Performance Instrumentation Techniques for Parallel Systems

Daniel A. Reed*

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

Abstract. Although the nascent state of parallel systems makes empirical performance measurement, analysis and tuning critical, rapid technological evolution, coupled with short product life cycles, has often made it difficult to isolate fundamental experimental principles from implementation artifacts. By definition, the apparatus for experimental performance analysis (i.e., instrumentation specification, data buffering, timestamp generation, and data extraction) is shaped by the intended experiment and the object of study. In some environments, certain experiments are not feasible. Balancing the volume of captured performance data against its accuracy and timeliness requires both appropriate tools and an understanding of instrumentation costs, implementation alternatives, and support infrastructure.

1 Introduction

The same production economics that have made personal computers so powerful, inexpensive and ubiquitous, are driving the development of scalable parallel systems. By exploiting commodity microprocessors and memory chips, it is now technically and economically feasible to build systems that scale from tens to hundreds or thousands of processors. However, achieving a large fraction of peak performance across a range of applications has proven much more difficult than first expected — many massively parallel systems exhibit performance instability (i.e., the variance in performance is high, both in a single application and across a group of applications). Even more distressing than performance instability is our current inability to predict the performance of a particular application on a given parallel system.

As an illustration of performance instability, consider a simple *gedanken* experiment involving workstations and parallel systems. Select ten application programs, measure their execution times on an arbitrarily chosen workstation, and

* This work was supported in part by National Science Foundation grants NSF CCR87-06653 and NSF CDA87-22836 (Tapestry), NASA ICLASS Contract No. NAG-1-613, DARPA Contract No. DABT63-91-K-0004, and by grants from the Digital Equipment Corporation External Research Program and the Intel Supercomputer Systems Division.

then rank the applications based on their measured execution times. Now repeat the process for another workstation with comparable peak performance and then compare the two rankings. Not only will the rankings be permuted, but the relative separation between ranked elements also will have changed. Finally, repeat the experiment using two parallel systems with comparable peak performance.[2] Not only will there be little correlation between rankings, but the differences in program execution times may well vary by multiple orders of magnitude.

Although single figures of merit (e.g., peak MIPS, MFLOPS, or clock rate) cannot be used to predict the performance of an isolated application code, for single processor systems they do provide rough performance guidelines, and one can be reasonably confident that a system with a 100 MHz clock will execute almost any application code faster than a comparable system with a 50 MHz clock. In contrast, a parallel system with lower peak performance may well execute a wide range of codes more quickly than another that has higher peak performance. Simply put, consistently achievable performance across a broad range of applications is the desired, though still elusive, goal.

The underlying causes of performance instability and low performance lie in the patterns of interaction among application software, the operating system, and the parallel hardware. For parallel systems, these interactions involve hundreds or thousands of processors and dynamic behavior on a microsecond time scale. Just as effective management techniques for small, human organizations do not readily scale to larger groups, well-understood techniques for harnessing the power of two or four processor systems are not directly extensible to massively parallel systems. In both contexts, accurate, timely information is the prerequisite to developing and implementing decision procedures that maximize performance. Obtaining this information is the goal of performance instrumentation.

Performance instrumentation itself is part of the larger discipline of experimental performance analysis. As Fig. 1 suggests, experimental performance analysis contains four phases: hypothesis construction, identifying measurement points, instrumentation and measurement, and data analysis.

All but instrumentation and measurement depend on the experimental goal. For example, an effective task scheduling strategy for a shared memory parallel system depends on the application programming model, the cost of task preemption, the expected multiprogramming level, and the hardware's memory hierarchy. Changes to any one of these will shift the scheduling strategy design point, the experimental hypothesis, the instrumentation points, and the data analysis, but usually not the instrumentation and data capture infrastructure.

Given the enormous breadth of possible performance analysis hypotheses, as well as space limitations, techniques for performance instrumentation and data capture are the primary focus of this survey. For lucid introductions to the broader issues of hypothesis testing and performance data analysis, see [2, 23, 24].

[2] In practice, conducting this experiment on two parallel systems is a formidable task. Programming models and system configurations differ so greatly that simply porting a code to multiple architectures is problematic.

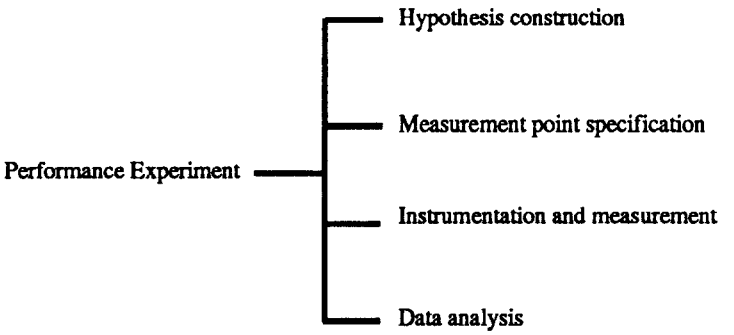

Figure 1 Experimental performance analysis phases

To provide a context for understanding experimental, parallel system performance analysis, §2 begins with a brief survey of parallel architectures and performance measurement levels, followed in §3 by a discussion of counting, timing, and tracing instrumentation. In §4, we compare hardware and software approaches to event tracing and discuss the importance of high resolution, low latency clocks. In §5 we describe potential performance instrumentation pitfalls and suggest guidelines for effective instrumentation, followed in §6 by a discussion of open problems and possible solutions. Finally, §7 concludes with a synopsis of our observations.

2 Parallel Processing and Instrumentation Levels

Although many of the techniques for experimental performance analysis apply generally to all classes of parallel systems, others are inextricably tied to particular classes of parallel architectures or particular programming models. Below, we briefly review common approaches to parallel processing, followed by a discussion of measurement levels and their instrumentation implications.

2.1 Parallel Architectures and Programming Models

Although a plethora of high-performance parallel systems have been proposed, the market is dominated by only three architecture classes: SIMD, shared memory MIMD, and distributed memory MIMD. Exemplars of these classes include the bit-serial SIMD Thinking Machines CM-2, the shared memory Cray C90 vector multiprocessor, and the Intel Paragon XP/S distributed memory multicomputer.[3] Not only does each have certain performance advantages and disadvantages, each also requires different performance measurements and differs in the ease of access to pertinent performance data.

[3] Other examples include, but are not limited to the SIMD Masspar MP-2, the Thinking Machines CM-5, Ncube/3, Cray T3D, and Convex MPP.

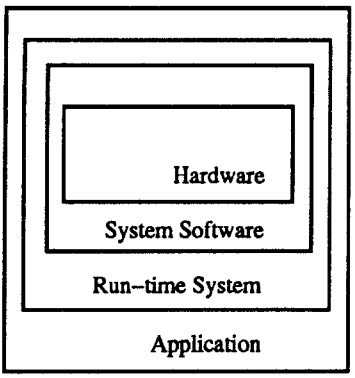

Figure 2 Performance instrumentation levels

The Thinking Machines CM-2 consists of up to 64K, one-bit processing elements (PEs), 2K high-speed floating point units, and a modest amount of local memory for each PE, all managed by a control processor that broadcasts instructions to the PEs. The PEs are connected by a two-dimensional mesh, for nearest neighbor communication, as well as a hypercube network, used for message routing to arbitrary destinations. The standard programming model is data parallel, arrays are distributed across the processors, and high-level array operations (e.g., array addition or reduction) are implemented by broadcasting instructions to the processor array. Key performance issues include maintaining a high degree of parallelism, minimizing delays for instruction broadcast, and minimizing interprocessor communication.

Because all instructions are issued by the control processor, capturing software performance data on a SIMD system is conceptually simple; one need only instrument instruction broadcast on the control processor to measure the execution time of particular operations (e.g., array or floating point operations). Moreover, because all PEs execute in lock step, one can halt instruction broadcast and interrogate the local memory of any PEs to extract additional data without perturbing the system state.

