Performance Measurement Intrusion and Perturbation Analysis

Allen D. Malony, *Member, IEEE*, Daniel A. Reed, *Member, IEEE*, and Harry A. G. Wijshoff

Abstract—**Software performance instrumentation perturbs the state of the measured system. The primary source of this perturbation is the execution of additional instructions. However, ancillary perturbations include disabled compiler optimizations and memory conflicts. Collectively, these perturbations can increase the measured system's execution time, change memory reference patterns, reorder events, and even cause register interlock stalls. The perturbation magnitude depends on the intended performance measurements and the granularity of the instrumentation.**

In this paper, we study the instrumentation perturbations of software event tracing on the Alliant FX/80 vector multiprocessor in sequential, vector, concurrent, and vector-concurrent modes. Based on experimental data, we derive a perturbation model that can approximate true performance from instrumented execution. We analyze the effects of instrumentation coverage (i.e., the ratio of instrumented to executed statements), source level instrumentation, and hardware interactions.

Our results show that perturbations in execution times for complete trace instrumentation can exceed three orders of magnitude. With appropriate models of performance perturbation, these perturbations in execution time can be reduced to less than 20% while retaining the additional information from detailed traces. In general, we conclude that it is possible to characterize perturbations through simple models. This permits more detailed, accurate instrumentation than traditionally believed possible.

Index Terms—**Instrumentation intrusion, instrumentation uncertainty principle, performance measurement, perturbation analysis.**

I. Introduction

SYSTEMATIC application of the scientific method is the foundation of modern science. Central to this *Weltanschauung* is the experimental testing of hypotheses and the operational paradigm (i.e., if an experiment cannot be constructed, even in principle, to measure a phenomenon, it cannot, operationally, be said to exist). In any field, experimental progress is inextricably coupled with technological advances; the latter provide the requisite tools to more accurately measure known phenomena and to test hypotheses that predict the existence of heretofore undetected phenomena. The richness of the computer system design space, with its seemingly infinite variations, makes the scientific method's systematic measurement and hypothesis testing both appropriate and desirable [4]. However, because computer system design is an experimental science, its practitioners are prey to many of the same instrumentation pitfalls facing any experimental scientist, notably uncertainty and instrumentation perturbation.

A multiplicity of measurement levels permeate all experimental sciences, including computer system performance analysis. For computer systems, the lowest level includes performance measurements of the hardware design. Determining this performance provides both a design validation and directives for system software design. Only by understanding the strengths and weaknesses of the hardware can system software designers develop an implementation and user interface that maximizes the fraction of the raw hardware performance available to the end user. Given some characterization of the balance between system resources, users can develop algorithms that best optimize their use. Finally, the best mix of key algorithms will maximize the performance of user applications. Together, these measurement levels form a hierarchy of experimental observations and associated theory that describe the system's performance behavior.

The detailed measurement obtained at low levels is not without price. In nineteenth century physics, advances in instrumentation technology and statistical analysis precipitated a theoretical crisis—measured data could no longer be reconciled within the existing theoretical framework. The resulting theoretical revolution defined limits on the accuracy and possible perturbations of experimental measurement. In physics, the Heisenberg uncertainty principle bounded the attainable measurement certainty. Moreover, it became clear that instrumentation and phenomenon must be commensurate to maintain instrumentation perturbations at acceptable levels (e.g., imaging very small systems requires electron, rather than optical, microscopy).

In computer system performance analysis, the problems of uncertainty and perturbation are no less profound. 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.[1] Excessive instrumentation perturbs the measured system; limited instru-

Manuscript received October 6, 1990; revised May 21, 1991. This work was supported in part by the National Science Foundation under Grant US NSF MIP-8410110, the Department of Energy under Grant US DOE-DE-FG02-85ER25001, the Air Force Office of Scientific Research under Grants AFOSR-85-0211 and AFOSR-86-0147, and a donation from IBM.

A. D. Malony is with the Department of Computer and Information Science, University of Oregon, Eugene, OR 97403.

D. A. Reed is with the Department of Computer Science, University of Illinois, Urbana, IL 61801.

H. A. G. Wijshoff is with the Department of Computer Science, Leiden University, Leiden, The Netherlands.

IEEE Log Number 9107405.

[1] In contrast to other experimental disciplines, computer systems instrumentation does permit the epistemological trickery of declaring instrumentation part of the system. The perturbation is then, *ipso facto*, null. In this paper, we exclude such possibilities.

mentation 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, event reordering, and even register interlock stalls. From a performance evaluation perspective, instrumentation perturbations must be balanced against the need for detailed performance data. Regrettably, there are no formal models of performance perturbation that would permit quantitative evaluation from instrumentation costs, measured event frequency, and desired instrumentation detail.

Given the lack of models and the potential dangers of excessive instrumentation, detailed 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 unduly pessimistic. To test this hypothesis, we conducted a series of instrumentation experiments to determine the magnitude of performance perturbations as a function of instrumentation frequency and execution mode (i.e., sequential, vector, and parallel). Based on this experimental data, we derived a perturbation model that can approximate true performance from instrumented execution. This permits more detailed, accurate instrumentation than traditionally believed possible.

We begin in Section II by describing the experimental environment, the Alliant FX/80, a shared memory, vector multiprocessor. In Section III, we discuss the instrumentation perturbations possible on this parallel system. The models of performance perturbation that permit removal of performance perturbations from instrumented programs are developed in Section IV. A description of the instrumentation environment is given in Section V. In Section VI and Section VII, we validate the models of Section IV using experimental data obtained from execution of the Livermore Loops. Finally, Section VIII summarizes our results and suggests avenues for further research.

II. Experimental Environment

Our hypothesis implies that the perturbations attributable to detailed performance instrumentation can be quantified in sequential, vector, and parallel execution modes. Ideally, each experiment systematically measures the effects of one, and only one, variable with respect to a control. Consequently, we conducted all experiments on a single system, the Alliant FX/80, and used the system's nominal behavior as reference. To understand the results of these experiments, we digress to review both the architecture and the idiosyncrasies of the FX/80; see Fig. 1.

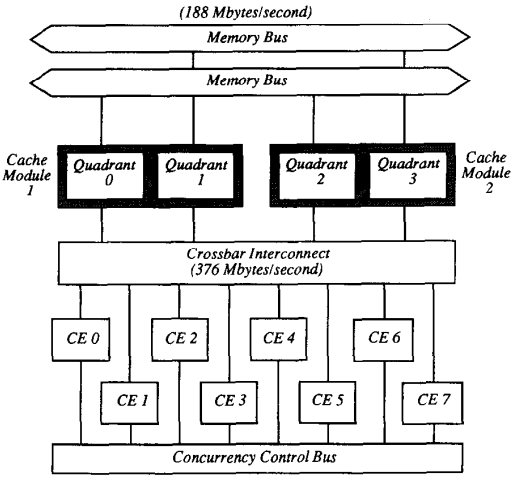


Fig. 1. Alliant FX/80 Architecture

The Alliant FX/80 consists of up to eight computational elements (CE's), each containing a vector processor [13]. The CE's are connected via a concurrency control bus that permits dispatching of small computation granules to cooperating CE's. Using this bus, parallel loop iterations can be directly allocated to the CE's through a hardware self-scheduling mechanism.

The memory system of the Alliant FX/80 combines parallel data access with a hierarchical memory structure. It is organized as three levels: a large main memory, a 512K byte cache shared by all CE's, and scalar and vector registers private to each CE. Each vector register contains 32, double precision (64-bit) words and is accessed by each CE's vector processing unit. The 64K word, write-back cache contains four banks that are connected to the eight CE's via a crossbar switch. The cache can service up to eight simultaneous accesses per cycle. The cache and the four-way interleaved main memory are connected via a main memory bus with a peak transfer rate of four words per cycle. Therefore, the peak bandwidth between main memory and the CE's is half that between the cache and the CE's.

The CE instruction set is a variation of the Motorola 68020 with certain extensions (e.g., vector and concurrency instructions). The Alliant vector instructions are of two types:

- *Internal*: all operands are contained in vector and scalar registers.
- *External*: one operand involves a memory request.

Within each type, most vector instructions have similar timing behavior, typically differing only in their startup time. Because the internal, register–register instructions do not depend on conditions extrinsic to each CE, their timings are deterministic. In contrast, the timing behavior of external vector instructions depends on memory activity. Access contention, either from the CE's previous requests or from requests issued by other CE's, and data location, either cache or main memory, both contribute to delays.

The memory hierarchy, multiple CE's, vector memory operations, and the concurrent execution modes all make the Alliant FX/80 a complex experimental environment. Successful hypothesis validation in this environment would provide strong evidence that our performance perturbation models are applicable to parallel systems with simpler execution environments (e.g., the multiple processor Cray X-MP [3] or the distributed memory Intel iPSC/2 [2]).

III. Performance Perturbation

The number and complexity of the Alliant FX/80's execution modes are equaled in both number and complexity by perturbation mechanisms for performance instrumentation. Unfortunately, perturbations are not additive, nor can perturbation magnitudes easily be deduced from measurement data without knowledge of perturbation mechanisms. Thus, understanding these perturbation mechanisms is a prerequisite for analysis of experimental performance data and development of sequential and parallel performance perturbation models. Because the possible perturbations during sequential execution are but a subset of those possible during parallel execution, we begin with an analysis of the former.

A. Sequential Perturbations

Although the range of possible instrumentation perturbations depends on the complexity of the underlying architecture and system software, the single stream of control flow in sequential programs localizes most perturbations about the instrumentation point. The localization of perturbation effects means that the computation's performance behavior is only affected within a local region of the instrumentation. Although the timing error introduced by the perturbation accumulates during the performance measurement, if the perturbations are assumed local, we can empirically characterize different instrumentation perturbations in isolation and apply models that remove the timing errors at the instrumentation source; see Section IV. Below, we discuss the possible perturbations in the context of our experimental environment, the Alliant FX/80.

