A Hardware-Based Performance Monitor for the Intel iPSC/2 Hypercube

Allen D. Malony*

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

Daniel A. Reed†

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

Abstract

The complexity of parallel computer systems makes *a priori* performance prediction difficult and experimental performance analysis crucial. A complete characterization of software and hardware dynamics, needed to understand the performance of high-performance parallel systems, requires execution time performance instrumentation. Although software recording of performance data suffices for low frequency events, capture of detailed, high-frequency performance data ultimately requires hardware support if the performance instrumentation is to remain efficient and unobtrusive.

This paper describes the design of HYPERMON, a hardware system to capture and record software performance traces generated on the Intel iPSC/2 hypercube. HYPERMON represents a compromise between fully-passive hardware monitoring and software event tracing; software generated events are extracted from each node, timestamped, and externally recorded by HYPERMON. Using an instrumented version of the iPSC/2 operating system and several application programs, we present a performance analysis of an operational HYPERMON prototype and assess the limitations of the current design. Based on these results, we suggest design modifications that should permit capture of event traces from the coming generation of high-performance distributed memory parallel systems.

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

†Supported in part by the National Science Foundation under grants NSF CCR86-57696, NSF CCR87-06653 and NSF CDA87-22836, by the National Aeronautics and Space Administration under NASA Contract Number NAG-1-613, and by grants from AT&T, Intel Scientific Computers and the Digital Equipment Corporation External Research Program.

1 Introduction

To observe and measure the performance characteristics of a parallel system, the performance analyst must implicitly or explicitly solve several problems. First, one must *specify* the desired performance data and determine instrumentation points. Given a description of the desired data and associated data capture points, one must *capture and record* the data while balancing instrumentation coverage against instrumentation intrusion. Finally, one must *present, reduce, and analyze* the data (e.g., via a statistical analysis or visualization tools). An integrated performance monitoring environment for a particular parallel computer system necessarily requires many design compromises in performance specification, data capture and recording, and data reduction and presentation [6]. Further, performance measurement, no matter how unobtrusive, introduces perturbations, and the degree of perturbation must be balanced against the need for detailed performance data [7].

In this paper, we focus on the simple, though difficult, problem of data capture and recording. As technology makes possible the creation of parallel systems far from the center of the von Neumann architecture spectrum, we believe that capture of detailed performance data will become increasingly important to both system designers and application software developers. Although software performance data can take many forms, including samples, counts, and running sums, event traces provide greater flexibility; given a trace, one can compute counts, sums, distributions, and profiles of both procedure occupancy and parallelism.

Designing a machine independent data capture system is exceedingly difficult — the variety of system interfaces limits generality. Because message-based parallel systems typically lack both a global clock for event timestamps and a common memory for event data buffering, they pose particularly vexing, and interesting, instrumentation problems. Without hardware support, event trace extraction either must compete with system and application programs for access to network communication bandwidth, or the traces must be buffered

locally in individual node memories, limiting both trace size and application program memory. Given these problems, hardware support for event data capture and recording is crucial to minimizing instrumentation perturbations.

In this paper, we describe the design of an operational hardware prototype, called HYPERMON, that supports hardware capture, timestamp generation, and recording of software trace events from the Intel iPSC/2 hypercube. In addition to reporting the measured data capture rate, we compare the performance perturbations of software data recording and hardware data capture using HYPERMON. Based on an analysis of this data and the HYPERMON design, we present some lessons that should guide design of performance data capture hardware for the coming generation of high-performance distributed memory parallel systems.

The remainder of the paper is organized as follows. In §2, we briefly describe the Intel iPSC/2 hypercube and its software environment. We follow in §3 with a detailed description of the HYPERMON design. In §4, we describe the particular hardware configuration used to test HYPERMON, followed in §5 by a performance analysis of the HYPERMON design. Based on this data, in §6 we describe guidelines for the design of performance data recording systems.

2 Intel iPSC/2 Description

The design of hardware support for performance data capture necessarily depends on the underlying parallel system — system software and hardware determine the requirements for and limitations of the data capture interface. Thus, an instrumentation environment must be understood in the context of the intended architecture — the Intel iPSC/2, a second generation, distributed memory parallel system.

The Intel iPSC/2 hypercube [1, 3] incorporates evolutionary advances in technology, including an Intel 80386/80387 microprocessor pair, a 64K byte cache, and up to 16 megabytes of memory on each node. The iPSC/2 includes an autonomous routing controller to support fixed path, circuit-switched communication between nodes. This communication system eliminates most of the store-and-forward latency that existed in earlier distributed memory systems.

The software development interface for the iPSC/2 is a standard Unix system that transmits executable programs to the nodes, accepts results from the nodes, and can, if desired, participate in the computation. Finally, the Unix host supports node file I/O to its local disk and to remote disks via a network file system protocol.

To reduce dependence on the Unix host and to provide input/output performance commensurate with computing power, the Intel iPSC/2 nodes also support link connections to I/O nodes. Each I/O node is identical to a standard compute node, with the exception of an additional daughter card that provides a SCSI bus interface. The SCSI bus supports up to seven peripherals and has a peak transfer rate of 4 megabytes/second. Physically, the I/O sub-system can be packaged as a separate cabinet with cable connections to the compute nodes.