In striking contrast to the Thinking Machines CM-2, the Cray C90 contains up to 16 high-speed, pipelined vector processors that share access to a highly interleaved memory system. As a replacement for the Cray X/MP and Y/MP, the C90 relies on aggressive compilation of sequential Fortran codes to exploit vector operations, and on tasking directives to exploit multiple processors. Hardware semaphores provide synchronization for task scheduling. Key performance issues include maximizing vectorization, minimizing memory bank conflicts, and maintaining good load balance across the processors.

Shared memory simultaneously exacerbates and ameliorates performance instrumentation difficulties. The shared memory programming model encourages small, frequent state changes with synchronization only where necessary to ensure correctness. This makes it exceedingly difficult to capture the pattern of

processor interactions. Conversely, shared memory does enable an instrumentation system to observe the entire system state, although care is necessary to avoid introducing memory bank conflicts or excessive context switching.

Finally, distributed memory systems like the Intel Paragon XP/S consist of hundreds or thousands of nodes that interact via message passing rather than through shared memory. On the XP/S, the processors are connected in two-dimensional mesh via a wormhole routing network. Each node contains a local memory, a commodity microprocessor, and a interface to the routing network. Key performance issues include hiding message passing latency by computation, balancing the computation across the processors, and choosing a distribution of data across processors that minimizes communication while maximizing parallelism.

In message passing, the interaction pattern among processors is explicit, and the maximum interaction frequency is low compared to that for shared memory systems. Although the relative isolation of the processors makes it easy to capture message passing performance data, the absence of a shared memory makes determining a global order for events difficult; see §4. Moreover, extracting performance data often must rely on the same network used to pass application messages; this can perturb the system.

The absence of a central control on both shared and distributed memory MIMD systems makes unobtrusive data capture and extraction more difficult than on SIMD systems. In consequence, the majority of vendor and research performance instrumentation efforts have focused on MIMD instrumentation implementations.

2.2 Performance Measurement Levels

The goal of performance instrumentation is to provide the requisite data to answer the basic question "How fast is it?" and its consequent "What should be modified to make it faster?" The meaning of the first question depends on its context. As Fig. 2 suggests, there are at least four potential instrumentation levels, namely hardware, system software, run-time software, and application code. In general, optimization requires correlation of performance data across two or more of these levels. For example, maximizing vector lengths is key to achieving good performance on most pipelined vector processors. An ideal performance instrumentation would include hardware support to count the number of scalar and vector floating point operations and software instrumentation in the application to record loop bounds and procedure call patterns. By combining hardware and application performance data, one could identify those code fragments that most need optimization.[4]

From the instrumentation perspective, the techniques used to obtain performance data depend strongly on whether hardware, system software or application data are sought; see Table 1. Capturing hardware performance data without

[4] The Cray Hardware Performance Monitor (HPM) [9], together with application tracing, provides precisely this capability.

Table 1 Example performance measurements and instrumentation techniques

Level	Measurement	Example Technique	Support
Hardware	cache misses	counting	hardware
	network contention	timing	hardware
	instruction mix	counting	hardware
Operating system	system calls	counting/tracing	software/hardware
	context switches	counting/tracing	software/hardware
	page faults	counting	software
Run-time system	task creation	counting/tracing	software/hardware
	task synchronization	counting/tracing	software
Application	procedure occupancy	profiling/timing	software
	message passing	tracing	software/hardware

hardware support is sometimes possible, though extremely difficult. Not only are many types of hardware data are not accessible via software (e.g., cache misses or pipeline stalls), but the trend is toward increasing inaccessibility. As microprocessors continue to replace discrete component designs, previously accessible measurement points are migrating onto the chip, and packaging constraints preclude the use of scarce pins for performance data extraction.[5] For example, it is not possible to capture a complete trace of physical memory references by monitoring memory accesses at a microprocessor's chip boundary; only misses to the on-chip cache are asserted on the chip's address pins.

Unfortunately, market pressures are unlikely to force microprocessor vendors to provide access to internal performance data. The parallel systems market, which increasingly relies on commodity processor building blocks, is a tiny fraction of the microprocessor market, and the predominant consumers of microprocessors, personal computer and workstation users, have not expressed interest in hardware instrumentation support.

Despite the lack of access to microprocessor internals, a plethora of hardware performance data remains accessible. Almost all parallel systems are constructed by augmenting microprocessors with ancillary logic to support either memory coherence (shared memory systems), message passing (distributed memory systems), or instruction issue (SIMD systems). By adding hardware counters and performance data extraction paths to these components, parallel systems vendors could provide ready access to a wealth of hardware performance data at minimal cost.

Hardware instrumentation can be unobtrusive, but is necessarily limited in scope and flexibility. The time scale for hardware events is small, their frequency is very high, and the number and type of instrumentation points must be chosen

[5] Increasing inaccessibility is a problem for chip testers as well. Builtin self-test (BIST) and scan logic for serial extraction of internal state are reactions to this limitation.

when the hardware is designed. In contrast, the software performance instrumentation options are much more rich and varied — both data capture techniques and instrumentation points can be changed long after the software has been designed.

The primary distinction between application instrumentation and that for operating system or run-time systems is the use of system services. Application instrumentation and data capture are free to use any system services if that use will not substantively change the application's performance or behavior. However, when designing operating system instrumentation and data capture, one must not use any system services that are potential instrumentation targets. For example, when measuring the performance of an input/output system, the performance data capture software must not rely on the input/output system for real-time data extraction. Similarly, a separate performance monitoring task can change the task scheduling pattern, and an instrumentation of a virtual memory system should not buffer performance data in virtual memory.

3 Performance Measurement Techniques

Regardless of the instrumentation level, there are four basic approaches to performance data capture: timing, counting, sampling, and tracing. Each represents a different balance between information volume, potential instrumentation perturbation, accuracy, and implementation complexity.

3.1 Timing

Speedup, the ratio of sequential to parallel execution time, relies on the simplest form of timing — a measure of aggregate execution time. If the execution time is sufficiently large, this measure requires no system support and introduces no perturbations, a simple stopwatch suffices. Aggregate system timing is a measure of success (i.e., it allows one to estimate how closely one approached the ideal), but it provides no insight when further performance optimization is required. Instead, one must measure the execution times of individual system components.

Although detailed timing data can identify *where* a system spends the majority of its time, it is insufficient to determine when or why. A procedure or hardware component may be in use a large fraction of time time, not because it was poorly optimized, but because some other component repeatedly and unnecessarily invokes it (i.e., timing can reveal proximate bottlenecks but not when or why particular hardware or software components were invoked).

To implement a timing facility, one needs only low latency access to a clock whose resolution is high compared to the elapsed time of the events being measured.[6] Both clock resolution and access latency are critical to accurate timing; the clock resolution limits the effective granularity of measurements, and the access latency bounds the instrumentation perturbation — access times for

[6] See §4 for a discussion of clocks and clock access.

a high latency clock can exceed the lifetime of the measured behavior. Finally, unless the number of timing points is large, the total volume of performance data produced by timing instrumentation is small.

3.2 Counting and Sampling

In contrast to timing, counting records the number of times an event occurred, but not where or why. Given both counts and total times, one can accurately compute average execution times. Unless the number of counters is exorbitant, counting is efficient, minimally intrusive, and produces only a modest amount of data. To implement a counting facility, one need only allocate sufficient storage for the array of counters, then during execution, index the counter array and increment the appropriate counter.

Sampling is an approximation to counting, obtained by periodically observing the system state and incrementing a counter that corresponds to the observed state. Standard profiles (e.g., Unix gprof [5]), sample the program counter at fixed time intervals, use the program counter as the index to a bin, and increment the associated counter. After program execution, the counter value in each bin is proportional to the total time spent executing code in the associated address range.

The primary limitations of profiling are its dependence on an external sampling task and the potential errors inherent in sampling. On single processor systems, the operating system implements profiling by sampling the task program counter at each clock tick, typically every 10–20 milliseconds. If the total program execution time is too low, the sampling error may be high. The operating system dependence of standard profiling techniques makes profiling operating system activity difficult.

On parallel systems, the existence of multiple processors can skew profiling statistics [16, 11]. Consider a code fragment that achieves linear speedup on P processors. The observed execution time for that code fragment is $\frac{1}{P}$ that for the comparable sequential code, and a program counter sample will underestimate its contribution to total execution time unless the samples across all P processors are combined.