As discussed in Section II, the pipelined Alliant FX/80 processors are connected to a complex hierarchy of registers, cache, and primary memory. A sophisticated compiler generates code to maximize access frequency to the smaller, faster components of this hierarchy. For instance, at the lowest level of the memory hierarchy, pipelined register dependencies arise when an instruction accesses a register whose value has not yet been produced by a previous instruction. Although these dependencies stall the processor or functional unit until the requisite value is produced, optimizing compilers can reduce the number of stalls by judicious register allocation and instruction scheduling.

At this level, instrumentation perturbations need not increase execution time; instrumentation can both add and remove register dependencies. The former can occur when the prologue of the instrumentation code uses registers that have STOREs pending. Typically, most instrumentation points first save the active registers on the local stack, generate the desired performance data, and then restore the active registers. Because most registers are addressed during save and restore, dependencies between instructions just before or after the instrumentation point are highly probable. Conversely, register dependencies can be removed by inserting instrumentation between two instructions that have an existing register dependency. Execution of the intervening instrumentation code will decouple the dependent instructions.

Even if instrumentation does not perturb register dependences, it can change both the spatial and temporal patterns of cache and memory references, with both positive and negative effects. Consider an application code fragment that contains a loop with a high density of memory references. If the instrumentation generates large volumes of data, the resulting cache and memory traffic may flush most application data from the cache. When application execution resumes, the data cache will be reloaded by a sequence of cache misses. The overhead for this "cold start" [14] may be comparable to that for a context switch.

If an application's memory references generate bank conflicts in the interleaved memory [15], instrumenting the code will distribute the application memory references across a larger interval of time, decreasing the memory access time as perceived by the application code. During execution of the instrumentation code, all outstanding memory references, including those with bank conflict stalls, will compete. Because time spent in instrumentation code is not charged to the application, the apparent memory access time decreases. The converse is also true. If application data access patterns are structured to minimize bank conflicts, inserted instrumentation code can disrupt the access pattern, perhaps creating a degraded, steady-state memory access pattern with bank conflicts.

As the magnitude of direct instrumentation perturbation grows (e.g., added register dependencies or modified memory reference patterns), the probability of indirect perturbation grows proportionately. For example, the probability of context switches is higher for instrumented applications because they execute longer. Although the cost of these context switches can be measured by instrumenting the operating system, identifying "real" and induced context switches is nontrivial.

Perhaps more subtle than either direct perturbations or induced context switches are changes in the application program's executed code. Although the object code for a program with source code instrumentation (i.e., instrumentation inserted in the application source code) clearly differs from the object code for the same program without instrumentation, removing the instrumentation from the object code does not result in identical codes. For example, source code instrumentation can prevent certain optimizations and can change register allocation. For vectorizing and parallelizing compilers, the potential for code perturbation is greater—inserting instrumentation in a vectorizable loop can easily prevent vectorization.

B. Parallel Perturbations

As noted earlier, the class of possible perturbations during sequential execution is but a subset of those possible for parallel programs. Parallel programs often stress their

execution environment. For example, a single processor of the Alliant FX/80 cannot generate contention at the shared cache, nor can it saturate the memory bus. In concurrent mode, however, both the cache and memory bus can be performance bottlenecks [6]. This also is true of other parallel machines with high-performance memory systems [15]. Instrumentation that causes only small perturbations in sequential mode (e.g., memory traffic to save instrumentation data) can create unacceptable perturbations in concurrent mode, including perturbations of the task execution order.

With the exception of asynchronous input–output, the execution states of sequential programs form a total order. This is a principle accepted due to hardware properties of the machine. Sequential trace instrumentation may change event times, but it rarely changes the event order. In contrast, the states of parallel programs inherently form a partial order. Consequently, the reordering of instrumented states is not only possible but likely. The number of reordered events depends on both the task scheduling algorithm and the frequency of parallel task synchronization.

If parallel tasks are assigned to processors statically (i.e., the mapping of tasks to processors is known *a priori*), the sequence of tasks executed by each processor cannot change as a consequence of instrumentation. However, the lengths of the respective tasks can change, and this may reorder events across tasks. Consider a parallel program with two tasks, where task A reaches a synchronization point before task B. If the instrumentation overhead for task A exceeds that for task B, the order that the tasks reach the synchronization point will be reversed, and the recorded event order will differ from nominal.[2]

In general, some tasks are dynamically assigned to processors. Indeed, the Alliant FX/80 permits dynamic assignment of single loop iterations. If instrumentation changes task execution times by disproportional, data dependent amounts, the sequence of tasks executed by each processor, and the order of inter-task events cannot be predicted without knowledge of the task scheduling algorithm.

Given the diversity and complexity of possible instrumentation perturbations, both direct and indirect, software instrumentation must be designed to ameliorate or eliminate as many perturbations as possible. However, timing perturbations cannot be removed solely by efficient instrumentation, and perturbation analysis must be applied to resolve timing errors. An instrumentation design for the Alliant FX/80 is the subject of Section V. The following section develops a theory of time-based performance perturbation and constructs perturbation models for removing timing perturbations during sequential and concurrent execution.

IV. Performance Perturbation Models

Models to capture and remove timing 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 time-based perturbation models for trace instrumentation. Given an understanding of possible performance instrumentation perturbations (see Section III), measures of *in vitro* trace instrumentation costs (see Section V), and an instrumentation trace, our goal is to recover the "true" trace of events 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. Given the semantic difficulties of event analysis, we restrict our attention in this paper to two classes of timing analysis models; see [8], [11], and [9] for a discussion of event-based perturbation models. The first, simpler model predicts total execution time given trace data. The second adjusts the times of individual trace events. For both models, we discuss, where appropriate, the perturbations that might be removed by appropriate event analysis models. We begin, however, with a formal description of the instrumentation problem.

A. Definitions

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

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

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.

Ideally, we would like to use a trace of program events void of instrumentation intrusion for performance analysis. A *logical event trace*, τ, is a time-ordered sequence of events $e_1, \cdots, e_m$ representing a program's actual execution. Each e_i is of the form $\{t(e_i), eid_i\}$, where eid_i is the event identifier for the ith event indicating the statement S_{eid_i} in the program, and $t(e_i)$ is the time when the event occurred. If the program is instrumented, we use the notation τ_m to denote a *measured event trace*. Because a program can have both sequential and concurrent components, we define the logical sequential event trace, τ^s (τ^s_m for the measured sequential

[2]Different parallel execution orders can occur, even without instrumentation, due to asynchronous task operation. The reproducibility of parallel executions is another aspect of uncertainty separate from instrumentation. For purposes of our perturbation models in Section IV, we assume the differences between possible event orders from uninstrumented executions is minimal.

trace), as the subsequence of events $e_p, e_q, \cdots, e_r$ generated in sequential mode. Similarly, the logical concurrent event trace for processor i, τ^i (τ_m^i for the measured concurrent trace), is the subsequence of events $e_p^i, e_q^i, \cdots, e_r^i$ executed in concurrent mode on processor i.

Given these definitions and letting $T(S_{eid_i})$ be the *true* execution time of statement S_{eid_i}, the total execution time of a sequential program P is

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

The *measured* program execution time of a full instrumentation of P is

$$T_m^s(\mathcal{I}(P)) = \sum_{e_i \in \tau_m^s} [T(S_{eid_i}) \oplus T(I_{eid_i})]$$

where $T(I_{eid_i})$ is the true 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 (e.g., the disruption of memory reference patterns).

For concurrent execution time one must determine the critical path during concurrent computation. Let $\tau^s = e_p, e_q, \cdots, e_r$ represent the logical trace of sequential events and $\tau^{cp} = e_s, e_t, \cdots, e_u$ the logical trace of concurrent events along the critical path, respectively. The total true 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 a full instrumentation of P is

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

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

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

$$AE = T_m(\mathcal{I}(P)) - T(P), \tag{1}$$

and the relative error, RE, is

$$RE = \frac{T_m(\mathcal{I}(P)) - T(P)}{T(P)}. \tag{2}$$

The direct perturbation, DP,

$$DP = DP^s + DP^c \tag{3}$$

is the increased execution time directly caused by instrumentation; 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) and (2) estimate the instrumentation perturbation, they do 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,

$$T_a(P) = T_m(\mathcal{I}(P)) - DP. \tag{4}$$

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

Finally, the relative approximate error, RAE, estimates the accuracy of a timing analysis model (i.e., how accurately one can determine actual execution time from an instrumentation trace):[3]

$$RAE = \frac{T_a(P) - T(P)}{T(P)} = \frac{T_m(\mathcal{I}(P)) - DP - T(P)}{T(P)}. \tag{5}$$

B. Execution Time Analysis

The test of a timing analysis model's veracity is its ability to predict a program's actual execution time given an instrumentation trace. In the remainder of this section, we present execution time models for both sequential and concurrent execution.

1) Sequential Execution: During sequential execution, the principal perturbation is direct—execution of additional instrumentation instructions. This does not mean that indirect sources of perturbations, such as those discussed in Section II, do not exist. Rather, the execution time overhead is known to occur with every instrumentation execution, where the indirect perturbations are less likely and less deterministic. Furthermore, instrumentation does 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.[4]

[3] In the remainder of the paper we will use the measure $T_a(P)/(T(P) = RAE + 1$ to express the accuracy of the model.

[4] The approximation to $T(I_{eid_i})$, α, is given by the mean instrumentation time dilation; see Section V and Table II.

2) $DP = \sum_{e_i \in \tau^s} T(I_{eid_i}) = \alpha N$, where N is the number of instrumentation points.
3) $T_a(P) = T_m(\mathcal{I}(P)) - DP = T_m(\mathcal{I}(P)) - \alpha N$.
4) $RAE = \frac{(T_a(P) - T(P))}{T(P)} \simeq \frac{(T_m(\mathcal{I}(P)) - \alpha N)}{T(P)}$.