Because the I/O nodes provide a superset of the compute node functionality, software support for disk and file access is realized by augmenting the NX/2 node operating system on both the compute and I/O nodes. The Concurrent File System (CFS) allows application programs to create, access, or modify files on both the hypercube host and the individual hypercube disks. As we shall see, this provides the mechanism for archiving performance data after its capture by our HYPERMON instrumentation support hardware.

Although each iPSC/2 node contains a local clock with one microsecond resolution, the node clocks are not globally synchronized and can drift apart at a measurable rate. Consequently, event timestamps on different nodes may violate casuality (e.g., a message might appear to be received before its transmission). Although software techniques can ameliorate the effects of clock drift by synchronizing the clocks and reordering timestamped events, a high-resolution, global time reference is the simplest and most desirable solution.[1] Even with a global time reference, the distributed event data still must be collected for analysis and presentation.

3 HYPERMON Design

The absence of a global, accurate, and consistent time exacerbates the already difficult measurement and recording of distributed events [5]. The constraints on measurement resolution created by distributed clock synchronization, coupled with the overheads of software tracing, limit the range of performance behavior that can be accurately observed. The design of the HYPERMON architecture attempts to circumvent these problems. Below, we describe the design and operation of HYPERMON, followed in §5 by a summary of results obtained from HYPERMON bandwidth experiments and performance tests using real message passing programs.

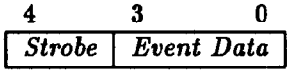
3.1 iPSC/2 Event Visibility

The hardware basis for HYPERMON is a little-known, though standard, feature of the iPSC/2 that makes possible external access to software events generated by each node. Five "performance" bits from a port in

[1] See [8] for a description of software causality maintenance and its limitations.

the I/O address space on each iPSC/2 node board are routed via the system backplane to an empty slot in the system cabinet. The standard cabinet holds up to 32 iPSC/2 nodes and all 160 signals are present at the spare node slot. However, the current HYPERMON prototype supports data capture from only the first 16 nodes.

Because the software performance instrumentation generates event data by writing to an I/O port, one bit must be reserved as a software strobe to signal that the remaining four event data bits are valid. The five data bits written to the I/O port are interpreted as follows.



A software event is generated by writing the event data to the 5-bit port, first with a strobe of zero then with a strobe of one. The 16 MHz 80386 microprocessor in the Intel iPSC/2 requires approximately six cycles to complete an I/O write operation, versus two for a standard memory access. Thus, a minimum of twelve cycles are needed to write a 4-bit software event.

Clearly, the sixteen possible events representable with a 4-bit event identifier greatly restrict the range of possible instrumentation. If additional events are needed, or, equally likely, if data are associated with an event, multiple I/O write operations will be required. A performance instrumentation of the Intel NX/2 operating system [10], produces a range of event sizes, ranging from two to thirteen bytes plus a timestamp. Moreover, an analysis of these operating system traces shows that most events are four or more bytes, excluding the software timestamp.

Unfortunately, transmitting larger events is expensive. In addition to the twelve cycle memory access penalty for each 4-bit I/O operation, there is software shift and mask overhead to extract 4-bit items from event words. Clearly, a larger I/O event field is desirable to increase event bandwidth and to reduce the cost of bit field extraction. However, backplane hardware constraints limit the number of available backplane signals. Despite these limitations, hardware support for data collection permits real-time extraction of performance data and, consequently, capture of larger traces than otherwise possible with node memory trace buffering.[2]

3.2 Hypermon Architecture

Figure 1 shows the primary components of the HYPERMON architecture and their physical relation to the Intel iPSC/2. Reflecting the physical packaging of the iPSC/2, HYPERMON is partitioned between the cabinet that contains the iPSC/2 compute nodes and the cabinet containing the I/O nodes and disks. As just described in §3.1, each iPSC/2 compute node independently sends 5-bit (four bits and a strobe) event data to HYPERMON.

In the compute node cabinet, the HYPERMON *event regeneration board* (ERB) converts the 5-bit TTL-level event signals from each node into differential form before transfer to the *event capture board* (ECB) residing in the I/O cabinet. The ECB captures the events, generates global timestamps, and stores the resulting event data in internal memory buffers for access by I/O nodes. To prevent disruption of event data capture and perturbation of user I/O requests, one or more I/O nodes are dedicated to event data recording.

In principle, the I/O nodes can be used for preliminary analysis of the event data. In practice, however, the desirability of real-time data reduction, with a possible decrease in disk I/O requirements for data recording, must be balanced against the probability of ECB data buffer overruns if too many I/O node compute cycles are diverted from disk service. Clearly, the most efficacious mix of real-time data reduction and disk recording, with post-mortem analysis, depends on the event data rate and instrumentation requirements.

3.2.1 Event Capture

Figure 2 shows the functional design of the event capture board, the primary hardware component of HYPERMON. There are five major parts: event data queueing, event strobe synchronization, timestamp generation, event frame construction, and I/O node interface; each is described below.