3.3 Event Tracing

Event tracing is potentially the most invasive of the four instrumentation techniques, but it also is the most general and the most flexible. Event tracing generates a sequence of event records. Each event is some significant physical or logical activity, and the event record is an encoded instance of the action and its attributes. Each record typically includes the following.

1. *what* action occurred (i.e., an event identifier),
2. the time *when* the event occurred,
3. the location *where* the event occurred (e.g., a line number), and
4. any additional data that defines the event circumstances.

Not only does event tracing identify what happened and where it happened, the event timestamps impose an order on the events that defines control flow and system component interactions.

Event tracing subsumes timing, counting, and sampling; one can compute times and counts from trace data. For example, given a trace of procedure entries and exits, one can compute the total number of calls to each procedure by counting the number of instances of each event type, as well as the total procedure execution times by matching procedure entry and exit events, computing the difference in their event times, and adding the difference to a running sum for that procedure. In addition, the trace provides the dynamic procedure call graph Similarly, a trace of message passing events on a distributed memory parallel system defines the sequence of processor interactions, as well as load imbalances due to message waiting (i.e., by computing the waiting time to receive messages).

The disadvantage of tracing is its potential intrusion, the implementation complexity, and the large volume of generated performance data. Like timing, event tracing require low latency access to a high resolution clocks, but it also must unobtrusively buffer event data and extract it without excessively perturbing the measured system. Because software events can occur on a microsecond or millisecond time scale, and hardware events on an even smaller microsecond or even nanosecond scale, even on a single processor system, the data volume can exceed a megabyte per second. Fortunately, data rates for many common events are not that high, and, as we shall see, there are techniques to reduce the rate while retaining the advantages of tracing.

Despite its potential disadvantages, tracing is the software equivalent of a hardware logic analyzer. With tracing, one can capture component interactions, dynamic behavior, and transients. Too often, performance analysts treat software as a "black box" that can be stimulated externally but not probed internally. Event tracing allows users and performance analysts to study software and hardware interactions; in short, to understand *why* components perform as they do.

Because we believe event tracing is the most powerful tool for system understanding, and because the implementation issues for event tracing are a superset of those for counting and timing, tracing is the focus of the remainder of this survey. For a discussion of counting and timing facilities, see [23, 24].

4 Event Tracing

Event tracing is possible at either the hardware or software level. As discussed in §2.2, some components of hardware performance instrumentation (e.g., probe points) are inherently system dependent, and these dependences have profound implications for the other components. This is particularly true for event tracing, where the data rates are high and hardware solutions are closely linked to the intended application. In consequence, hardware event tracing is rarely used except in isolated circumstances (e.g., to obtain address traces). In contrast, software

event tracing is effective with operating systems, run-time systems, and application codes. Moreover, the majority of the software implementation issues are independent of specific system idiosyncrasies. For this reason, our focus is support for capture of software events (i.e., events that occur in software).

Whether used with system or application software, any software event tracing implementation must resolve the following six issues:

1. timestamp generation,
2. trace buffer allocation,
3. event recording,
4. trace extraction,
5. data volume constraints, and
6. intrusion.

The expected data rates, system software environment, and the parallel architecture all constrain a particular implementation. To minimize the instrumentation intrusion, hardware support for some aspects of software data capture and extraction may be required.

As an example, on a distributed memory parallel system, event trace data can be buffered in individual processor processor memories, but when the volume of trace data exceeds the allocated storage, it must be removed from the node or some trace data must be discarded. Unless all nodes are preempted while the event trace data is extracted via the interprocessor communication network, using the network will interfere with system and application message passing and potentially change the timestamps or order of any events captured during the extraction. Given a separate, external performance data collection network, the data can be extracted from the system without using the standard communication network. However, writing the data to the collection network still consumes processor cycles. If the event data rate is extremely high, a data extraction co-processor may be needed as well.

In short, the instrumentation circumstances may dictate a software implementation, a hybrid of software and hardware, or a hardware implementation of software event tracing. Choosing an appropriate combination of hardware and software is part of the performance analyst's art.

4.1 Event Orders and Clocks

On a single processor system, sequential execution totally orders the event sequence — there is only one thread of control. For operating system or run-time systems, the event order is dependent on asynchronous events, and the event sequence may change across multiple executions. However, for most sequential application codes the event sequence is repeatable, given the same inputs. Moreover, at the application level, perturbations created by instrumentation can change the elapsed time between events, but they cannot change the event order; see Fig. 3.

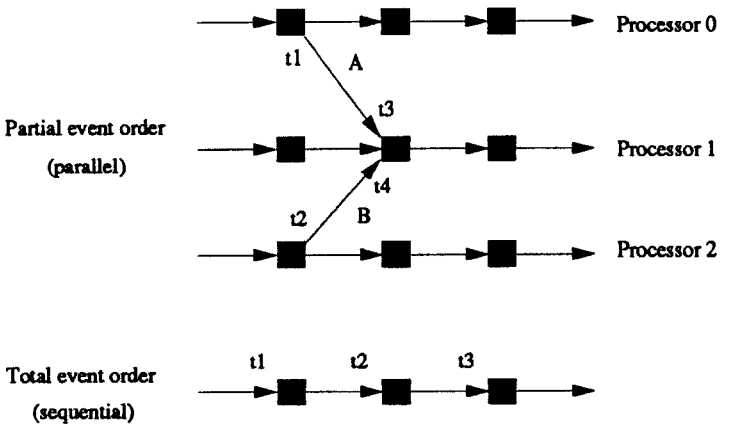

Figure 3 Partial and total event orders

On parallel systems, there are multiple threads of control, each potentially generating an event sequence. Just as for sequential execution, the event sequence for each thread is totally ordered. The global event order for the entire computation is obtained by merging the event orders from the individual event sequences.

If instrumentation differentially delays the threads, not only will the elapsed time between events change, but the global event order itself may change. Consider Fig. 3 where $t1$ and $t2$ denote the times that two different processors send messages to a third, $t3$ and $t4$ denote the times that those messages arrive, and the events A and B denote the two message arrivals. Delays on the parallel execution paths may result in the messages arriving in the order AB (i.e., $t3 < t4$) or the order BA (i.e., $t4 < t3$).

More perniciously, when merging the individual traces, it may be impossible to determine the correct event order. Continuing the example of Fig. 3, it may be impossible to determine which message was sent first (i.e., if $t1 < t2$). Three factors can prevent accurate event ordering: low resolution clocks, high latency clock access, and clock skew.

First and simplist, if the clock resolution is less than the nominal inter-event time, multiple events may have the same timestamp. On an individual processor or thread, these events still are totally ordered by the sequential execution, but across threads or processors they are unordered and apparently occur simultaneously. In the example of Fig. 3, if the measured times are such that $t1 = t2$, the order that the two message sends began cannot be resolved.

Unfortunately, many operating systems provide a user-accessible clock with a resolution equal to the power line frequency, either 50 or 60 Hz. In many cases, however, the hardware includes a higher resolution timer; it simply is not exported to the user by the system software. As processor speeds have increased, event frequencies have risen dramatically. This, coupled with low resolution clocks, has made accurate measurement of common software constructs

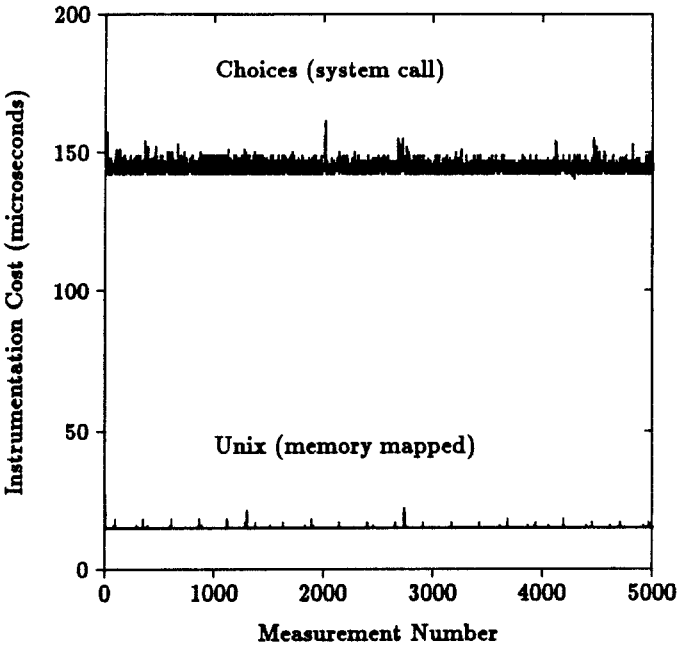

Figure 4 Encore Multimax event recording times

(e.g., procedure lifetimes) impossible.