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; see Section V. To remove these indirect perturbations, we first apply the simple model above to approximate the actual execution time denoted by $T_a^1(P)$. This prediction reflects direct perturbations from instrumentation execution. We then assume the instrumentation code is removed from the instrumented program's assembly code (or object code) and measure the execution time of the resulting program, producing a second execution time estimate $T_a^2(P)$ that measures only the code perturbations. The final execution time approximation in the case of source code instrumentation is given by

$$T_a^{source}(P) \simeq T_a^1(P) * \frac{T(P)}{T_a^2(P)}. \tag{6}$$

The ideal approximation uses a nonuniform perturbation model (i.e., one that considers the effects of each individual instrumentation instance). However, the number of different cases to consider is huge and requires an analysis of the differences in the code generated with and without instrumentation. In light of these complications, the linear approximation above is reasonable when source code instrumentation is necessary.

2) Concurrent Execution: 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 [17]. Unfortunately, perturbation mechanisms such as those described in Section II-B 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 event analysis models, 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.[5] 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 $\tau^{\max}$.
3) $T_a(P) = T_m(\mathcal{I}(P)) - DP^c = T_m(\mathcal{I}(P)) - DP^s - \alpha N_{\max}$.
4) $RAE = \frac{(T_a(P) - T(P))}{T(P)} \simeq \frac{(T_m(\mathcal{I}(P)) - DP^s - \alpha N_{\max})}{T(P)}$.

In simpler terms, 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; see Fig. 2.

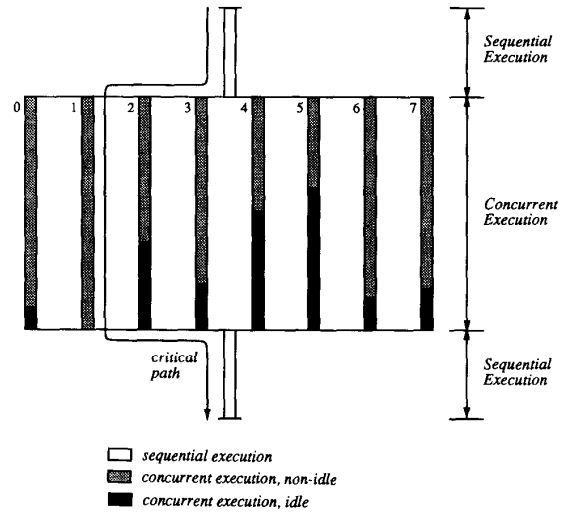


Fig. 2. Critical concurrent path selection.

C. Event Trace Timing Analysis

To recover the true sequence of trace event times, one must consider not only the total execution time but all possible inter-event dependencies and associated perturbations. Even given careful analysis and a predictive model, one cannot directly determine the accuracy of the predicted event times. If this were possible, event tracing would be unnecessary. Instead, one must infer the stability of the 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.

1) Sequential Trace Timing Analysis: 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 iteratively calculate each event time, given the perturbations of previous events.

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

$$t_a(e_i) = t_m(e_i) - (i-1)\alpha, \tag{7}$$

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.[6]

[5] If not, an event analysis model is needed; see [8], [11], and [9]. However, in Section VII, we show that timing analysis alone can yield significant insight in many practical cases.

[6] For example, if e_{11} is (854, 10) in the trace, the approximation of the actual time of occurrence would be $t_a(e_{11}) = 854 - 10.9 * 10 = 745$, if $\alpha = 10.9$.

Unfortunately, it is possible that two events e_i and e_j occur so close together that $t_m(e_i) \leq t_m(e_j)$ but $t_a(e_i) > t_a(e_j)$. Simply put, software instrumentation may be unable to separate closely spaced events because the hardware timer lacks the resolution to measure instrumentation overhead and elapsed time with sufficient accuracy. For example, on the Alliant FX/80, the timer resolution is 10 μs, but the machine cycle time is 170 ns. In a 10 μs window, several events may occur. Although, for sequential programs, these events form a total order, their times have a 10 μs uncertainty. As a consequence, we must assume simultaneity for all events whose estimated times differ by less than 10 μs.

2) Concurrent Trace Timing Analysis: 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 among concurrent threads of execution is required. To simplify analysis, we assume that events on separate concurrent threads are independent and that the program contains only a single level of fork-join concurrency, although multiple phases of sequential and concurrent computation are allowed. With the requisite event analysis model, these assumptions can be relaxed.

Given a trace τ^i of concurrent events $e_1^i, \cdots, e_n^i$ for each concurrent thread i, and a trace τ^s of sequential events $e_1^s, \cdots, e_m^s$, we approximate the actual time of a concurrent event e_k^i as follows.

1) If e_k^i is the first concurrent event after a sequential event e_j^s in the time ordered trace, then

$$t_a(e_k^i) = t_m(e_k^i) - t_m(e_j^s) + t_a(e_j^s).$$

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

2) If e_k^i immediately follows a concurrent event in the trace on thread i, then

$$t_a(e_k^i) = t_m(e_k^i) - t_m(e_j^s) + t_a(e_j^s) - \alpha c_k^i,$$

where c_k^i is the number of events in concurrent thread i after the last sequential event e_j^s 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_k^i, but the direct perturbation along thread i also is removed.

We approximate the actual time of a sequential event e_k^s as follows.

1) If e_k^s is the first sequential event in the trace after the last concurrent event from a concurrent phase of computation, then

$$t_a(e_k^s) = t_m(e_k^s) - t_m(e_j^i) + t_a(e_j^i)$$

where $t_a(e_j^i) > t_a(e_n^m)$ for all n and m such that e_i^j and e_n^m appear before e_k^s in the trace. It is here that we determine the critical concurrent path in the instrumented execution. The concurrent event appearing before e_k^s in the trace with the greatest approximated timestamp is used as the time basis to approximate the sequential event occurrence.

2) If e_k^s follows a sequential event in the trace, then

$$t_a(e_k^s) = t_m(e_k^s) - t_m(e_j^i) + t_a(e_j^i) - \alpha c_k^s$$

where c_k^s is the number of events that have occurred in sequential mode since the last approximated concurrent event e_j^i 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.

V. Performance Instrumentation

To validate our time-based perturbation models against the Alliant FX/80 execution environment, we constructed a performance instrumentation facility for the target machine. Below, we describe the performance instrumentation implementation for the FX/80 and its instrumentation overhead, both in execution time and memory bandwidth. Measures of *in vitro* instrumentation costs are necessary for practical application of the perturbation models.

For our experiments on the Alliant FX/80, we constructed a tracing library that includes static trace buffer declarations and a set of Assembly language tracing routines. The library maintains a trace buffer for each of the eight potentially concurrent threads of execution on the FX/80 and one additional buffer for events that occur during sequential execution. Each invocation of the tracing routine records a 32-bit timestamp, a 32-bit event identifier, the concurrency status, and the processor identifier in the appropriate trace buffer.

Trace instrumentation can be inserted at either source or assembly code levels. Although easier to automate, source instrumentation can create register allocation and access problems. These register management problems are in addition to those discussed in Section III-A. For example, the procedure calling convention on the Alliant FX/80 has the caller save registers. As a consequence, the invocation overhead for a source code trace event depends on the number of registers whose values must be retained across the instrumentation code boundaries. Because the caller does not know what registers are used by the tracing library, it must save all active registers. The magnitude of this register management overhead depends on the current register state. The alternative, saving registers in the callee, fixes the overhead—only the registers used by the trace library need be saved. The disadvantage is that registers may be saved unnecessarily.

Not only does source code instrumentation require register management, it may inhibit or force different code optimizations. The latter depend on both the language and the code generator. On the Alliant FX/80, the C compiler does not restructure the source code prior to code generation and optimization, nor does it generate vector or parallel code. Inspection of the generated assembly code after source instrumentation shows only simple optimizations; source state-

ment boundaries were clearly delimited. As an artifact of the C compiler's limited optimization, the overhead for source instrumentation was nearly invariant across instrumentation points.

Unlike source instrumentation in C, the overhead for a Fortran source instrumentation point is context dependent. The preprocessor of the Alliant Fortran compiler generates restructured Fortran source based on detected data dependencies. Although the resulting Fortran is functionally equivalent, statements can be reordered, loops can be fused or distributed, and new variables can be created [16]. Inserting trace instrumentation can inhibit a subset of these transformations.

As an example of the interaction of instrumentation and source restructuring, Table I shows the generated code for the following code drawn from the Livermore loops [12], both with and without source code instrumentation.

```
      DO 10 k=1, n
C        instrumentation point
         X(k) = Q + Y(k)*(R*ZX(k+10)
                + T*ZX(k+11))
C        instrumentation point
10    CONTINUE
```

When no source code instrumentation is included, the Alliant Fortran compiler generates code to pre-load the scalar operands (Q, R, and T) in floating point registers (`fp7`, `fp5`, and `fp6`, respectively) before the beginning of loop execution; these instructions are not shown in Table I. When instrumentation is added to the source code, the Fortran optimizer recognized that loading the registers during each loop iteration would eliminate the register save and restore overhead for the instrumentation call. Thus, the instrumented loop iterations fetch the scalar operands from memory. As a consequence, the first two floating point multiplications in the code without instrumentation become a sequence of two floating point register loads followed by two floating point multiplications in the instrumented code.

Given the perturbation variance for Fortran source instrumentation and the desire to postprocess the trace data using models of performance instrumentation with known costs, we opted to instrument all applications at the assembly code level. In this mode, the compiler generates code without knowledge of the instrumentation, the fixed instrumentation is inserted in the resulting code, and the code perturbation is context independent. We avoided the problems of variable costs for register save and restore by implementing an assembly code version of our trace library with the callee saves convention. With this instrumentation approach, an *a priori* measurement of instrumentation overhead is possible and, by hypothesis, *a posteriori* removal of perturbations via performance perturbation models.

To determine instrumentation overheads, we conducted a series of preliminary experiments that measured the *in vitro* costs of trace instrumentation. These costs provide the basis for the *in vivo* tests of our primary hypothesis, that instrumentation overheads can be removed from detailed performance traces. Among the possible perturbations discussed in Section III, the most significant are increased execution time (i.e., time dilation) and increased memory bandwidth. We began by measuring the cost to record an instrumentation trace event for both sequential and concurrent execution on the Alliant FX/80; see Table II.