After signal regeneration by the HYPERMON ERB board, the 4-bit event data from each iPSC/2 node are placed in a separate FIFO buffer that is clocked by the corresponding software event strobe signal (i.e., the fifth bit from each node). Each node FIFO is 64, 4-bit entries deep and provides buffering during the event frame construction process; see below. Figure 3 shows the timing diagram for the two writes needed to assert valid event data from a node.

Because the individual node clocks are asynchronous, this is no direct timing relationship among the valid event data in FIFOs for different nodes. Thus, early in the HYPERMON design, we were forced to decide where to synchronize the externally received event data with the internal ECB clock. Because the event data FIFOs provide implicit synchronization of the data bits, only the event strobe signals need be sampled relative to the internal ECB clock. In the HYPERMON design, the software event strobes are sampled in successive 800 nanosecond time windows.[3] Valid event strobes within

[2] In §5 and §6, we will return to the question of node hardware support for data extraction. The current HYPERMON design reflects the 5-bit iPSC/2 data extraction interface.

[3] The synchronization period can be shortened to 400 nanosec-

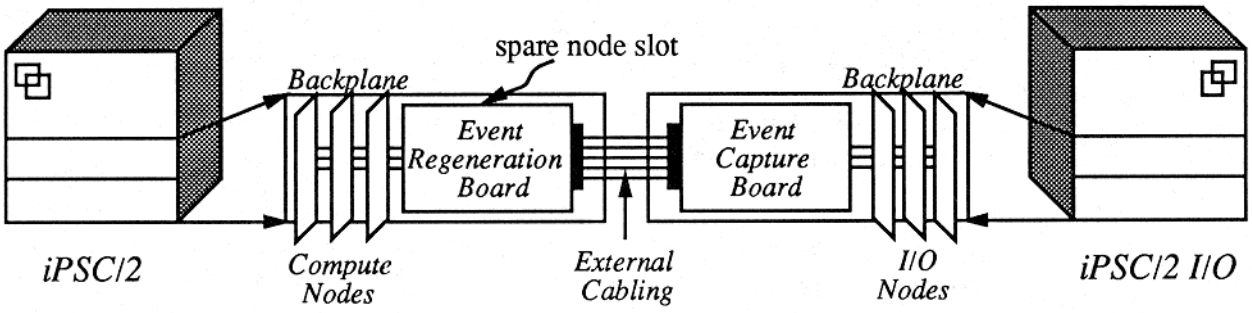


Figure 1: HYPERMON Architecture

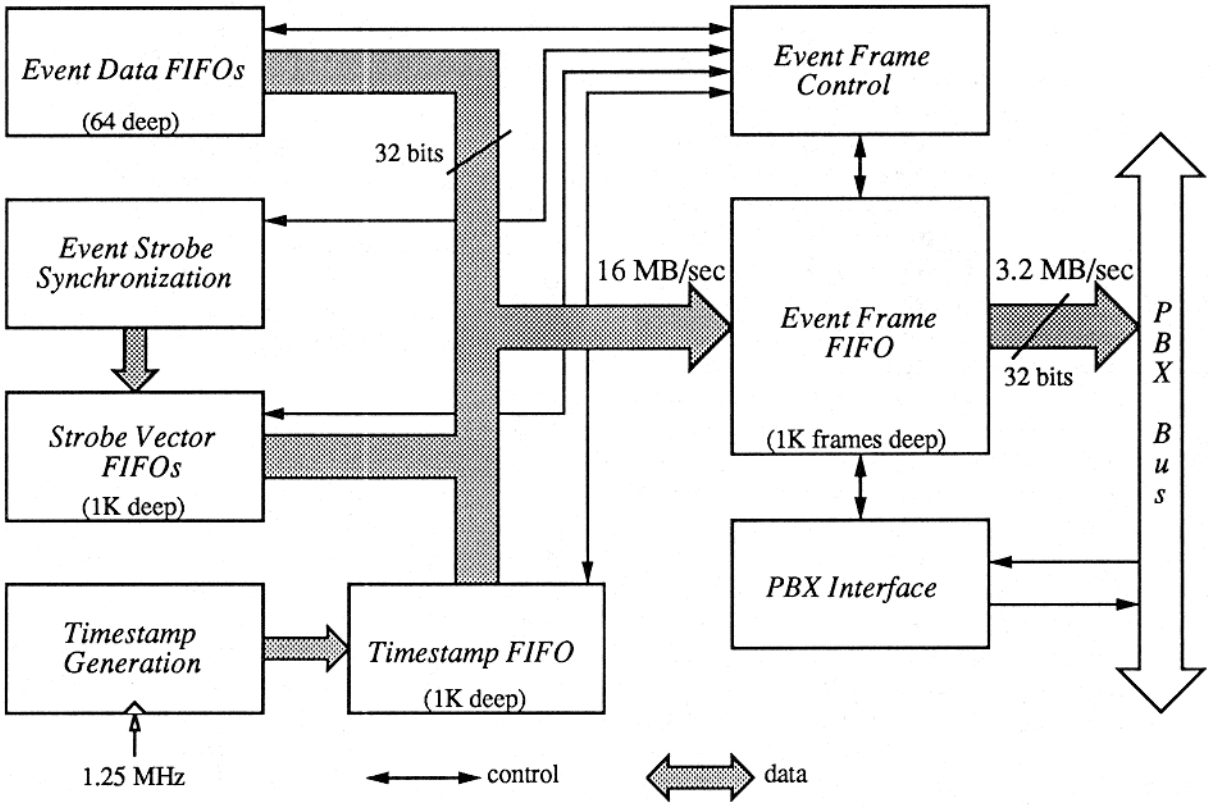


Figure 2: HYPERMON Event Capture

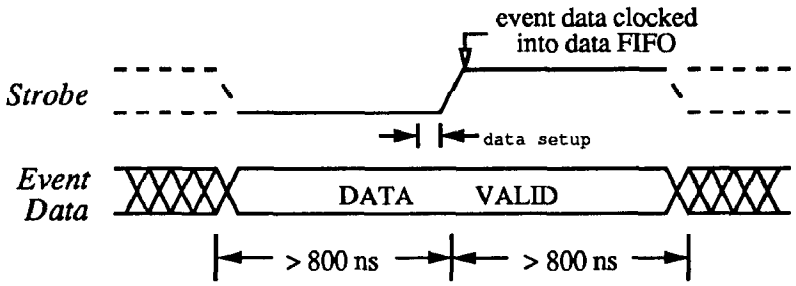


Figure 3: Event Data Timing

a time window are marked in an *event strobe vector* and stored in the strobe vector FIFO.

Figure 4 illustrates event strobes from different nodes and the corresponding strobe vectors during three successive time windows. The shaded bits in each strobe vector indicate which event data FIFOs have valid data from compute nodes.

Separate from the synchronization issue is the generation of event data timestamps. Although high-level events can be of any length, and one need generate only one timestamp for each event, the HYPERMON hardware must assume that every four bits of event data represent a separate software event. Why? The HYPERMON hardware embodies no notion of event types or lengths. Thus, each event data nibble must be assigned a 32-bit timestamp.

To avoid the expense of separate timestamp hardware for each node, a single timestamp generator is used. For each generated strobe vector, a timestamp also is stored in the timestamp FIFO. In general, the choice of timestamp resolution need not be dependent on the event strobe synchronization period. However, in the current ECB hardware, they are equal.

The design motivation for strobe vector and timestamp unification is to reduce hardware complexity. As a design alternative, each event nibble, a 4-bit node identifier, and a 32-bit timestamp could be sent to the I/O node. However, in this approach, 80 percent of the information would represent timestamp data. Moreover, for events occurring within the same time 800 nanosecond window, the timestamps for these events would be redundant. Instead, we adopted a strategy that packages event data as *event frames*. These event frames are the unit of transfer to the I/O node.

