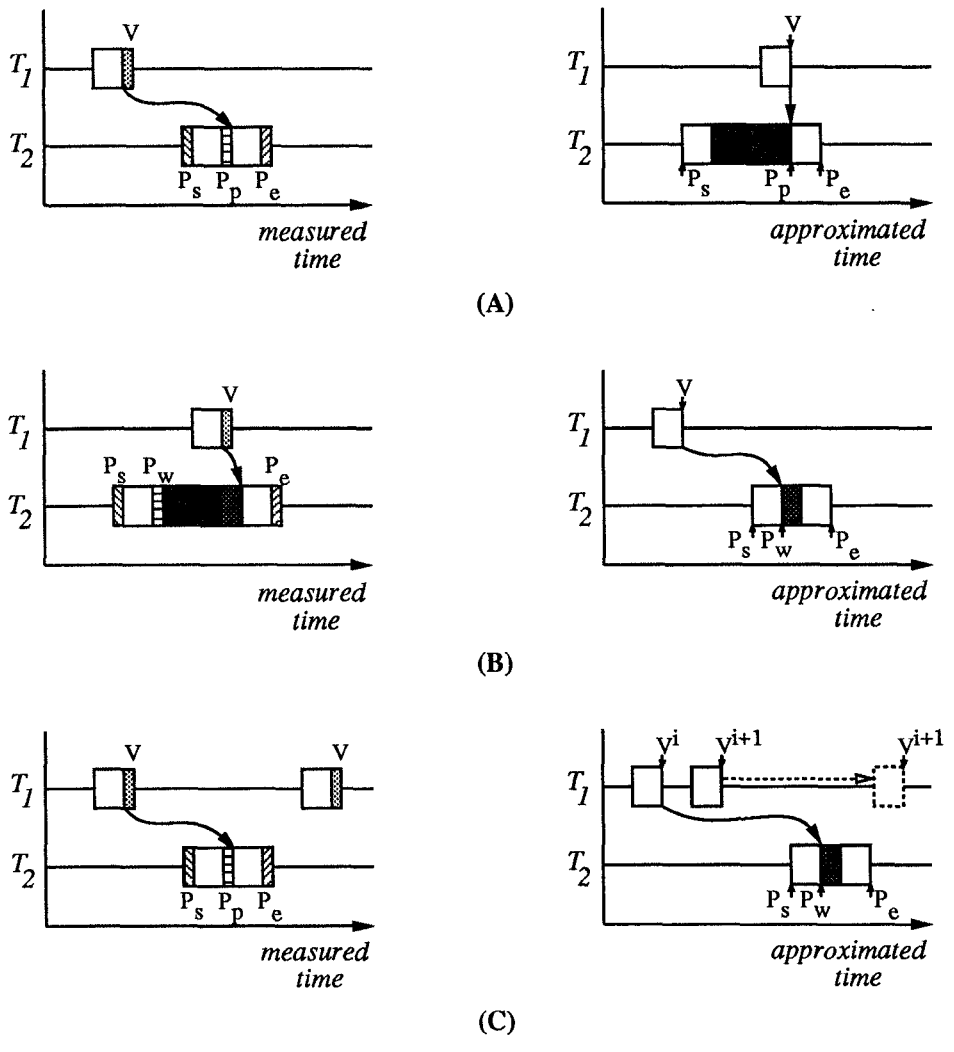


Figure 3: Single-P, Single-V Perturbation Analysis

Without knowledge of the scheduling policy for assigning work to threads, the perturbation analysis must maintain not only the partial order of the *advance* and *await* events but also the event-to-thread assignment.

In this case, the conservative advance/await perturbation model is very similar to the general single P, single V semaphore model.[9] Obviously, due to prior perturbation removal, the relative ordering of the pieces of work bounded by advance/await synchronization, as well as the waiting delays, may change from the measured execution. If additional information is made available to the perturbation analysis indicating that the work bounded by advance/await synchronization can be assigned to any available thread as long as the partial order is maintained, alternative scheduling strategies could be applied in the approximation to reduce waiting delays. The result could be a smaller total execution time approximation, with corresponding reductions in waiting delay in the threads of execution. In theory, the perturbation analysis could even compute a minimum critical path execution time for the partial order computation by applying a generalized bin packing algorithm that ignores thread scheduling issues.