TABLE I
FORTRAN PERTURBATIONS WITH SOURCE CODE INSTRUMENTATION

Expression	Normal	Instrumented
$a = R^*ZX(k+10)$	fmuls zx+10[d7],fp5,fp0	fmoves zx+10[d7],fp1 fmuls r,fp1,fp0
$b = T^*ZX(k+11)$	fmuls zx+11[d7],fp7,fp1	fmoves zx+11[d7],fp3 fmuls t,fp3,fp2
$c = a + b$	fadds fp1,fp0,fp0	fadds fp2,fp0,fp0
$d = c^*Y(k)$	fmuls y[d7],fp0,fp2	fmuls y[d7],fp0,fp4
$X(k) = Q + d$	fadds fp6,fp2,fp2 fmoves fp2,x[d7]	fadds q,fp4,fp5 fmoves fp5,x[d7]

TABLE II
INSTRUMENTATION TIME DILATION (MICROSECONDS)

Execution Mode	Level	Mean Time	Standard Deviation
Sequential	C Source	10.26	±1.75
Sequential	Assembly	11.01	±2.52
Concurrent	Assembly	12.25	±2.60

A test program performed one thousand calls to the tracing routine, and call overhead statistics were calculated from the trace data.

Table II shows instrumentation statistics for three forms of instrumentation. The smaller time for sequential source instrumentation is directly attributable to the C compiler's optimizations for register save and restore. Because the assembly code instrumentation always saves all registers, it has a slightly higher cost. With concurrent tracing, all eight CE's must write to different trace buffers. In this case, the total cache and memory traffic is greater and is distributed across a larger fraction of the address space. Because there are more cache misses and memory contention, the time dilation is greater.

In our implementation, each trace instrumentation point generates 48 bytes of memory traffic. In addition to the data placed in the trace buffer, other memory operations are needed to set up the subroutine stack, fetch the concurrency status, and read the trace buffer pointer. Using the execution time measurements above, sequential and concurrent tracing can generate at most 4.36 megabytes/second and 31.35 megabytes/second of memory traffic, respectively. Although these are well below the peak bandwidth of the Alliant FX/80 memory bus, the additional memory traffic is substantial, particularly for concurrent tracing. If the application program is memory intensive, the potential memory perturbations of Section II-A become real.

VI. EXECUTION TIME EXPERIMENTS

Our instrumentation hypothesis suggests that the perturbations attributable to detailed performance instrumentation can be minimized in sequential, vector, and parallel execution modes. To test this hypothesis, we conducted a series of instrumentation experiments to determine the magnitude of performance perturbations as a function of instrumentation frequency and execution mode (i.e., sequential, vector, and

TABLE III
EXPERIMENT CATEGORIES FOR THE LAWRENCE LIVERMORE LOOPS

Language	Mode	Type	Description
C	Sequential	Source	sequential C loops, instrumentation at source level
C	Sequential	NOP	sequential C loops, NOP instrumentation using $asm()$ construct
Fortran	Sequential	Source	sequential Fortran loops, instrumentation at source level
Fortran	Sequential	Null	sequential Fortran loops, instrumentation at source level but with instrumentation removed
Fortran	Sequential	Assembly	sequential Fortran loops, assembly level instrumentation
Fortran	Sequential	NOP	sequential Fortran loops, NOP instrumentation of assembly code
Fortran	Vector	Assembly	vector Fortran loops, assembly level instrumentation
Fortran	Vector	NOP	vector Fortran loops, NOP instrumentation of assembly code
Fortran	Concurrent	Assembly	concurrent Fortran loops, assembly level instrumentaion
Fortran	Vector Concurrent	Assembly	vector-concurrent Fortran loops, assembly level instrumentation

parallel on the Alliant FX/80).

All experiments used C and Fortran versions of the Lawrence Livermore loops (LLL) [12], a set of 24 loops often used to benchmark high-performance computer systems. These loops contain a diversity of language constructs, yet remain small enough to permit detailed analysis of performance perturbations. Table III summarizes the combinations of language and instrumentation used in our experiments.

We emphasize that the purpose of our analysis was *not* to characterize the performance of the Livermore loops. Instead, the Livermore loops constitute a set of test cases for our performance perturbation models. Successful prediction of loop execution times and recreation of event times would validate our models. Furthermore, it would suggest that we can confidently apply perturbation analysis to larger applications where performance characterization is important.[7]

For each Livermore loop, two experiments were conducted in each category; see Fig. 3. In the first experiment, trace instrumentation was placed at the beginning and the end of the loop to determine total loop execution time, the so-called *raw* instrumentation. The second experiment produced traces from a *full* instrumentation with trace events for each source statement.[8] For a typical loop, this instrumentation generated over 2000 trace events, with a total instrumented execution time of less than 0.1 s. Despite this density of trace instrumentation, our perturbation models often recover actual execution time with less than 10% error, confirming our hypothesis that detailed performance data need not be incompatible with accurate measurement.

In the remainder of this section, we describe the result of an execution time analysis of the experimental results for C, Fortran, vector Fortran, concurrent Fortran, and vector-concurrent Fortran. In each case, we first analyze the the direct perturbations caused by the performance instrumentation using the simple models of Section IV.

We then explain any deviations from the models by investigating sources of indirect perturbation (e.g., changes in the generated code).

[7] In [10], we discuss the use of perturbation analysis in the performance measurement of application codes on the Cray X-MP and Cray 2 systems.

[8] A performance analyst would probably not chose to instrument at the level of every source statement. However, because we are interested in determining where the perturbation analysis fails with respect to instrumentation frequency, the full instrumentation experiments are used as a stress test for the models. In practice, reducing instrumentation detail will decrease the severity of the perturbations and, hence, will result in more accurate approximations when the perturbation models are applied.

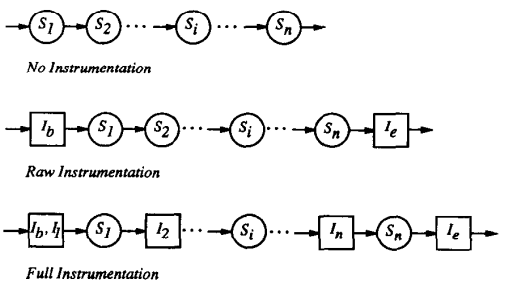


Fig. 3. Instrumentation alternatives.

A. Sequential C Experiments

The goal of our sequential C experiments was to compare source level instrumentation across languages. As mentioned earlier, the Alliant C compiler's optimizations are but a subset of those performed by the Fortran compiler, and we conjectured that source instrumentation in C was less susceptible to indirect performance perturbations. Fig. 4 shows the results of our experiments with complete source instrumentation for some of the loops. Fig. 4 shows the outlier results from the set of sequential C experiments. Loop results not shown fall within this range presented. In the figure, the black bars represent the ratio of full instrumentation execution time to the raw instrumentation execution time. The dotted bars represent the ratio of the predicted execution time, using the model of Section IV-B1, to the raw execution time for the same loop. Clearly, the simple perturbation model predicts the major sources of perturbation and accurately predicts actual execution time. This result is true for all of the loops.

Although the simple perturbation model bounds the possible indirect perturbations of source code instrumentation, it cannot quantify these effects. However, replacing the C instrumentation statements with NOPs, retains the perturbations of code generation but removes all ancillary perturbations of instrumentation (e.g., memory referencing). The direct effects of instrumentation, assumed to be constant in our model, are emulated by the fixed execution time of the NOPs. Thus, the difference between the real source instrumentation approximation and the approximation from the NOP instrumentation are largely attributable to the ancillary perturbations. For the loops in Fig. 4, Fig. 5 shows the result of predicting performance with NOP instrumentation at every source statement. All predictions are within 3%. This small error is attributable solely to

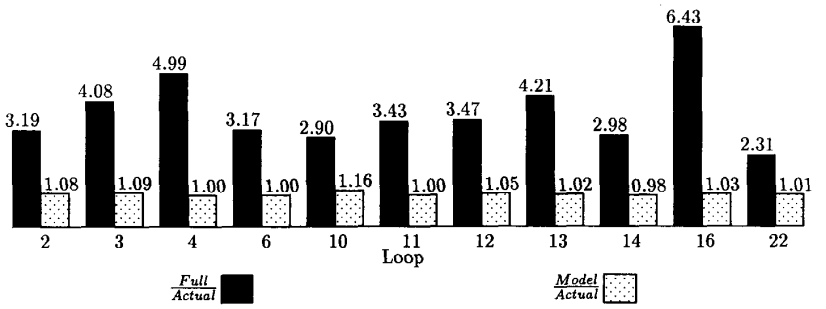


Fig. 4. Source instrumentation of C loops.

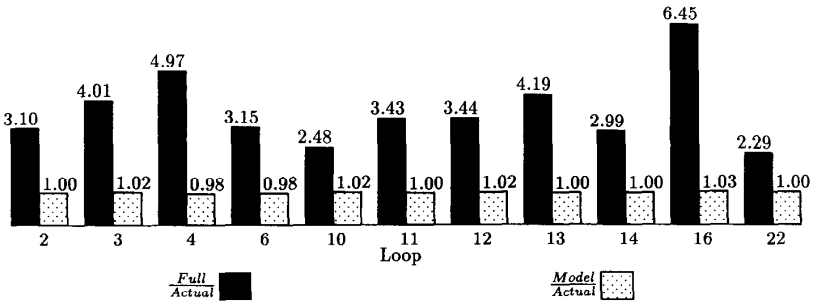


Fig. 5. NOP source instrumentation of C loops.

the indirect perturbations of source code instrumentation. This supports our claim that the perturbations of code generation are small; a study of the assembly code produced from the C source instrumentation also shows that the changes are minimal. As an example, the model's approximation for loop 10 improves from 1.16 to 1.02. This suggests that there are perturbations different from instrumentation execution time, which the C-NOP experiments model, accounting for the slowdown of the trace-instrumented run. Upon inspection of the generated assembly code one sees that a register dependency is introduced by the tracing instrumentation that is not present in the code without instrumentation and the NOP instrumented testcases. This dependency stalls the instruction following the instrumentation by three cycles on the Alliant FX/8 or 510 ns. If we modify the simple sequential model to include this stalling, the approximation of the instrumented execution improves to 1.09 (within 9% of actual execution time).