In the event frame approach, the strobe synchronization period is the time basis for event frame construction. Each such event frame consists of four 32-bit words; see Figure 5. Four event data bits from each node FIFO always are placed in the frame. However, only those FIFO's with valid data, as determined by the strobe vector, for this time window, will be shifted into the event frame; the other event data fields in the frame are undefined. The strobe vector is recorded as part of the frame to identify the valid event data fields. Finally, the corresponding 32-bit timestamp is saved with each frame. Once an event frame is constructed, it is saved in the ECB's frame FIFO. To eliminate useless data, an event frame is constructed for a time window only if at least one of the nodes produces an event during the window.

If we compare separately transferring each event nibble to the use of event frames, it is clear that when there is only one valid event nibble in a frame, the overhead is substantial. In this case, only three percent of the frame is data. Only when four or more nodes have valid event data will transfer of event frames require fewer bits. The motivation for merging event data from multiple nodes is that the efficiency of event transfer is more important when more nodes are producing event data. In this case, the likelihood of multiple nodes producing event data within the same time window increases, and the ratio of valid event data to overhead also increases. When few nodes are generating event data, the need for efficient transfers to the I/O node is not as great.

onds or 200 nanoseconds through jumpers on the ECB. This allows faster event generation rates to be accommodated in the future.

3.3 Event Processing

The ECB supports a parallel bus interface (the PBX bus) to an iPSC/2 I/O node. Via this bus, event frames can be transferred to the I/O node's memory. These PBX bus transfers are mapped through the I/O node's memory space. In addition, the ECB provides a writable 8-bit control register to reset the board. There is a 4-bit status register used to signal error conditions, most often FIFO overruns. The number of event frames generated since the last reset is accessible through a 12-bit frame count register. Finally, the event frame can be accessed using a single frame FIFO address. Referencing this PBX address will transfer one 32-bit frame word to the I/O node.

Once in the I/O node's memory, event frames can be decomposed into separate event streams for each instrumented node. Additionally, the I/O node's processor can be used to compress the event trace by computing statistics directly from the event data. Finally, the event trace data can be stored on the I/O node CFS disks for post-mortem analysis or transferred to the iPSC/2 host and associated workstations for analysis and presentation.

The ability to record trace data on local CFS disks or on remote disks attached to either the iPSC/2 host or a workstation, coupled with real-time or deferred trace analysis, provides a wide variety of trace storage and analysis configurations with distinct costs; see Figure 6. Selection of a trace processing mode depends on event frequency, density, and complexity. If the mean time interval between valid event frames is small, and we have observed that this often is true for operating system event traces, real-time event processing (e.g., statistical analysis) may not be possible — the I/O node minimally must record event data without loss.