Second, even if the clock resolution is high, a highly variable access cost can negate its effects. If events are timestamped under software control, a memory mapped clock that can be read by a single memory reference is imperative. As an example of its importance, Fig. 4 shows the cost to record an event on an Encore Multimax using two different operating systems, one with a memory mapped hardware clock and the other with the same clock accessible only via an operating system call [7]. With Encore's Unix implementation, accessing the clock requires only a memory read, and events can be recorded in as little as fifteen microseconds. Under the experimental *Choices* operating system, the system call not only increases the event recording time ten-fold, it also increases the access time variance. In addition to a protection boundary crossing, there are multiple procedure calls, memory references, and cache misses. Closely separated events on different processors may lie within the measurement uncertainty of the clock system call.

The third cause of uncertainties in global event orders is clock skew. Normally, we accept the classical physics view of time; we assume it flows at an equal rate at all locations, that it orders local and remote events, and that causality violations are impossible. Intuitively, an omniscient observer would see all events occurring in their "true" order with cause preceding effect.

Unfortunately, the classical view of time is inconsistent with reality on many parallel systems. If each processor has its own clock, measured time can po-

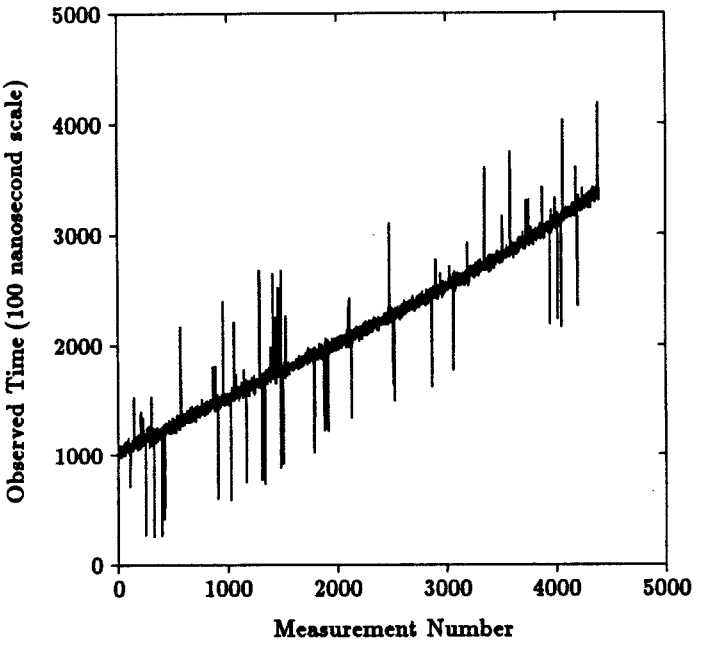

Figure 5 Clock drift in measured message transmission times

tentially flow at a different, non-uniform rate on each processor.[7] Non-uniform rates mean that the measured time between events on a single processor may be inaccurate. More troubling, however, is the effect of different rates — when comparing event timestamps on two different processors, causality violations can occur (i.e., effect can, based on timestamps, seem to precede cause).

Figure 5 illustrates the effect of inconsistent clocks on a Intel iPSC/860; each processor has a local clock with 100 nanosecond resolution. The figure shows a sequence of measured, message transmission times between a pair of nodes. The elapsed times were determined by computing the difference between the timestamp for a message send event on the transmitter and the timestamp on the associated message receive event on the receiver. The increasing time for a round trip message transmission is an artifact of clock drift, and the spikes are due to context switches on one or the other of the two processors. Near the beginning, the two clock values are nearly the same. As the measurement proceeds, the clocks drift apart. In this example, the separation is positive, but only a few of the estimates are accurate. If the identities of the receiver and sender were exchanged, the message transmission times would be negative.

Clock drift can be eliminated either by distributing the value of a single clock to all processors or by synchronizing all clocks to a master time base. Unless the clock resolution is very low, clock distribution requires hardware support via a

[7] This problem is not unique to parallel systems. The existence of unsynchronized local clocks motivated the creation of an international time base, Universal Time (UT).

clock distribution network. Even for a system with hundreds or thousands of processors, the cost of such a network is low.[8]

Software clock synchronization [3, 17] is the alternative to a global time base. Intuitively, one chooses one processor's clock as the master and synchronizes all other clocks to that master. To bound the potential difference between clocks, one initially measures the drift rate using measures similar to that in Fig. 5. and uses that to compute a resynchronization interval. The frequency of clock resynchronization depends on the drift rate, the desired error bound, and the tolerable synchronization cost, but it must be high enough to prevent causality violations in the measured event times.

To summarize, obtaining accurate, total event orders for parallel systems is dependent on high resolution, globally consistent, low latency clocks. Techniques for minimizing instrumentation overhead may involve software, hardware, or a combination of the two. Failure in any area can lead to inaccurate data and incorrect event orders.

4.2 Software Support

Because software support for performance instrumentation can take many forms, implementation issues are best understood in a specific context. Hence, we describe three different software implementations of event tracing, Crystal, Pablo, and CTrace, each intended for a different environment. Crystal [20] supports operating system and application performance data capture on the Intel iPSC/2 hypercube, the Pablo instrumentation library [19] supports portable application event tracing, and the CTrace library [10] supports application and operating system tracing on a hierarchical, shared memory parallel system.

Crystal: Operating System Instrumentation. The Intel iPSC/2 hypercube typified second generation distributed memory systems. The iPSC/2 hypercube nodes were based on an Intel 80386/80387 pair, each node contained up to sixteen megabytes of memory, and the nodes sent messages via fixed path circuit-switching [1]. In addition, a subset of the nodes supported a parallel input/output system [15] with on commodity disks. Because the iPSC/2's salient features are an integral part of current systems (e.g., the Intel Paragon XP/S and Thinking Machines CM-5), most of the performance instrumentation issues are directly transferable to newer architectures.

Crystal [13, 20, 22, 21], based on a modified version of the Intel NX/2 operating system, was an event tracing facility designed to capture both application and operating system events. Application instrumentation could be inserted either manually by users or automatically by a compiler. In either case, the generated events were passed to a modified version of Intel's NX/2 operating system, which executed on each of the hypercube nodes.

[8] Despite its low cost, many commercial systems still lack a global time base.