B. Sequential Fortran Experiments

Unlike source level instrumentation in C, the breadth of the Alliant Fortran compiler's optimizations creates large perturbations with source instrumentation. To quantify these perturbations and to determine an acceptable instrumentation methodology, we conducted experiments using four combinations of source and assembly code instrumentation; see Table III.

Fig. 6 shows the simplest test, complete instrumentation at the source code level. Unlike the comparable C experiment in Fig. 4, the Fortran perturbations cannot be explained by our simple perturbation model, The model's approximations differ as much as 80% from the actual execution times. Clearly, some perturbations are indirect.

To determine the indirect perturbations, including changes to the generated code, we compiled each loop with source instrumentation inserted. We then removed the generated instrumentation from the assembly code. This *instrumentation* retains all perturbations of code generation without the direct perturbations that accrue from execution of instrumentation. As Fig. 7 shows, these indirect perturbations are substantial; inserting source instrumentation inhibits many code optimizations. Do the perturbations of code generation account for a significant fraction of all perturbations? The white bars in Fig. 7 show the approximations of the model of Section IV-B1 when (6) is applied; that is, the code generation perturbations have been removed from the original approximations. As can be seen, the model predicts well for some loops. However, significant perturbations remain, and one must consider additional perturbations to fully explain the instrumentation's effect on loop performance in all cases.

The complex interactions of source instrumentation and the Fortran compiler make isolation of instrumentation perturbations difficult, if not impossible. Because our goal is the systematic application of a standard perturbation model, the remainder of our experiments were conducted by instrumenting the generated assembly code of each loop. With suitable modifications, a compiler could generate this instrumentation after all optimization, eliminating indirect perturbations during code generation.

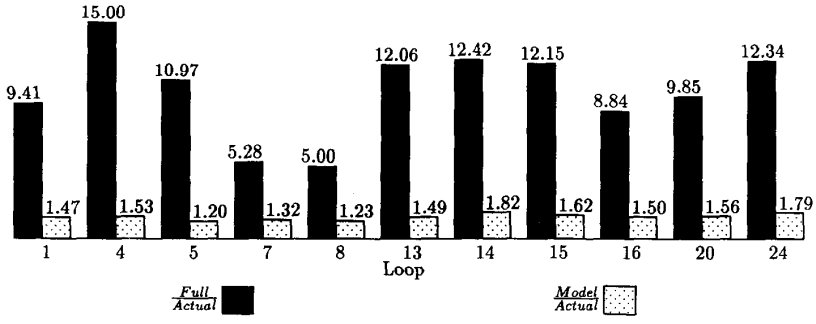


Fig. 6. Source instrumentation of Fortran loops.

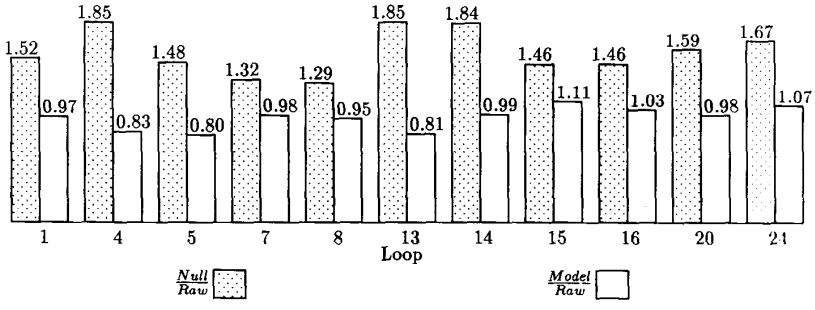


Fig. 7. Source instrumentation of Fortran loops with instrumentation removed.

Fig. 8 shows the merit of Fortran assembly instrumentation, and, indirectly, the need for compiler supported performance instrumentation. If code generation is not perturbed, mechanical application of our simple perturbation model permits recovery of total execution time with small error, typically less than 5%. For those Livermore loops where this approach fails, the error never exceeds forty percent. For example, for loop 4 (not shown) $Full/Raw = 14.22$ and $Model/Raw = 0.73$, and for loop 15 (not shown) $Full/Raw = 13.36$ and $Model/Raw = 1.35$. In each case, there is an extraordinary ratio of instrumentation to application code (e.g., instrumentation of a single machine instruction). Often, accounting for a single, 170 ns cycle register stall in the instrumented code reduces the error to less than 5%. It is also possible that the instrumentation causes a register dependency, present in the code without instrumentation, to be eliminated. Single cycle instrumentation typically requires hardware support; even in these stress tests, the accuracy of the software model is surprising.

C. Vectorized Fortran Experiments

Instrumenting the iterations of a sequential loop produces a sequence of trace events for each loop iteration. If the same loop is vectorized and the loop bound is less than the vector register length, a single machine instruction may represent the entire computation. Thus, a vectorized loop contains fewer potential instrumentation points. For example, if the vector loop bound exceeds the vector register length, the loop can be instrumented only at those points where register reloads occur.

TABLE IV
FORTRAN TRACE EVENT COUNTS

Livermore Loop	Scalar Events	Vector Events	Livermore Loop	Scalar Events	Vector Events
1	2006	68	12	2004	68
3	2007	70	13	1092	43
4	1820	68	14	17024	3424
6	4162	381	15	10648	1040
7	1994	68	18	4498	203
8	1395	75	21	32554	2554
9	206	12			

Table IV shows those Livermore loops that the Alliant Fortran compiler vectorized and the number of associated instrumentation trace events. The number of vector events depends on the vector length and loop complexity.

Because a vector instruction performs many identical operations, it represents a larger computational granule than a simple scalar instruction. Consequently, the relative perturbation for instrumented vector instructions is less than that for scalar instructions, and perturbation models should more accurately predict execution time. Fig. 9 shows precisely this result. Due to space restrictions, loop 8 ($Full/Actual = 1.62$ and $Model/Actual = 1.00$) is not shown.

D. Concurrent Fortran Experiments

Unlike vectorization, compiling for concurrency does not condense multiple statements; all observable events in a sequential trace remain present, but a subset of the events execute concurrently. Moreover, tracing instrumentation may dominate

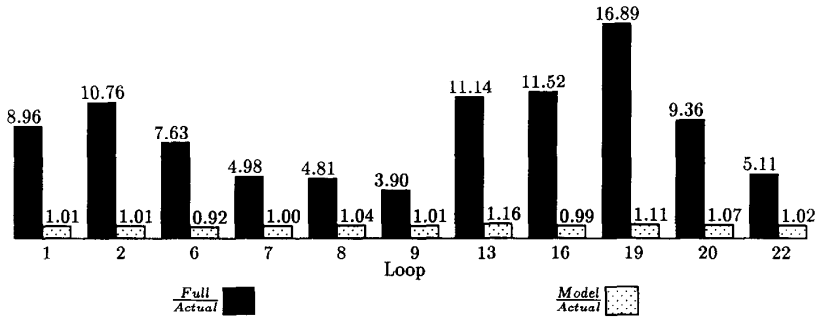


Fig. 8. Assembly code instrumentation of Fortran loops.

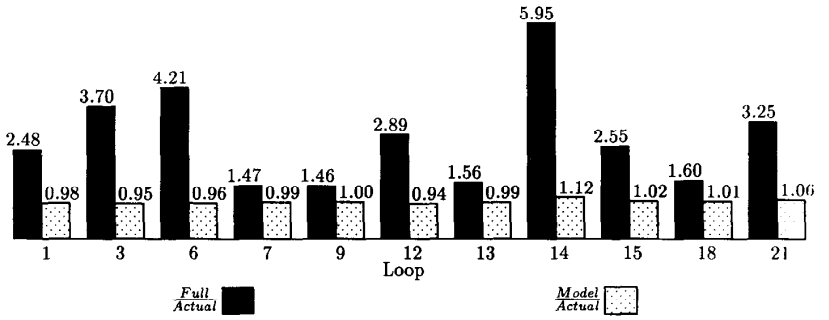


Fig. 9. Assembly code instrumentation of vectorized Fortran loops.

concurrent execution of some loops (i.e., a significant fraction of concurrent time may be spent in instrumentation code). In these cases, critical paths must be identified with care. Recall that the concurrent model of Section IV-B2 makes restrictive assumptions about execution behavior.

Fig. 10 shows the result of our concurrent perturbation model for those loops concurrentized by the Alliant Fortran compiler. Loop 8 ($Full/Actual = 3.67$ and $Model/Actual = 1.32$) is not shown.

Clearly, the concurrent perturbation model yields approximations that are both high and low. For most loops, the errors can be explained via a combination of register dependencies and increased memory traffic. For instance, by applying concurrent trace instrumentation overheads in the timing analysis for those loops where tracing dominates execution (8, 12, 13, 14, and 18) we were able to reduce errors to within 10%. For other codes, in particular loops 3, 4, and 17, the concurrent execution model assumed by the timing model is violated due to data dependent behavior. Because the timing model does not include critical path analysis, substantial approximation errors occur in these codes. We consider loops 3 and 17 in more detail to explain why these errors occur.

For Livermore loop 3 (Inner Product), our execution time model overestimates the instrumentation perturbation (i.e., the model's estimate of total execution time is too low). Because the perturbation model charges only for direct perturbations, one infers that the instrumentation *reduces* the execution time of the application code. For loop 3, the Alliant Fortran compiler creates a critical section around the update of the inner product sum. Without instrumentation, most processors are blocked on entry to the loop's 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 the code is instrumented; subtracting a fixed overhead underestimates the total execution time. Interestingly, this analysis also applies in the case of loop 4 (Banded Linear Equations).

For loop 17 (Implicit, Conditional Computation), our model underestimates the instrumentation perturbation. Surprisingly, the reason is the same as that for loop 3—the loop contains a critical section. Unlike loop 3, however, the trace instrumentation lies mostly inside the critical section. This increases the probability of contention, and the critical section becomes a larger fraction of the total execution time in the instrumented code. Subtracting a fixed overhead overestimates the total execution time.