4 Current Configuration

The HYPERMON prototype only recently became operational. It is implemented as a wire-wrapped, two-board set consisting of 115 integrated circuits — 20 on the ERB and 95 on the ECB. The experimental re-

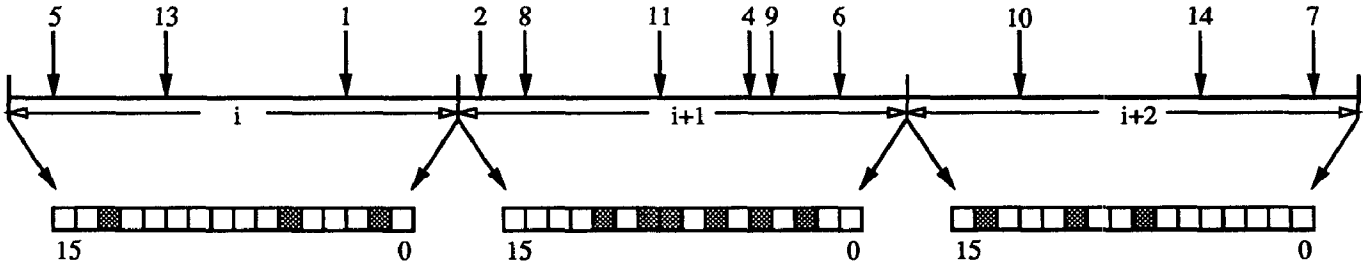


Figure 4: Strobe Vector Example

Event 15	Event 14	Event 13	Event 12	Event 11	Event 10	Event 9	Event 8
Event 7	Event 6	Event 5	Event 4	Event 3	Event 2	Event 1	Event 0
Unused				Strobe Vector			
Timestamp							

Figure 5: Event Frame Format

sults reported in this paper were obtained with eight nodes. We currently are testing HYPERMON with sixteen nodes.

Figure 7 shows the current HYPERMON configuration. At present, we are using a PBX-equipped node processor to communicate with HYPERMON. Disk access must be done through the iPSC/2 host using the hypercube message communication links. Upon addition of an I/O node with PBX support, we will be able to test I/O transfer to CFS storage.

5 HYPERMON Evaluation

The value of the HYPERMON design can only be determined through experiments with real applications and software instrumentation support. Architecturally, HYPERMON has the advantages of external event capture and real-time data access, but this must be weighed against the substantial overheads in event production; see §3.1.

In the remainder of this section, we describe the results of a set of experiments conducted with the HYPERMON prototype. First, we used a series of synthetic benchmarks to measure the raw data bandwidth of the current configuration. Here, the goal was to determine potential data recording bottlenecks. Second, we used a software instrumentation [8] of the Intel iPSC/2 NX/2 operating system as a source of event data for HYPERMON. This instrumentation generates a detailed event trace of operating system and application program interactions. The results of these experiments are discussed below.

5.1 Bandwidth Tests

A cursory examination of the HYPERMON architecture suggests several points where measurement of raw bandwidth might reveal fundamental performance constraints. Below, we examine three: event generation from the node processors, ECB internal event frame construction, and PBX event frame transfers.

Given the interface constraints on the individual iPSC/2 nodes (software event strobing, a three-fold increase in access time to an I/O port, and software extraction of event nibbles), the overhead to send event data to the HYPERMON ECB is substantially greater than that needed to record event data in a node's memory. To quantify this overhead, we began our tests with a simple, synthetic benchmark that generated events at the maximum possible rate on a single node. With software event recording, 3.9 seconds were needed to produce 1,000,000 events (assuming four bytes per event). This translates to a maximum software event recording rate of 1.02 Mbytes/second. In contrast, HYPERMON recorded an equal amount of event data in 88.5 seconds, a hardware data recording rate of 45 Kbytes/second.[4] Simply put, the Intel iPSC/2 interface to HYPERMON transfers data at a rate roughly 22.7 times less than that for software recording. Although the timestamp generation is done automatically by the HYPERMON hardware, this savings in data transfer is greatly overshadowed by the costs of nibble extraction, software event strobing, and the use of an I/O instruction rather than a memory MOVE operation.

Internally, the HYPERMON ECB can sustain a high

[4] Recall that the total amount of data recorded by HYPERMON is much larger and includes the strobe vectors and invalid data nibbles in each event frame.