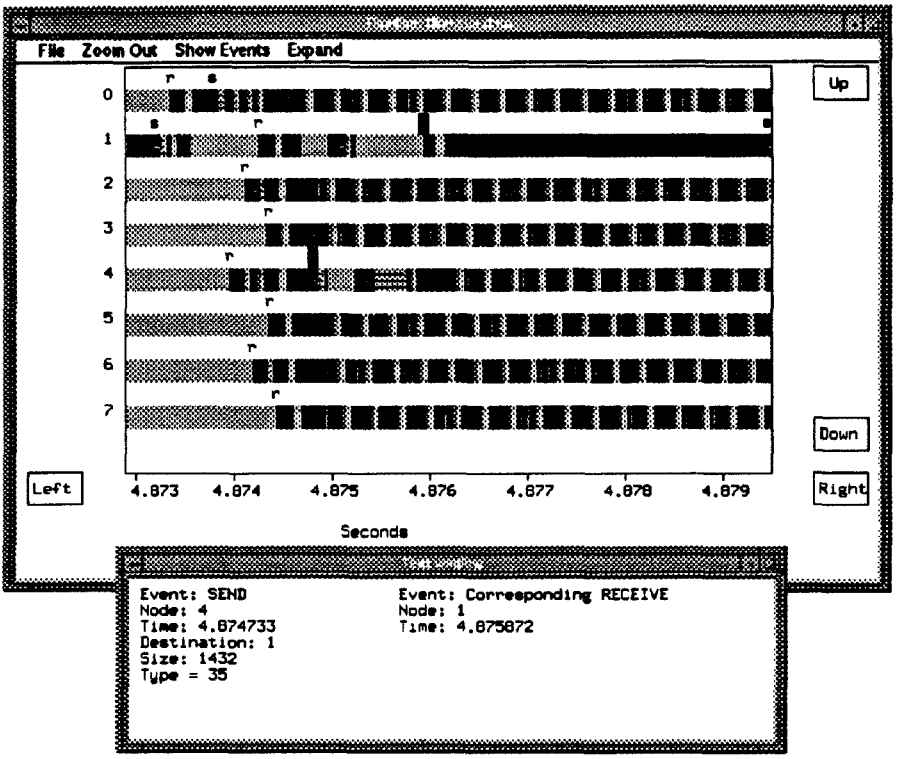

Figure 6 Crystal Intel iPSC/2 event time line

In addition to recording the application data, the modified NX/2 internally captured three classes of operating system events: message passing, process context switches, and all system calls. On each node, the application and operating system events were merged and stored in a trace buffer that was preallocated in the node's local memory. Because each node of the iPSC/2 had a local clock, the modified NX/2 synchronized all node clocks before event recording began and compensated for clock skew using the known clock drift rates.

Finally, when a node's trace buffer filled, tracing on that node normally was disabled. As a more intrusive alternative, a double buffering scheme allowed extraction of one trace buffer via the circuit-switched network while the other was being filled. The NX/2 operating system did not support virtual memory, and the Crystal instrumentation was a delicate balance between application memory needs and trace storage capacity. Allocating too much memory to trace capture left too little for the application code to execute. Allocating too little memory to trace buffers limited the amount of captured data.

Experience showed that the limited operating system instrumentation supported by Crystal was surprisingly powerful. Tracing the operating system message passing code showed the contributions of message buffer management, hard-

ware setup, and transmission time to message latency, as well as the effects of the Intel iPSC/2 communication protocol.[9] Moreover, because all application file requests were realized using messages, the instrumented message passing showed all file-related communication traffic synthesized by the operating system. Finally, the context switch and system call data exposed the coupling of application requests for services with the operating system responses, as well as idle time due to load imbalances.

Figure 6 shows a portion of a graphical time line, constructed using trace data captured by Crystal. The event trace is from an eight processor execution of a parallel linear optimization code [25]. Notice the message send, highlighted on processor 4 and the corresponding receive, highlighted on processor 1. The series of parallel horizontal lines following the "s" are the hardware message transmission of a fixed size message header. Following this, the sending node is idle (indicated by light gray) while the receiver operating system on node 1 processes the message header (indicated by dark gray). Processor 4 then transmits the remainder of the message, shown by the second series of horizontal lines between times 4.875 and 4.876. More generally, the alternating light and dark gray pattern on the time line is a sequence of context switches between the application code and the operating system, as the application probes for message arrivals.

The primary strength of Crystal, its access to operating system internals, also proved to be its greatest weakness. Retrofitting instrumentation to a proprietary operating system required source code access, and licensing restrictions prevented redistribution of modified source code. Unless the operating system source code widely available (e.g., Mach or OSF/1), operating system instrumentation is best supported by a parallel systems vendor.

Pablo: Application Instrumentation. The Pablo Performance Analysis Environment[10] [19, 18] is a portable performance instrumentation and data analysis environment designed for large-scale parallel systems, with primary emphasis on the Intel Paragon XP/S and Thinking Machines CM-5. Unlike the Crystal instrumentation, Pablo's instrumentation software is designed to be architecture neutral and easily portable to new systems.[11]

Intended primarily for capturing application performance data, the Pablo instrumentation is implemented as a library that isolates architecture-independent data buffering and recording software from architecture-dependent aspects such as processor synchronization and timestamp acquisition. The data recording

[9] The iPSC/2 used a two-phase protocol to send messages longer than 100 bytes — first, a fixed size header was sent that contained the message length and an initial portion of the message. After the receiver acknowledged the receipt and its willingness to accept additional data, the sender transmitted the remainder of the message.

[10] Pablo is a trademark of the Board of Trustees of the University of Illinois.

[11] PICL, the portable, instrumented communication library [4], developed by the Oak Ridge National Laboratory, shares these attributes, though its primary focus is on portable message passing.

model is similar to that for the Crystal instrumentation; there is a separate trace buffer for each processor or thread of control, and performance data are written to these buffers. When any buffer fills, all processors are interrupted and all write their trace data to secondary storage. The cost of buffer dumping is recorded in the trace data, allowing buffering dumping overheads to be removed from the trace data during post-processing. For parallel systems that lack a global time base, the Pablo instrumentation periodically synchronizes the processors using an implementation of Dunigan's distributed synchronization algorithm [3].

The architecture-independent instrumentation interface supports counting, interval timing, and event tracing. Counts can be accumulated or periodically flushed to trace buffers. If they are flushed only once, at the end of data capture, the canonical definition of counting holds; conversely, flushing a count each time it changes is equivalent to event tracing. Periodic flushing of counts allows the performance analyst to balance data volume against instrumentation data granularity.

To further constrain data rates and to provide user control of instrumentation perturbations, the Pablo instrumentation library supports both user-specified and internal event rate controls. The instrumentation library monitors the data recording rate for each event. While the rate lies below a pre-specified event threshold, the event stream is recorded. However, when the rate exceeds the threshold, the instrumentation library substitutes less invasive data recording (.e.g., by converting trace events into periodic counts). When the event rate declines, more detailed data recording is re-enabled. By adjusting the event rate threshold, the user can balance data rates, event volume, and instrumentation perturbation against the need for specific performance data.

For counting, interval timing, and event tracing, the Pablo instrumentation library supports user-written extension functions. Because all event data is passed to these functions before being written to trace buffers, users can create higher-level events, selectively discard certain events, or modify the event data. For example, given a sequence of procedure entry and exit trace events, an extension function could replace the raw event trace with dynamic procedure profiles, a histogram of procedure lifetimes, or a matrix of procedure call transition probabilities.

To support user extensions and to maximize portability, Pablo generates performance data files in a self-describing data format (SDDF). These files include definitions of the record formats contained in the file; the definitions are then used to parse the record instances. Because new record definitions can be easily added, new types of performance data relevant to specific application or architecture contexts can be added without modifying the Pablo data capture library.

The strengthes of the Pablo instrumentation library's approach are its portability and extensibility. However, this emphasis does limit the library's ability to exploit system-specific features and to easily capture system-level performance data.

CTrace: Shared Memory Instrumentation. CTrace [10] is an event tracing system for the experimental Cedar multiprocessor. Cedar [8] consists of multiple processor clusters connected via a multistage Omega network to a global, shared memory. In turn, the individual clusters are modified Alliant FX/8 systems, each with eight vector processors, a shared cache, and a shared cluster memory.

Cedar programs are expressed in Cedar Fortran, a Fortran dialect that supports both loop and task parallelism. Parallel loop iterations can be either restricted to a particular cluster, or they can be distributed across multiple clusters. In either case, loop iterations can execute in parallel, vector, or parallel-vector mode, Parallel tasks executing on different clusters can cooperate via the global shared memory.

CTrace supports both operating system and application event tracing, with a default set of events captured by an instrumentation of the Cedar operating system and the Cedar Fortran run-time library. Specifically, operating system context switches are recorded by instrumenting the process switching code, and task creation, activation, suspension, and deletion, as well as invocations of synchronization primitives are captured by instrumenting the run-time library. Procedure, basic block, and loop entry/exit trace event instrumentation is generated on request by the Cedar Fortran compiler; additional trace events can be specified by manually by the user.