The concurrency experiments reveal both the importance of instrumentation placement and the need for accurate critical path analysis. The latter permits identification and removal of indirect perturbations (e.g., synchronization stalls). This is considered in detail in [8], [11], and [9].

E. Vector Concurrent Fortran Experiments

Vector concurrent execution mode combines the perturbations of both vectorization and concurrentization. Fortuitously, the respective perturbations are not additive. Vectorization damps perturbation by increasing computation granules while reducing the number of instrumentation events. Concurrency,

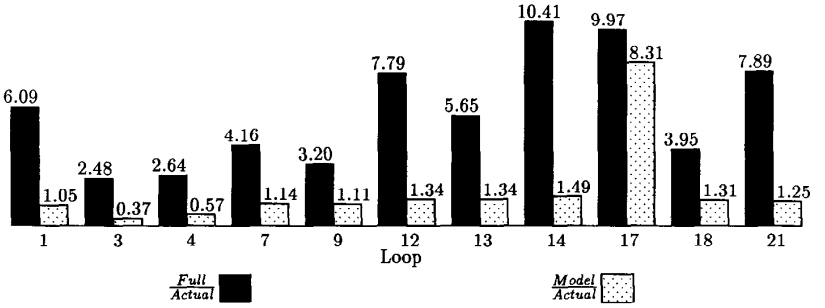


Fig. 10. Assembly code instrumentation of concurrent Fortran loops.

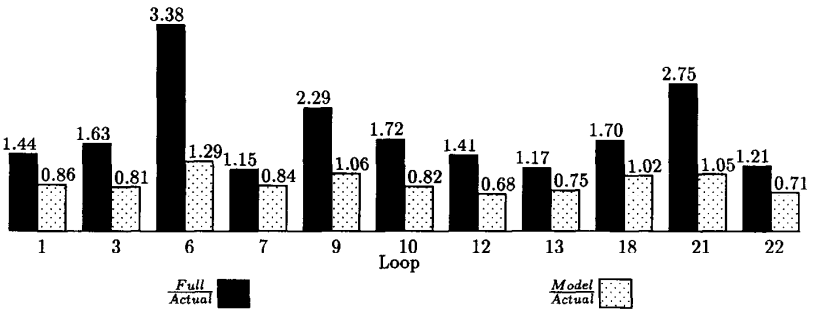


Fig. 11. Assembly code instrumentation of vector concurrent Fortran loops.

however, reduces the execution time and increases the stress on the memory hierarchy.

Fig. 11 shows the result of our experiments for those loops that executed in vector concurrent mode on the Alliant FX/80. Loop 8 ($Full/Actual = 1.92$ and $Model/Actual = 1.04$) is not shown.

In contrast to Fig. 8 or even Figs. 9 or 10, the ratio of instrumented to actual execution time is small; each processor records a smaller number of events. With one exception, our perturbation model consistently underestimates total execution time. As discussed below, we conjecture that the underlying reason is the same for each loop.

Most loops contain sequences of vector memory operations, separated by instrumentation. Moreover, earlier studies have shown that the Alliant FX/80 is susceptible to cache bank contention in vector concurrent mode [1], [5]. Although the trace instrumentation also references the cache, the request rate is modest compared to that for vector instruction sequences. We conjecture that tracing changes the temporal distribution of memory references, reducing the number of cache and memory bank conflicts. This, in turn, reduces the total execution time.

With rare exception, our performance instrumentation models accurately approximate total execution time for the Lawrence Livermore loops when run in sequential, vector, concurrent, and vector-concurrent modes with full instrumentation on the Alliant FX/80. In those instances where the models are inaccurate, most inaccuracies can be explained by simple analysis of the instrumented code. This suggests that simple perturbation timing models, coupled with event trace models, would permit detailed, but accurate, performance tracing.

VII. Event Trace Analysis

The experiments described in Section VI evaluated the feasibility of predicting total execution time in the presence of massive instrumentation. The results show that global performance measures, such as total execution time, are computable to a acceptable accuracy via application of performance perturbation models. Clearly, one need not instrument every source language statement to determine total execution time; simpler, less intrusive methods exist. The obvious motivation of this approach is to obtain additional performance data for more detailed performance analysis. However, the benefit of this trace data depends on its accuracy (i.e., how well it reflects actual event times and orders). More detailed measurement will introduce more perturbations that the models must resolve to maintain an acceptable level of accuracy in performance approximations.

To determine the accuracy of trace data, one needs a standard of reference. Without a passive hardware monitor to capture and record events, no such standard exists. Instead, one must compare a sequence of event traces, each produced with successively smaller subsets of the complete trace instrumentation. As the number of trace events decreases, the presumed accuracy of the event times and orders increases.

From the 24 Livermore loops, we selected two loops for detailed study. The first, loop two, executes in sequential mode; the second, loop eight, executes in sequential, vector, concurrent, and vector-concurrent modes. For each loop, we created a trace by instrumenting each source language statement, the *full* trace. In addition, we generated traces using two partial instrumentations, each a successively smaller subset

TABLE V
EVENT TIME DIFFERENCES FOR LOOP TWO

Reference			Analyzed			Total Delta μs	Mean Delta μs	Percent Delta
Trace	Events	Time μs	Trace	Events	Time μs			
Full	333	383.4	Partial-1	236	378.7	933.4	3.96	1.04
Full	333	383.4	Partial-2	139	380.6	402.8	2.90	0.76
Partial-1	236	378.7	Partial-2	139	380.6	429.3	3.09	0.81

```
          call trace_event(0)
      II = 101
          call trace_event(1)
      IPNTP = 0
          call trace_event(2)
222   CONTINUE
          call trace_event(3)
      IPNT = IPNTP
          call trace_event(4)
      IPNTP = IPNTP + II
          call trace_event(5)
      II = II/2
          call trace_event(6)
      I = IPNTP
          call trace_event(7)
      DO 2 K = IPNT+2, IPNTP ,2
            call trace_event(8)
         I = I+1
            call trace_event(9)
         X(I) = X(K) - V(K)*X(K-1) - V(K+1)*X(K+1)
            call trace_event(10)
2     CONTINUE
            call trace_event(11)
      IF(II.GT.1) GO TO 222
            call trace_event(12)
```

Fig. 12. Instrumented Livermore loop two.

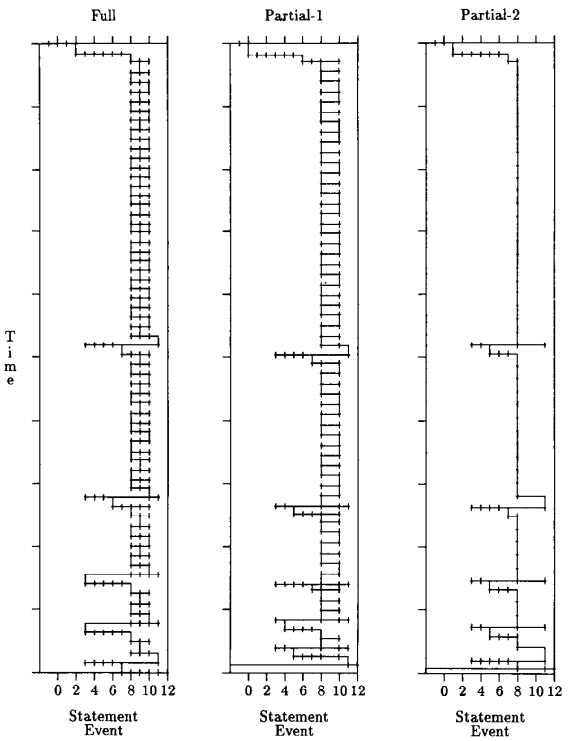


Fig. 13. Sequential execution of loop two.

of the complete instrumentation, traces *partial-1* and *partial-2*, respectively. After applying the trace perturbation model to each trace, we used Gantt charts [7] to verify qualitative agreement; comparison of the mean percent difference between event times confirmed quantitative agreement.

A. Loop Two—Incomplete Cholesky Conjugate Gradient

Livermore loop two, shown in Fig. 12, is an excerpt from an incomplete Cholesky conjugate gradient code that executes in sequential mode. Fig. 13 shows the Gantt charts of traces from three levels of instrumentation.[9]

Events 3 and 11 mark the beginning and end, respectively of each outer loop iteration. Traces *full* and *partial-1* also show the inner loop iterations, marked by events 8 and 10, respectively.

Although the three traces agree qualitatively, there are small quantitative differences. The total execution times, as predicted by the trace perturbation models, do differ by a small amount, but not greater than 1.25%. To verify event times, we correlated trace events and compared their times for three combinations of a *reference* trace, which contains the larger number of events, and an *analyzed* trace, which contains a subset of the events in the reference trace. The sum of the absolute differences between matched event times is the total event delta. Dividing this number by the number of events in the analyzed trace yields the mean event delta. Finally, dividing the mean event delta by the number of events yields the percent event delta. Table V shows the result of this analysis for loop two.

The mean difference between corresponding event times is less than 2% of the total execution time, or 8 μs. This is an excellent match, given the Alliant FX/80 timer resolution of 10 μs.

B. Loop Eight—ADI Integration

Loop eight, an ADI integration, can be both vectorized and parallelized. This permits evaluation of trace perturbation for all the Alliant's execution modes. Figs. 14 and 15 show the Gantt charts for sequential and vector mode, respectively.

Although the number of sequential events makes visual comparison difficult, the temporal event correlation for vector

[9] In Figs. 13 and 15, individual events are marked by the symbol +; the display scale for other Gantt charts does not permit display of individual events. In the figures, event simultaneity is a consequence of limited instrumentation clock resolution; see Section V.