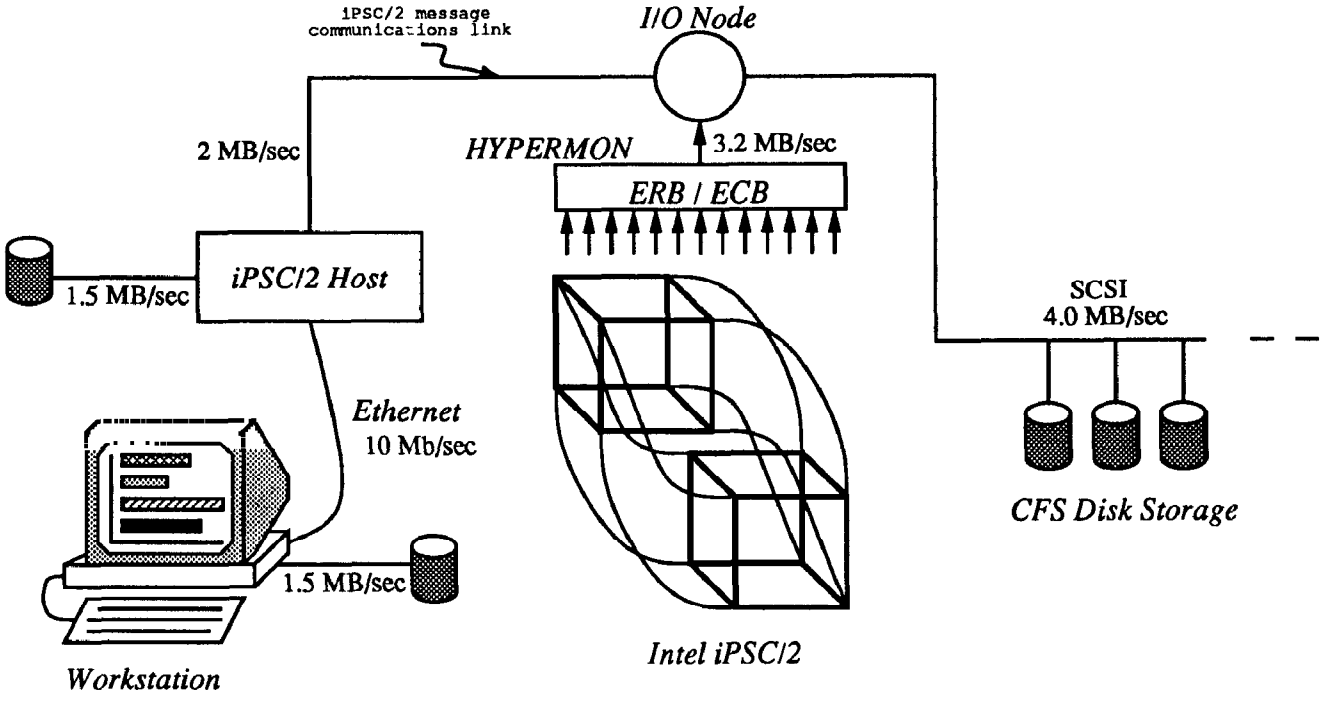


Figure 6: Event Trace Processing/Storage Options

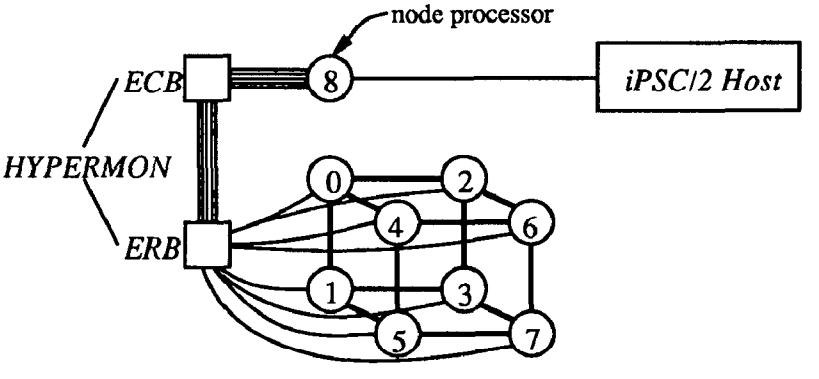


Figure 7: Current HYPERMON Configuration

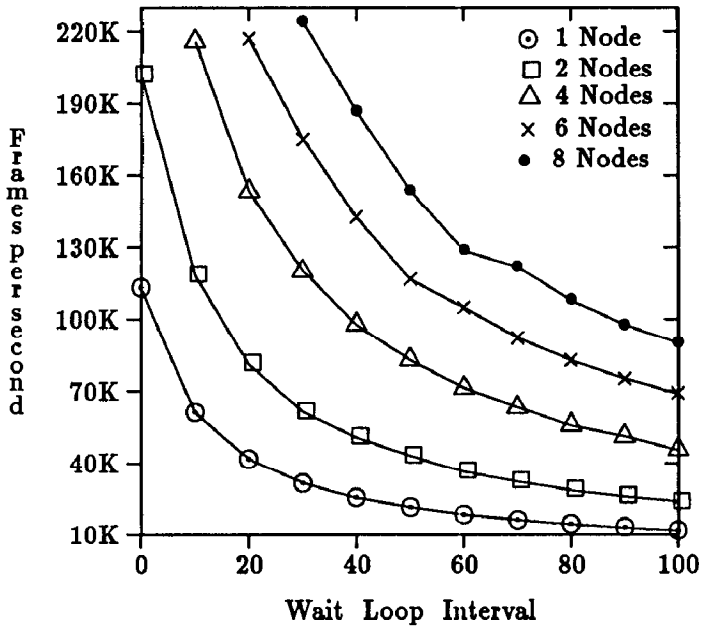


Figure 8: HYPERMON Bandwidth Results

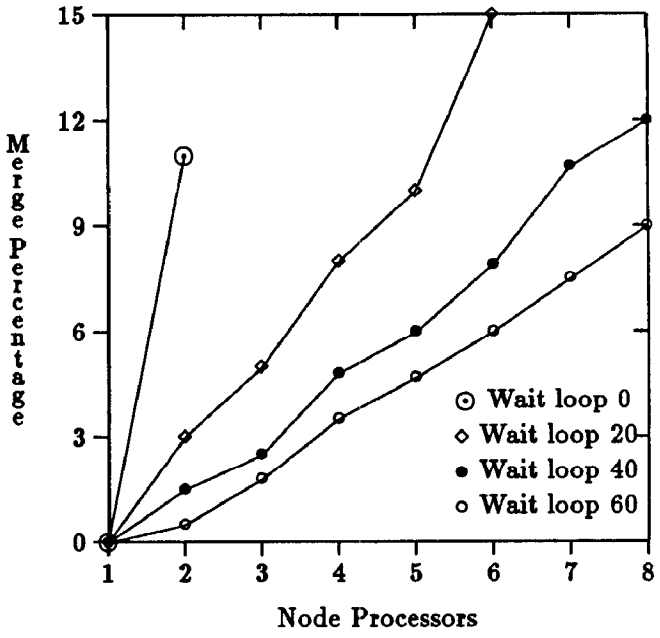


Figure 9: Event Frame Merging

event frame production rate. This value can be calculated directly from the internal ECB timing. The finite state machine controlling event frame generation operates at 20 MHz and requires eight cycles to build a frame. When all sixteen nibbles of event data in a frame are valid, this translates to a peak event data bandwidth of 10 Mbytes/second. As the number of valid data nibbles decreases, the event bandwidth decreases proportionally.

Finally, an I/O node must retrieve the event data from the ECB via the PBX bus. The PBX is an asynchronous, master-slave bus with a peak bandwidth of 18 Mbytes/second. However, when synchronization, address decoding, and data enabling overheads are included, the potential transfer rate across the PBX interface drops to 8 Mbytes/second. In practice, the software overheads for PBX transfers to the I/O node (the master) degrade bandwidth performance further. Using optimized PBX interface software, we have achieved PBX transfer rates of 5 Mbytes/second to an I/O node.

To assess the interactions of the bandwidth limitations just described, we constructed a synthetic benchmark that generates a specified volume of event data for a user-selectable number of nodes. Figure 8 shows the maximum experimental bandwidths obtained via HYPERMON when running this benchmark. In the benchmark, a delay parameter controls each node's event nibble generation rate; this delay parameter is the number of iterations of a null wait loop on the 16 MHz 80386. As expected, when the delay interval decreases, the bandwidth through HYPERMON increases.

When only one node generates event data, every four bits of event data forces the creation of a separate event frame. A single node can generate significant frame bandwidth (113,000 frames/second, or equivalently, 1.8 Mbytes/second at sixteen bytes per frame). However, when two nodes produce event data at their maximum rates, the PBX bandwidth limits are challenged (202,000 frames/second or 3.2 Mbytes/second). Beyond two nodes, the ECB reports a frame FIFO overrun condition, and no experimental data can be obtained unless nodes delay the creation of successive event nibbles.