Like Crystal, Pablo, and PICL, CTrace is implemented as a library with multiple trace buffers to eliminate contention and synchronization when recording data. The Cedar hardware maintains a global time base across all clusters, no clock synchronization is required, and event causality is assured.

Unlike most distributed memory parallel systems, the shared memory Cedar system is multiprogrammed. Elapsed times, computed using the difference between to values of the real-time clock, may be inaccurate — a task may be forced to relinquish its processor during the measured interval. Using operating system context switch trace, elapsed times are adjusted to remove these anomalies and to correctly charge each task.

4.3 Hardware Support

When the event data rate is very high, trace buffer storage capacities are low, or clock synchronization costs are high, hardware support for performance data recording and extraction becomes essential. By shifting portions of the instrumentation implementation from software to dedicated hardware, larger event traces can be captured with lower overhead.

Ideally, the balance between hardware and software implementations is determined during system design. Unfortunately, many instrumentation systems are added late in the design process, necessitating accommodation with existing design features. Below, we describe two examples of hardware support for software performance data capture, Hypermon [11, 12], a retrofit to the Intel iPSC/2 hypercube, and Multikron [14], a performance data recording chip.

Hypermon: An Instrumentation Hardware Retrofit. Users of Crystal's software instrumentation on the Intel iPSC/2 often struggled to overcome its two major limitations: insufficient event data storage capacity and the lack of an accurate, global time base. In an attempt to remedy these limitations and to explore the feasibility of retrofitting an existing system with hardware support for performance data capture, Malony developed Hypermon [11, 12], a board set for hardware data buffering and timestamp generation.

Hypermon exploited a little-known feature of the Intel iPSC/2, a five bit interface from each node board to a spare node slot in the system cabinet. This interface was mapped to the input/output address space of each iPSC/2 node, with one bit used as a valid data strobe and four bits for input/output. By modifying the Crystal instrumentation to write performance data to this address, rather than buffering it in memory, performance data from all nodes was accessible at a single location.

The Hypermon hardware exploited the data collection interface to capture, buffer, and timestamp the four-bit event data. Crystal trace events normally included an event identifier and several bytes of ancillary data; transmitting these events from each node required several, four-bit writes to the memory-mapped interface. Because there was no hardware mechanism to identify event boundaries, Hypermon generated hardware event frames, rather than trace events, when one or more nodes wrote data to the interface within any 800 nanosecond window. Each frame contained four bits from each node, a bit vector indicating which nodes had sent valid data, and a timestamp. The resulting event frames were buffered and then transferred to a Intel iPSC/2 input/output node. Based on the event data rate, the frames could either be processed as they arrived or written to secondary storage for post-processing.

Hypermon performance measurements revealed two serious bottlenecks. The four-bit interface from the nodes proved debilitating. First, and not surprisingly, the nodes were forced to assert data validity via software strobing and to shift and mask the event bytes before writing to the data capture interface; this overhead proved two orders of magnitude higher than that for software data buffering. This was an unfortunate artifact of retrofitting. An eight bit wide interface with hardware strobing would have greatly reduced the overhead and made the overheads comparable to those for software event recording.

Second, event data rates were bursty; these bursts can lead to hardware buffer data overruns. Moreover, the total event data volume increased superlinearly with the number of nodes. As an example, Fig. 7, from [12], shows the Crystal event data rate, in one millisecond windows, for a standard cell placement code run on the Intel iPSC/2. The single processor trace includes only the context switch events that occur each fifty milliseconds. As the number of nodes increases, the number of message passing events increases and the data rate rises dramatically. In one second intervals, the data rate can exceed one megabyte/second for even a modest number of nodes.

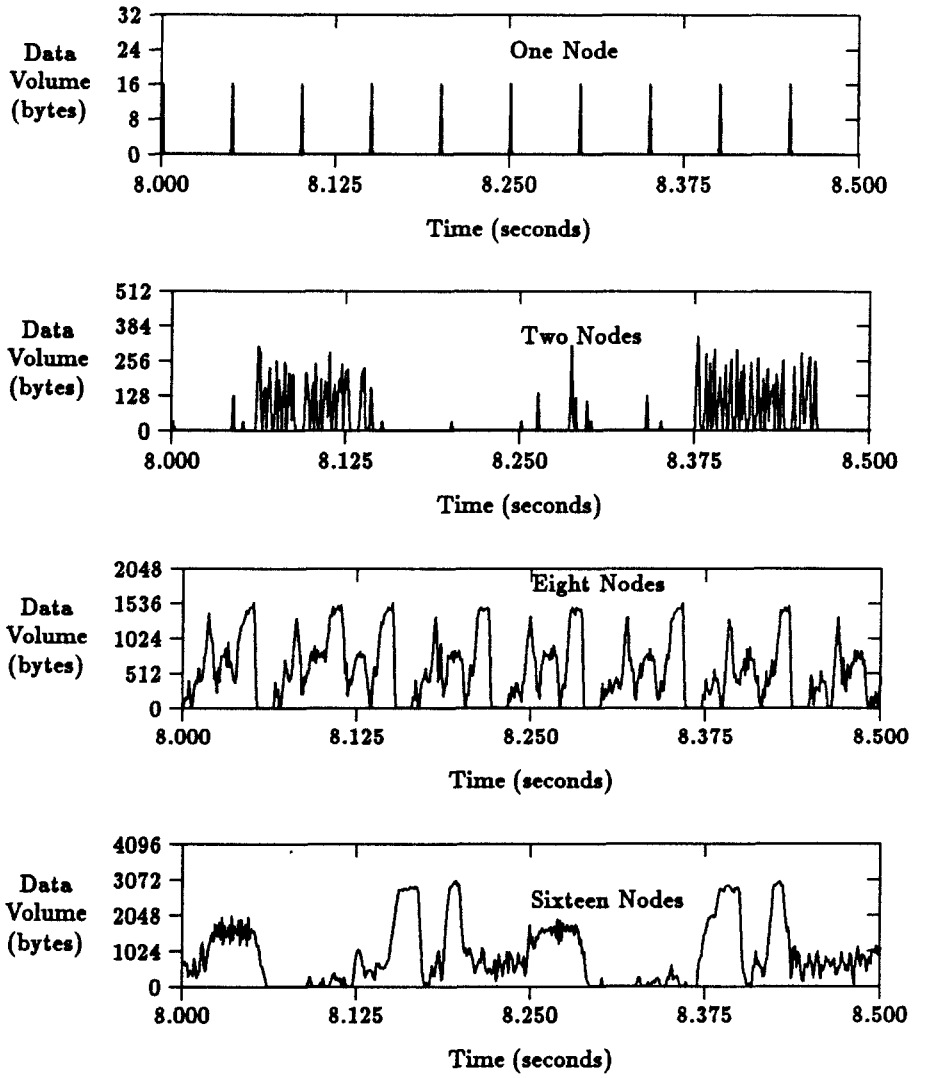

Figure 7 Crystal event data rates for standard cell placement

Multikron: A Performance Monitoring Chip. The NIST Multikron [14] integrates support for counting, event trace buffering, timestamp generation, and data extraction on a single chip. In its intended operational mode, each node or processor of a parallel system would include a Multikron chip for unobtrusive data recording.

Unlike Hypermon's constrained, four-bit interface, Multikron supports a set of 64-bit, memory-mapped interfaces that can be both read and written. Some of the interfaces are used for configuration commands, some to record event trace data, and some to query the Multikron state. Because the data recording and

timestamp generation are managed by the Multikron hardware, instrumentation points consist only of instruction stores to the appropriate Multikron addresses.

The Multikron chip generates sixteen-byte hardware trace records that contain a 40 bit timestamp from a 10 MHz clock, processor and process identifiers, and 48 bits of event trace data. Optionally, the hardware trace record can also contain sixteen, 32-bit resource counters. These resource counters can be either physically connected to hardware signals if accessible (e.g., cache misses or interrupts) or incremented under software control.