TABLE VI
EVENT TIME DIFFERENCES FOR LOOP EIGHT, SEQUENTIAL EXECUTION

Reference			Analyzed			Total Delta	Mean Delta	Percent Delta
Trace	Events	Time μs	Trace	Events	Time μs	μs	μs	
Full	1395	4100.0	Partial-1	801	4051.8	22415.1	27.98	0.69
Full	1395	4100.0	Partial-2	402	3990.0	24038.8	59.80	1.50
Partial-1	801	4051.8	Partial-2	402	3990.0	12992.1	32.32	0.81

TABLE VII
EVENT TIME DIFFERENCES FOR LOOP EIGHT, VECTOR EXECUTION

Reference			Analyzed			Total Delta	Mean Delta	Percent Delta
Trace	Events	Time μs	Trace	Events	Time μs	μs	μs	
Full	75	1285.2	Partial-1	59	1280.5	174.9	2.96	0.23
Full	75	1285.2	Partial-2	38	1278.5	238.3	6.27	0.49
Partial-1	59	1280.5	Partial-2	38	1278.5	226.9	5.97	0.47

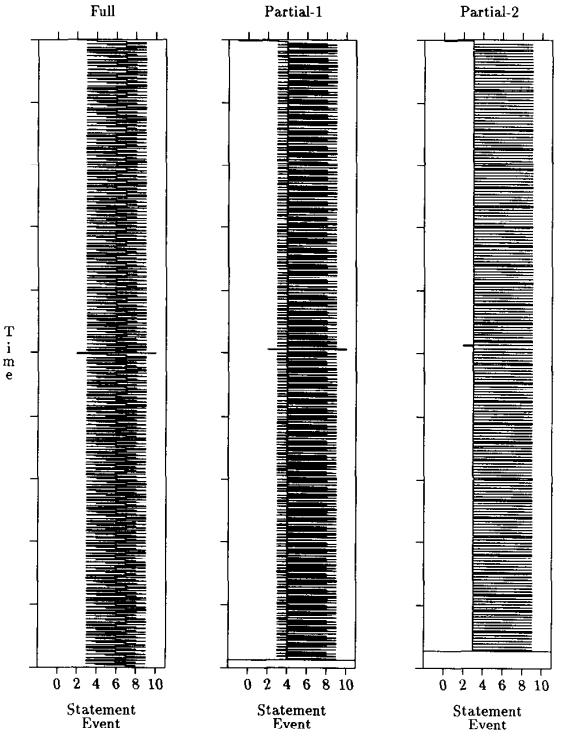


Fig. 14. Sequential execution of loop eight

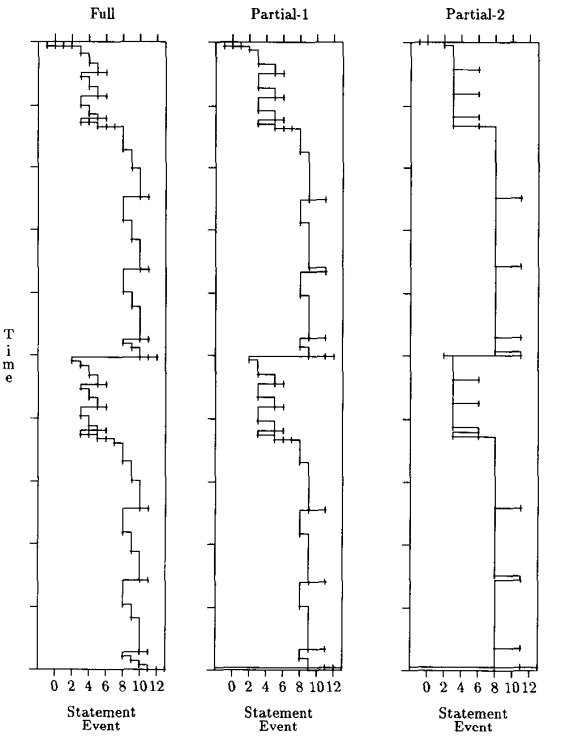


Fig. 15. Vector execution of Loop eight.

mode is striking. This is a direct consequent of the high accuracy of the total execution time approximations.

The accuracy of the event times, as predicted by the trace perturbation model for sequential and vector execution, is confirmed by Tables VI and VII. In sequential execution mode, event timing differences, although as much as 60 μs on average, are less than 1.5% of the total execution time. The discrepancies for vector mode are even smaller—the mean difference is less than the timing resolution, and the percent difference is less than 0.5%.

Fig. 16 shows the Gantt charts for the sequential thread and all concurrent threads from the concurrent execution of Livermore loop eight on six processors.[10]

In this code, there are two concurrent loops of equal complexity, separated by a sequential operation. During concurrent computation, each thread receives roughly the same amount of work. These execution behavior characteristics are similar for each set of traces.

Although there are qualitative similarities between the traces of Fig. 16, the differences in predicted total execution time suggest substantial error in individual event times. Indeed, the execution time difference between the *full* and *partial-2* trace

[10] We used only six Alliant CE's in these experiments to reduce the visual complexity of the figures.

TABLE VIII
EVENT TIME DIFFERENCES FOR LOOP EIGHT, CONCURRENT EXECUTION

Reference Trace			Analyzed Trace			Total Delta	Mean Delta	Percent Delta
Thread	Events	Time μs	Thread	Events	Time μs	μs	μs	
Full			Partial-1					
Seq	9	1126.6	Seq	9	1052.1	355.3	39.48	3.75
0	238	1020.3	0	136	989.4	2926.1	21.51	2.17
1	231	1030.3	1	132	989.4	4930.2	37.35	3.78
2	238	1030.3	2	136	989.4	3272.4	24.06	2.43
3	224	1046.6	3	128	963.0	6294.3	49.17	5.11
4	231	1046.6	4	132	963.0	4890.5	37.05	3.85
5	224	1077.5	5	128	1013.0	5631.7	44.00	4.34
Full			Partial-2					
Seq	9	1126.6	Seq	6	952.4	444.6	74.10	7.78
0	238	1020.3	0	68	910.6	4045.4	59.49	6.53
1	231	1030.3	1	66	910.6	5304.8	80.38	8.8
2	238	1030.3	2	68	910.6	4240.9	62.37	6.85
3	224	1046.6	3	64	892.4	4453.3	69.58	7.80
4	231	1046.6	4	66	892.4	4937.9	74.82	8.38
5	224	1077.5	5	64	922.4	5590.1	87.35	9.47
Partial-1			Partial-1					
Seq	9	1052.1	Seq	6	952.4	267.9	44.65	4.69
0	136	989.4	0	68	910.6	2664.8	39.19	4.30
1	132	989.4	1	66	910.6	2859.6	43.33	4.76
2	136	989.4	2	68	910.6	2710.3	39.86	4.38
3	128	963.0	3	64	892.4	3400.2	53.13	5.95
4	132	963.0	4	66	892.4	2535.4	38.42	4.30
5	128	1013.0	5	64	922.4	2762.1	43.16	4.68

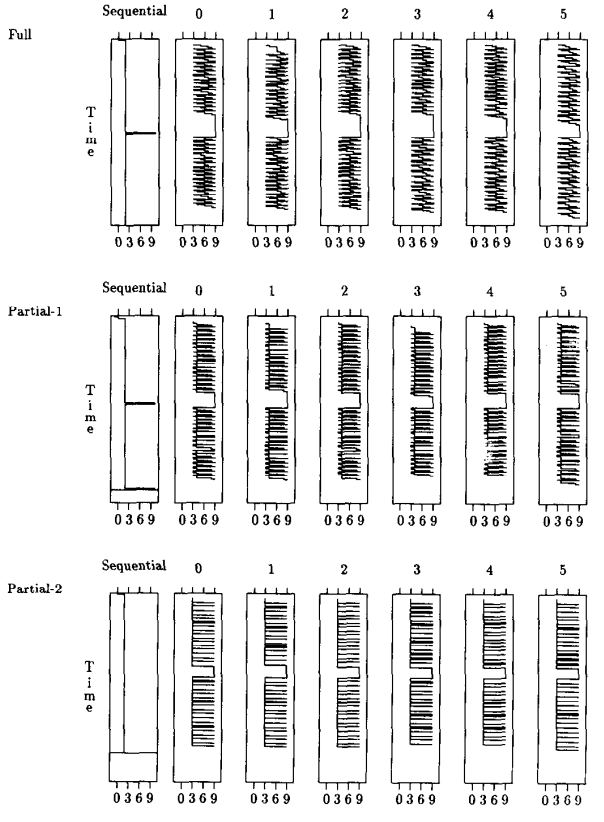


Fig. 16. Concurrent execution of Loop eight.

is 18%; see Table VIII. Surprisingly, however, the percent event delta between the *full* and *partial-1* traces is less than 6%; the same is true for the comparison of traces *partial-1* and *partial-2*. As the difference between the number of events in the reference trace and those in the analyzed trace increases (e.g., in the comparison of *full* and *partial-2*), the percent event delta rises. However, even in the worst case it does not exceed 10% of the total execution time. Stated another way, the average uncertainty between two matched events in the *full* and *partial-2* traces is less than 10%.

Finally, Fig. 17 shows the Gantt charts for vector-concurrent execution mode. Vectorization reduces the number of possible instrumentation points and the total number of trace events, and the predicted execution times differ by less than 3%. Because the two loops in the code are statically scheduled in vector-concurrent mode, unlike the dynamic schedule in concurrent mode, the execution behavior across processors should not differ. In Fig. 17, the execution signatures are indistinguishable. Moreover, Table IX shows that the percent delta in event times is less that 5%. More importantly, the mean differences are at the limits of the timer resolution.