We indicated earlier that HYPERMON was designed to be more efficient as the demand for event data bandwidth increased. This requires increased frame merging and amortization of the strobe vector and timestamp overhead over move valid event nibbles (i.e., the event frame creation rate should increase sublinearly with the number of nodes that simultaneously generate event data). Unfortunately, our experimental results suggest that the event data bandwidths must be very high to achieve effective merging. Figure 9 shows the percentage of merged event data as a function of the number of active nodes. Although the figure suggests that a high merging percentage should occur with high event data rates, merging does not increase quickly enough to offset the increase in the total event frame bandwidth seen at the PBX interface.

Although the high event data rates that cause PBX bandwidth saturation (evidenced by frame FIFO overruns) are beyond the sustained rate requirements for the HYPERMON design, such conditions can exist during event data bursts; see §6. Overrun conditions depend on the length of these bursts relative to the size of data buffers in the ECB. Our prototype implementation limits the frame FIFO size to 1K frames. As our results below suggest, this buffer size makes HYPERMON sus-

Code	Description
Simplex	a parallel linear optimization program based on a column-wise Simplex algorithm [11]
Life	a parallel implementation of Conway's famous cellular automaton [4]
MatMul	a simple distributed memory matrix multiplication code
Place	a standard cell placement algorithm based on simulated annealing [2]

Table 1: Application Test Programs

ceptible to event data bursts of moderate duration.

5.2 Monitoring Real Applications

Ultimately, the software instrumentation level required to capture the execution behavior of a parallel computation will dictate the frequency and volume of data that must be recorded by the monitoring system. Not surprisingly, certain instrumentation levels can exceed the capabilities of any data recording system. Event processing and analysis requirements impose additional constraints on feasible performance experiments. As a consequence, performance observability mandates a balance between the rate of event data generation and analysis, and the fundamental limitations of the monitor's operation.