To reduce the overhead for data recording, the Multikron supports a set of registers that contain the identity of both the local node and the currently executing process. The node identity is specified by wiring appropriate Multikron pins to a hardware node identification source; the process identifier register can be maintained by instrumenting the operating context switch code to update it appropriately. Because the contents of these registers are automatically prepended to all Multikron trace records, the overhead for most trace events is one or two store instructions.

The Multikron chip also contains a sixteen-bit filter register. The contents of this trace register and the low-order four bits of the memory-mapped Multikron address where the trace data is stored determine the trace record's disposition. The low-order four address bits select one of the sixteen bits in the trace filter register. If that bit is set, the just stored data are used to construct a trace record, otherwise the data is discarded. With a filter register, code instrumentation can be selectively enabled and disabled simply by changing the contents of the filter register.

Finally, each Multikron chip supports a synchronous, byte-wide, external data collection interface with a two-way handshaking protocol. The maximum network data extraction rate is roughly 1.5 million, sixteen byte trace records per second.

5 Instrumentation Guidelines and Pitfalls

On parallel systems, the range of potential performance analysis goals is broad, and the types of performance data needed to test performance hypotheses are equally diverse. To meet their instrumentation needs, vendors, performance analysts, researchers, and application software developers have all developed performance instrumentation hardware and software. Some instrumentation tools were designed to explore the design space for parallel computer systems, others to optimize system or application software performance.

Despite the diversity of intents and the variety of instrumentation techniques, several general lessons have emerged from the design, implementation, and use of multiple generations of performance instrumentation hardware and software. Succinctly,

1. instrumentation is best included early in a system's design, rather than retrofitted to an existing system,

2. performance data rates must be balanced against instrumentation overhead and data utility,
3. no single instrumentation technique is appropriate in all circumstances, and finally,
4. some aspect of the captured data usually surprises the analyst, motivating additional instrumentation.

Although these observations apply to performance instrumentation on any computer system, designing instrumentation for parallel systems poses special challenges. First, only a small portion of the large parallel system design space has been explored, and both hardware and software architectures for parallel systems are evolving rapidly. By the time instrumentation techniques for a particular parallel architecture are tested and well-understood, that architecture may have been abandoned in favor of another.[12] Second, parallelism introduces partial, rather than total, event orders and the data volume problems inherent with large numbers of processors. Below, we summarize some guidelines and pitfalls when developing instrumentation specifically for parallel systems.

5.1 Instrumentation Infrastructure

Implementation of a performance instrumentation system is necessarily dependent on extant hardware and software features and services. The absence of particular feature may preclude certain measurements. For example, measuring individual procedure invocation lifetimes is impossible with a clock whose resolution is equal to the power line frequency. To capture timestamped event traces on a parallel system, one minimally needs

1. high resolution clocks (i.e., microsecond or better),
2. memory-mapped, low latency clock access, and
3. global clock synchronization,

with the maximum allowable clock drift bounded by the clock resolution. For software events, the timestamp clock resolution need not be equal to that of the processor clock, but it should be close to an instruction execution time. When capturing hardware events, the clock resolution must equal or exceed that for the processor clock.

The data buffering and extraction facility must support bursty, potentially high volume, event data while minimizing the number of system services used. If the data rates are sufficiently high to perturb execution and stress a software data capture implementation, one should first ask if all the data is really necessary to understand the phenomenon being studied. If not, less invasive instrumentation (e.g., counting rather than tracing) is appropriate. Otherwise, hardware support for data buffering and extraction (e.g., like that provided by the NIST Multikron chip [14]) may be necessary.

[12] This temporal dependence is a cogent argument that vendors should include support for performance instrumentation early in their system designs.

Finally, quantifying instrumentation perturbation is difficult. To determine if instrumentation is perturbing system behavior, disable some subset of the instrumentation points or substitute less invasive instrumentation (e.g., counting rather than tracing). Where possible, compute equivalent performance metrics from both instrumentations and compare their values for consistency. Although this does not guarantee the absence of perturbation, it does lessen the likelihood that it is undetected.

5.2 Instrumentation Probes

Although choosing appropriate instrumentation points is constrained by the performance analysis goal, inappropriate instrumentation can grossly perturb system behavior and quickly generate large volumes of inaccurate performance data. Across a broad range of parallel architectures and performance experiments, capturing certain types of performance data has repeatedly proved valuable, while not excessively perturbing system activity.

In an operating system, capturing

1. processor context switches,
2. interrupts, and
3. system calls

provides the largest return on instrumentation investment. These instrumentation points capture the interactions of application code with system services, task scheduling patterns, busy and idle times, and many internal operating system component interactions. Moreover, only a few instrumentation points are required, and changes to system services (e.g., file systems or scheduling algorithms) normally do not affect the instrumentation.

In an application, standard instrumentation points include

1. procedure entries and exits,
2. loop entries and exists,
3. on shared memory systems, synchronization and tasking primitives, and
4. on distributed memory systems, message passing primitives.

When inserting instrumentation, it is best to begin with the smallest possible set of instrumentation points, then insert additional points based on an analysis of the previously captured data. When instrumenting nested loops, ensure that lifetime of the inner loop is substantially larger than the instrumentation overhead; otherwise instrumentation will dominate the lifetime of the loop nest.

Finally, recognize that inserting instrumentation in source code can inhibit certain compiler optimizations. For example, bracketing the body of a procedure with two instrumentation points to capture its lifetime may prevent a compiler from inlining the code at the point of call. Similarly, source code instrumentation can prevent loop interchanges, change register allocations, and inhibit local code motion. To mitigate many of these effects, compilers should automatically generate standard types of instrumentation in response to user requests.

5.3 Instrumentation Scalability

Scalability is a key feature of most high-performance parallel systems. Using standard building blocks that contain a processor, local memory, and a network interface, a single parallel architecture can scale from tens to hundreds or thousands of processors. To achieve high performance and to exploit architectural scalability, system and application software must scale as well.

Regrettably, some instrumentation techniques do not scale to hundreds or thousands of processors. As an illustration, Table 2 shows the event data rates and expected event volumes when capturing processor utilizations and message send events on a 1024 processor system. Message tracing produces nearly 300K[13] bytes second, and even the simple processor utilization metric produces performance data at 32K bytes/second.

Moreover, as §4.3 illustrated, event data volume is not a linear function of the number of processors. For the example of Table 2, the time interval between message passing events will decrease as the number of processors increases, and the total data rate will increase. In small time intervals, the message passing event data rate might approach 3–5 megabytes/second for a thousand processor system.

Simply put, for systems with hundreds or thousands of processors, either hardware support for data capture or, preferably, real-time data reduction is imperative, For massively parallel systems, real-time data reduction is itself a parallel task. The number of data reduction processors must scale with the par-

Table 2 Example event data rates for 1024 processors

Processor Utilization	Message Traffic
processor identifier utilization estimate	source processor source task destination processor destination task timestamp message length message type
Data capture interval (milliseconds)	
250	100
Total data rate (bytes/second)	
32K	287K
Total data volume (one hour)	
118 MB	1 GB

[13] fpractice, the actual amount is nearly twice this high because both message transmission and message receipt are traced.

allel system size, and a separate data extraction network must connect the data reduction processors to the data sources. Using the NIST Multikron chip as an example, one might place a Multikron chip on each processor, connect the external interfaces of each group of 8–16[14] Multikron chips to a single data reduction processor, and connect the set of data reduction processors via a high-speed network. In essence, one constructs two parallel systems, one executing the application and a second, smaller system, connected to the first via the Multikron chip external data collection interfaces.

6 Open Instrumentation Problems

Despite our breadth of experience with instrumentation techniques for sequential systems, and our growing experience with parallel systems, many open problems remain. Of these, two of the most pressing are those associated with data parallel languages and performance queries.

The tacit assumption underlying source code instrumentation is that the organization and structure of the compiler-generated code are similar to that in the source code.[15] When this assumption is false, instrumentation may either inhibit or change the normal optimizations or it may measure something other than what might expected when examining the source code. Compilation of data parallel programs for distributed memory parallel systems is an apt illustration.