VIII. CONCLUSIONS

The foundation of computer system performance analysis is measurement and experimentation. 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. Regrettably, there are no formal models of performance perturbation that would permit quantitative evaluation from instrumentation costs, measured event frequency, and desired instrumentation detail. Given the lack of models and the potential dangers of excessive instrumentation, detailed software event traces often are rejected for fear of corrupting

TABLE IX
EVENT TIME DIFFERENCES FOR LOOP EIGHT, VECTOR-CONCURRENT EXECUTION

Reference Trace			Analyzed Trace			Total Delta	Mean Delta	Percent Delta
Thread	Events	Time μs	Thread	Events	Time μs	μs	μs	
Full			Partial-1					
Seq	10	465.4	Seq	10	470.2	79.0	7.90	1.68
0	22	446.4	0	18	452.0	170.3	9.46	2.09
1	22	446.4	1	18	451.1	198.3	11.01	2.44
2	22	446.4	2	18	452.0	170.3	9.46	2.01
3	22	446.4	3	18	451.1	308.2	17.12	3.80
4	22	446.4	4	18	451.1	198.2	11.01	2.44
5	22	446.4	5	18	452.0	189.5	10.53	2.33
Full			Partial-2					
Seq	10	465.5	Seq	6	460.2	23.4	3.91	0.85
0	22	446.4	0	12	429.7	101.0	8.42	8.42
1	22	446.4	1	12	440.2	84.1	7.01	1.59
2	22	446.4	2	12	429.7	105.1	8.76	2.04
3	22	446.4	3	12	440.2	80.1	6.67	1.52
4	22	446.4	4	12	429.7	101.0	8.42	1.96
5	22	446.4	5	12	439.2	96.8	8.07	1.84
Partial-1			Partial-2					
Seq	10	470.2	Seq	6	460.2	63.5	10.58	2.30
0	18	452.0	0	68	429.7	195.7	16.31	3.79
1	18	451.1	1	66	440.2	182.4	15.20	3.45
2	18	452.0	2	68	429.7	184.3	15.35	3.57
3	18	451.1	3	64	440.2	253.8	21.15	4.81
4	18	451.1	4	66	429.7	204.8	17.06	3.97
5	18	452.0	5	64	439.2	204.2	17.02	3.87

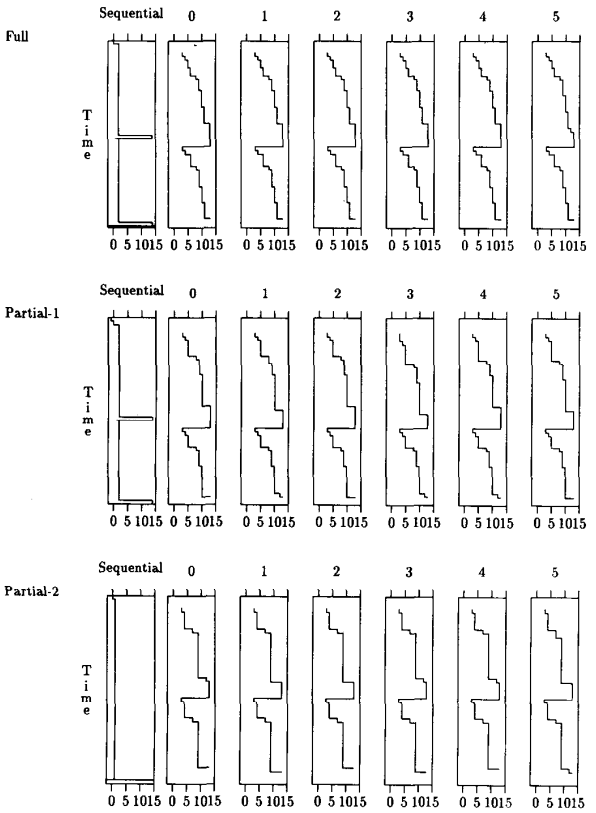


Fig. 17. Vector-concurrent execution of loop eight.

the data (i.e., a small volume of accurate, though incomplete, instrumentation data is preferred).

We hypothesized that current restrictions on the volume of performance data were unduly pessimistic. To test this hypothesis, we developed a series of simple perturbation models that approximate trace event times from instrumented execution. Using these models, we conducted a series of instrumentation experiments to determine the magnitude of performance perturbations as a function of instrumentation frequency and execution mode, and the accuracy of the performance approximations.

The experiments discussed in Section VI and Section VII are stress tests for the time-based performance perturbation models. The ability to approximate actual code execution times to within 15% from full trace instrumentations, with execution time perturbations exceeding four orders of magnitude, is remarkable, especially for such relatively simple models. Even for those Livermore Loops that are not well approximated, often minor adjustments in the models to account for register interlock stalls or increased memory reference density due to tracing operations can account for the error. Not only do the models perform well when approximating global performance measures, but individual event times are computable to acceptable accuracy, even in the presence of massive trace instrumentation.

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 (scalar and vector) execution because the execution states of sequential programs form a total order, and event times are affected only by instrumentation overhead. Even for some concurrent execution scenarios, typically those with fork-join behavior and no inter-thread dependencies, the time-based perturbation models are 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. Under the timing model assumptions of event independence, approximated event timings for concurrent execution can also violate the required partial order. Furthermore, critical performance phenomena such as synchronization behavior cannot be accurately modeled using timing information alone. Clearly, concurrent perturbation analysis necessitates a model of event dependencies and instrumentation. In [8], [11], and [9], we describe models for event-based perturbation analysis that use the measurement and subsequent analysis of synchronization operations to resolve perturbation effects in cases where there are execution event dependencies.

Although there remain fundamental limits on the attainable volume of accurate performance data, we believe further development of trace perturbation models will permit acquisition of more data than traditionally believed possible.

REFERENCES

[1] W. Abu-Sufah and A. Malony, "Vector processing on the Alliant FX/8 multiprocessor," in *Proc. 1986 Int. Conf. Parallel Processing*, Aug. 1986, pp. 559–566.

[2] R. Arlauskas, "iPSC/2 system: A second generation hypercube," in *Proc. Third Conf. Hypercube Concurrent Comput. Appl., Vol. I*, Pasadena, CA, ACM, Jan. 1988, pp. 38–42.

[3] S. Chen, "Large-scale and high-speed multiprocessor system for scientific applications: Cray X-MP-2 series," in *Proc. NATO Workshop High Speed Computat.* J. Kowalik, Ed., Springer-Verlag, 1984, pp. 59–67.

[4] D. Ferrari, "Considerations on the insularity of performance evaluation," *IEEE Trans. Software Eng.*, June 1986, pp. 678–683.

[5] K. Gallivan, D. Gannon, W. Jalby, A. Malony, and H. Wijshoff, "Behavioral characterization of multiprocessor memory systems: A case study," in *Proc. 1989 ACM SIGMETRICS Conf. Measurement Modeling Comput. Syst.*, Berkeley, CA, May 1989, pp. 79–88.

[6] D. Gannon, W. Jalby, and K. Gallivan, "Strategies for cache and local memory management by global program transformation," in *Proc. 1987 Int. Conf. Supercomput.*, Athens, Greece, ACM, 1987.

[7] H. Gantt, "Organizing for work," *Indust. Management 58*, pp. 89–93, Aug. 1919.

[8] A. Malony, "Performance observability," Ph.D. dissertation, Dep. Comput. Sci., Univ. of Illinois at Urbana-Champaign, Sept. 1990.

[9] ——, "Event based performance perturbation: A case study," in *Proc. Third ACM SIGPLAN Symp. Principles Practice Parallel Programming*, Apr. 1991.

[10] A. Malony, J. Larson, and D. Reed, "Tracing application program execution on the Cray X-MP and Cray 2," in *Proc. 1990 Supercomput. Conf.*, Nov. 1990, pp. 60–73.

[11] A. Malony and D. Reed, "Models for performance perturbation analysis," in *Proc. Workshop Parallel Distributed Debugging*, ACM Sigplan/Sigops and Office of Naval Research, May 1991.

[12] F. McMahon, "The Livermore Fortran kernels: A computer test of the numerical performance range," Tech. Rep. UCRL-53745, Lawrence Livermore National Laboratory, Dec. 1986.

[13] R. Perron and C. Mundie, "The architecture of the Alliant FX/8 computer," in *Proc. Spring COMPCON '86*, Mar. 1986, pp. 390–393.

[14] A. Smith, "Cache memories," *ACM Comput. Surveys 14*, vol. 3, 473–530, Sept. 1982.

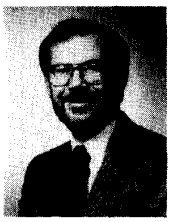
[15] J. Smith, "A simulation study of the Cray X-MP memory system," *IEEE Trans. Comput.*, vol. C-35, pp. 613–622, July 1986.

[16] M. Wolfe, "Optimizing supercompilers for supercomputers," Ph.D. dissertation, Dep. Comput. Sci., Univ. of Illinois at Urbana–Champaign, 1982.

[17] C. Yang and B. Miller, "Critical path analysis for the execution of parallel and distributed programs," in *Proc. 8th Int. Conf. Distributed Comput. Syst.*, June 1988, pp. 366–375.

Allen D. Malony (M'88) received the B.S. and M.S. degrees in computer science from the University of California, Los Angeles, in 1980 and 1982, respectively, and the Ph.D. degree in computer science from the University of Illinois at Urbana–Champaign in 1990.

He was a Fulbright Scholar to the Netherlands in 1991 and is currently an Assistant Professor in the Department of Computer and Information Science at the University of Oregon. His current research interests include performance evaluation, multiprocessor architectures, and parallel programming environments.



Daniel A. Reed (S'80–M'82) received the B.S. (summa cum laude) in computer science from the University of Missouri, Rolla, in 1978, and the M.S. and Ph.D. degrees, also in computer science, from Purdue University, West Lafayette, IN, in 1983.

He is currently a Professor in the Department of Computer Science, University of Illinois, Urbana–Champaign. He holds joint appointments in the Center for Supercomputing Research and Development and the Beckman Institute.

Harry A. G. Wijshoff was born in 1960 in Grevenbicht, The Netherlands. He received the M.Sc. degree (cum laude) in mathematics and computer science in 1983, and the Ph.D. degree in 1987, from the University of Utrecht, The Netherlands.

From 1987 until 1990 he was a visiting senior computer scientist at the Center for Supercomputing Research and Development at the University of Illinois. He was an Associate Professor at the Department of Computer Science, University of Utrecht, until July 1992. Currently, he is a Professor in the Department of Computer Science at Leiden University, Leiden, The Netherlands. His current research interests include performance evaluation, sparse matrix algorithms, programming environments for parallel computers, and parallel numerical algorithm development.