However, the same basic question arises in the case of advance/await perturbation analysis as with semaphore synchronization. What is reasonable for the perturbation analysis to be able to assume about how the work between synchronization points is executed? The conclusion is that without additional knowledge from the user regarding work independence and reordering, no assumptions can be made without risking approximation errors or causing execution ordering violations.

9. The full advance/await perturbation model can be found in [10]. We will not repeat the discussion here.

5 Experimental Results

The time-based and event-based perturbations models have been validated using a series of instrumentation experiments [9, 11, 10]. The results of these experiments suggest that a systematic application of performance perturbation analysis techniques will allow more detailed, accurate instrumentation in practice. A complete report of these experimental results can be found in [9].

6 Conclusion

Perturbation due to instrumentation has two effects on the events occurring during concurrent execution: temporal effects and resource assignment effects. In addition to the slowdown caused by instrumentation overhead, temporal effects include possible event reorderings as the measurement alters the set of likely partial order executions. Resource assignment effects occur because the instrumentation changes the dynamic resource demands. In instances where the computation dynamically adapts to resource availability, instrumentation can perturb resource allocation and utilization. This is particularly important to understand with respect to processor assignment. Performance approximations can differ significantly from actual execution unless resource assignment effects are taken into account.

Unfortunately, the conservative perturbation models do not attempt to quantify performance effects dues to dependent event re-ordering or resource use. Any perturbation analysis approximation must be safe [5] (i.e., must not violate the partial ordering relationships) and, thus, must be provided sufficient measurements that capture the ordering dependencies during execution. However, the accuracy of perturbation analysis depends not only on more precise synchronization measurements, but also on additional knowledge of actual (likely) execution behavior, unattainable from measurements alone. This can include the scheduling policies used by a program, more detailed data dependency information, and even the performance characterization of certain synchronization operations (e.g. barrier performance). This information can be used together with the performance measurements obtained from the event data to drive (in a sense) a simulation of the execution. Although the approximations resulting from this more "liberal" perturbation analysis approach are potentially closer to the set of likely executions, such a result is difficult to analytically verify.

References

[1] Andrews, G., and Schneider, F. Concepts and Notations for Concurrent Programming. *ACM Computing Surveys 15*, 1 (Mar. 1983), 3–43.

[2] Axelrod, T. Effects of Synchronization Barriers on Multiprocessor Performance. *Parallel Computing 3*, 2 (Nov. 1986), 129–140.

[3] Emrath, P., Ghosh, S., and Padua, D. Event Synchronization Analysis for Debugging Parallel Programs. In *Proc. of the 1989 International Conference on Parallel Processing* (Aug. 1989).

[4] Fidge, C. Partial Orders for Parallel Debugging. In *Proc. of the Workshop on Parallel and Distributed Debugging* (May 1988), ACM Sigplan/Sigops, pp. 183–194. Univ. of Wisconsin.

[5] Helmbold, D., and Bryan, D. Design of Run Time Monitors for Concurrent Programs. Tech. Rep. CSL-TR-89-395, Stanford Univ., Oct. 1989.

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

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

[8] Lynch, N., and Fisher, M. Semantics of Concurrent Computations. *Theoretical Computer Science 13*, 1 (Jan. 1981), 17–43.

[9] Malony, A. *Performance Observability*. PhD thesis, Univ. of Illinois at Urbana-Champaign, Department of Computer Science, Sept. 1990.

[10] Malony, A. Event Based Performance Perturbation: A case study. In *Third ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming* (Apr. 1991), pp. 201–212.

[11] Malony, A., Reed, D., and Wijshoff, H. Performance Measurement Intrusion and Perturbation Analysis. Tech. Rep. CSRD-923, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, Oct. 1989. to be published in the IEEE Transactions on Parallel and Distributed Computing.

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

[13] Pfister, G., and Norton, V. Hot Spot Contention and Combining in Multistage Interconnection Networks. In *Proc. of the 1985 International Conference on Parallel Processing* (Aug. 1985), pp. 790–797.

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

[15] Yang, C., and Miller, B. Critical Path Analysis for the Execution of Parallel and Distributed Programs. In *Proceedings of the 8th International Conference on Distributed Computing Systems* (June 1988), pp. 366–375.