Historically, most distributed memory parallel systems were programmed in single program multiple data (SPMD) mode using an explicit message passing style, and standard workstation compilers were used to generate code. Data parallel languages like High-Performance Fortran (HPF) [6] express parallelism by specifying parallel operations on arrays that have been distributed across the memories of the system. Compilers for data parallel languages then create code that reads and writes the distributed arrays using compiler-synthesized message passing.

Not only must the translation from data parallel source code to message-based executable code bridge a large "semantic gap," but the translation procedure is hidden from the application programmer. Moreover, the translation is strongly dependent on how the arrays are distributed and accessed; small changes to either can dramatically alter the generated code.

Instrumenting the data parallel source code will not reveal the causes of poor performance; they lie in both the application source code and the compiler-synthesized code. Conversely, instrumenting the compiler-synthesized code provides accurate performance data but no mechanism to relate that data to source code constructs.

[14] The exact number depends on the event data rate, the complexity of the data reduction operations, and the speed of the data reduction processors.

[15] This assumption also underlies the implementation of most breakpoint debuggers. New techniques for debugging optimised code remain an active research topic, and only a few commercial debuggers now support it.

Obtaining accurate performance data that can be correlated with source code is an open research problem. However, it is clear that effective performance tuning and performance correlation for data parallel codes will require compiler support; it is not possible via standard source code instrumentation.

Ideally, the compiler would synthesize performance instrumentation and ancillary tools would reduce the resulting data to satisfy user performance queries. This query-response model differs from current approaches in two ways. First, the instrumentation points would be generated by the compiler, based on its knowledge of program structure and the synthesized code, and would not be visible to the user. Second, data analysis is inextricably tied to compilation and code generation. Only with access to program dependencies and generated data access patterns can query responses be computed. Implementing a query-response performance analysis model for data parallel languages will require close coupling of performance analysis tools, compiler-synthesized instrumentation, data capture libraries, and the compiler's program analysis data base.

7 Summary

Although parallel systems continue to change rapidly, a set of standard performance instrumentation techniques for parallel systems has begun to emerge. High resolution clocks with low access costs are fundamental to unobtrusive instrumentation. Similarly, data capture and extraction techniques must support high volume, bursty performance data rates. For massively parallel systems, hardware support for data extraction and real-time data reduction may be necessary if detailed event data are required.

Despite advances, many open issues remain, notably techniques for performance instrumentation and analysis of codes written in data parallel languages. To bridge the semantic gap between program source and generated code, performance analysis and instrumentation must be closely coupled with compilation.

Acknowledgments

My heartfelt thanks to the past and present members of the Pablo and Picasso research groups, without whom this work would not have been possible. Special thanks to Allen Malony, now at the University of Oregon, for many fruitful, pleasant discussions about instrumentation techniques.

References

1. ARLAUSKAS, R. iPSC/2 System: A Second Generation Hypercube. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, Volume I* (Pasadena, CA, Jan. 1988), Association for Computing Machinery, pp. 38–42.
2. DONGARRA, J. J., AND TOURANCHEAU, B., Eds. *Environments and Tools for Parallel Scientific Computing.* North-Holland Publishing Company, 1992.

3. DUNIGAN, T. H. Hypercube Clock Synchronisation. *Concurrency: Practice and Experience 4*, 3 (May 1992), 258–268.
4. GEIST, G. A., HEATH, M. T., PEYTON, B. W., AND WORLEY, P. H. A User's Guide to PICL A Portable Instrumented Communication Library. Tech. Rep. ORNL/TM-11616, Oak Ridge National Laboratory, Aug. 1992.
5. GRAHAM, S., KESSLER, P., AND MCKUSICK, M. gprof: A Call Graph Execution Profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction* (Boston, MA, June 1982), Association for Computing Machinery, pp. 120–126.
6. HPFF. High-Performance Fortran Language Specfication, version 1.0. Tech. rep., High Performance Fortran Forum, May 1993.
7. KOHR, D. R., ZHANG, X., REED, D. A., AND RAHMAN, M. Object-Oriented, Parallel Operating Systems: A Performance Study. Tech. rep., University of Illinois at Urbana-Champaign, Department of Computer Science, May 1993.
8. KUCK, D. J., DAVIDSON, E. S., LAWRIE, D. H., AND SAMEH, A. H. Parallel Supercomputing Today and the Cedar Approach. *Science 231* (February 28 1986), 967–974.
9. LARSON, J. Cray X-MP Hardware Performance Monitor. *Cray Channels* (1985).
10. MALONY, A. D. Multiprocessor Instrumentation: Approaches for Cedar. In *Instrumentation for Future Parallel Computing Systems*, M. Simmons, R. Koskela, and I. Bucher, Eds. Addison-Wesley, 1989, pp. 1–33.
11. MALONY, A. D. *Performance Observability*. PhD thesis, University of Illinois at Urbana–Champaign, Department of Computer Science, Aug. 1990.
12. MALONY, A. D., AND REED, D. A. A Hardware-Based Performance Monitor for the Intel iPSC/2 Hypercube. In *Proceedings of the 1990 ACM International Conference on Supercomputing* (June 1990), Association for Computing Machinery, pp. 213–226.
13. MALONY, A. D., REED, D. A., AND RUDOLPH, D. C. Integrating Performance Data Collection, Analysis, and Visualisation. In *Parallel Computer Systems: Performance Instrumentation and Visualization*, M. Simmons and R. Koskela, Eds. Addison-Wesley Publishing Company, 1990, pp. 73–97.
14. MINK, A., AND CARPENTER, R. J. Operating Principles of MULTIKRON Performance Instrumentation for MIMD Computer. Tech. Rep. NISTIR 4737, National Institute of Standards and Technology, Mar. 1992.
15. PIERCE, P. A Concurrent File System for a Highly Parallel Mass Storage Subsystem. In *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers and Applications* (Monterey, CA, Mar. 1989), Association for Computing Machinery, pp. 155–160.
16. PONDER, C., AND FATEMAN, R. Inaccuracies in Program Profiling. *Software: Practice and Experience 18*, 5 (May 1988), 459–467.
17. RAMANATHAN, P., SHIN, K. G., AND BUTLER, R. W. Fault-tolerant Clock Synchronisation in Distributed System. *IEEE Computer 23* (1990), 33–42.
18. REED, D. A., AYDT, R. A., MADHYASTHA, T. M., NOE, R. J., SHIELDS, K. A., AND SCHWARTZ, B. W. The Pablo Performance Analysis Environment. Tech. rep., University of Illinois at Urbana-Champaign, Department of Computer Science, Nov. 1992.
19. REED, D. A., OLSON, R. D., AYDT, R. A., MADHYASTHA, T. M., BIRKETT, T., JENSEN, D. W., NAZIEF, B. A. A., AND TOTTY, B. K. Scalable Performance Environments for Parallel Systems. In *Proceedings of the Sixth Distributed Memory Computing Conference* (1991), IEEE Computer Society Press, pp. 562–569.

20. REED, D. A., AND RUDOLPH, D. C. Experiences with Hypercube Operating System Instrumentation. *International Journal of High-Speed Computing 1*, 4 (Dec. 1989), 517–542.
21. RUDOLPH, D. C. Performance Instrumentation for the Intel iPSC/2. Master's thesis, University of Illinois at Urbana–Champaign, Department of Computer Science, July 1989.
22. RUDOLPH, D. C., AND REED, D. A. CRYSTAL: Operating System Instrumentation for the Intel iPSC/2. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications* (Monterey, CA, Mar. 1989), pp. 249–252.
23. SIMMONS, M., AND KOSKELA, R., Eds. *Parallel Computing Systems: Performance Instrumentation and Visualization*. Addison-Wesley Publishing Company, 1990.
24. SIMMONS, M., KOSKELA, R., AND BUCHER, I., Eds. *Instrumentation for Future Parallel Computing Systems*. Addison-Wesley Publishing Company, 1989.
25. STUNKEL, C. B., FUCHS, W. K., RUDOLPH, D. C., AND REED, D. A. Linear Optimisation: A Case Study in Performance Analysis. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications* (Monterey, CA, Mar. 1989), pp. 265–268.