To understand HYPERMON performance in an instrumentation environment for real applications, we compared the execution of four programs in three monitoring environments: no event data generation (raw), software event recording to node memory, and hardware event recording with HYPERMON. The execution data for the latter two cases was restricted to that captured by an instrumentation of the iPSC/2 NX/2 operating system source code [9]. This system generates three classes of operating system events: message transmissions, process states transitions including context switches, and system calls. Below, we describe the characteristics of the application programs, the observed performance data, and HYPERMON's performance.

5.2.1 Execution Statistics

Table 1 shows the four application programs used in our study of data capture for operating system instrumentation. Each of these applications was run on 1, 2, 4, and 8 nodes for each of the three data capture scenarios. To prevent event data bursts that might saturate the PBX bandwidth and overrun the event frame FIFO, we set the delay interval for hardware event recording to twenty (i.e., writing of successive event nibbles was separated by twenty iterations of a null loop).

Not surprisingly, Table 2 shows that hardware data recording using HYPERMON consistently causes greater perturbations than software recording in the node memories. This was expected from our earlier analysis of performance penalties imposed by the node interface to HYPERMON. However, the degree of perturbation differs for each application program and number of nodes. Simply put, differences in program behavior are manifest as differences in the time varying demands placed on the data recording system. For instance, the minor perturbations of the *MatMul* code contrast sharply with the substantial slowdown of the *Place* code when data are recorded from eight nodes with HYPERMON. Unlike matrix multiplication, which generates only a small number of instrumentation events, the cell placement code is highly dynamic and the variance in its event generation rate is high; see §6.

Table 2 further shows that the difference in total data volume between software and hardware data recording is large. With software event recording in individual node memories, each recorded event includes the associated data and a 16-bit timestamp delta. In contrast, the total data volume recorded using HYPERMON is calculated from the total number of event frames transferred across the PBX interface. With software data recording, only one timestamp is assigned to a multiple byte event; HYPERMON must timestamp each data nibble.

Clearly there is greater overhead with hardware data recording, but one might expect merging to increase the efficiency at higher event rates by amortizing timestamps across multiple event nibbles. Unfortunately, the software recording rates indicate that the potential for merging is small. Thus, we conclude that the large difference in the volume of recorded data reflects the fact that most event frames contain only one valid event data field.

Even when each node delays for twenty iterations of a null loop between transfer of event data nibbles, data overruns occur in real applications (e.g., the eight node *Place* execution). Although the sustained hardware recording rate of 2.5 Mbytes/second for this program (as extrapolated from the software recording rate) does not exceed the PBX bandwidth, the burst event rate is higher. In this case, the only alternative is to increase the interval between the output of successive event nibbles.

5.2.2 Dynamic Monitoring Requirements

As just noted, sustained recording rates do not reflect instantaneous demands on the monitoring system. Understanding the dynamics of event creation is important

Application	Total Time (seconds)			Logged Data (bytes)		Logging Rate (bytes/second)	
	Raw	Soft	Hard	Soft	Hard	Soft	Hard
Simplex							
1 node	151.059	151.261	154.838	63437	2302144	419	14868
2 node	77.850	78.396	86.524	417476	10273040	5325	118730
4 node	42.068	42.561	54.338	1245712	27763296	29269	510937
8 node	24.277	24.931	41.045	3190873	67493104	127988	1644368
Life							
1 node	152.222	152.335	154.157	34103	1204608	224	7814
2 node	111.869	112.274	120.280	277359	10155584	2470	84433
4 node	78.505	79.112	93.048	752373	27607776	9510	296705
8 node	43.981	44.741	58.202	1670188	60352768	37330	1036954
MatMul (256x256)							
1 node	245.660	245.840	248.808	54706	1928096	223	7749
2 node	123.118	123.210	124.696	55215	1947792	448	15620
4 node	61.975	62.045	62.677	56277	1985376	907	31676
8 node	31.369	31.423	31.859	58456	2069136	1860	64947
Place							
1 node	318.012	318.473	322.325	74225	2628192	233	8154
2 node	82.000	85.071	143.356	2018749	74003616	23730	516223
4 node	69.668	72.851	137.145	4147564	149195680	56932	1087868
8 node	38.498	41.427	102.221	7977226	(overrun)	192561	(overrun)

Table 2: HYPERMON Application Results

for two reasons. First, it suggests where buffers in HYPERMON are most needed to ameliorate the effects of event data bursts. Second, it identifies those portions of a parallel computation where program execution might be most susceptible to performance perturbation from performance instrumentation.

For each of the four application programs of Table 1, we used HYPERMON to record and compute statistics on the time varying rate of operating system event generation. For each application program, we recorded in the PBX node's memory the number of event frames and the elapsed time between the first and last frame for each group of event frames read.[5]

To generate Figures 10–13, the elapsed time for each program was divided into one hundred intervals of fixed size and the average event frame rate was computed for each interval. To show differing numbers of nodes on a single graph, we show normalized time intervals (i.e., for each number of nodes, an interval represents a different absolute amount of time). The total time range for each curve is shown in the legend.

Figure 10 shows the time varying event frame rate for the *Simplex* code. Clearly, the event frame rates follow a periodic pattern, and analysis of the code shows a regu-

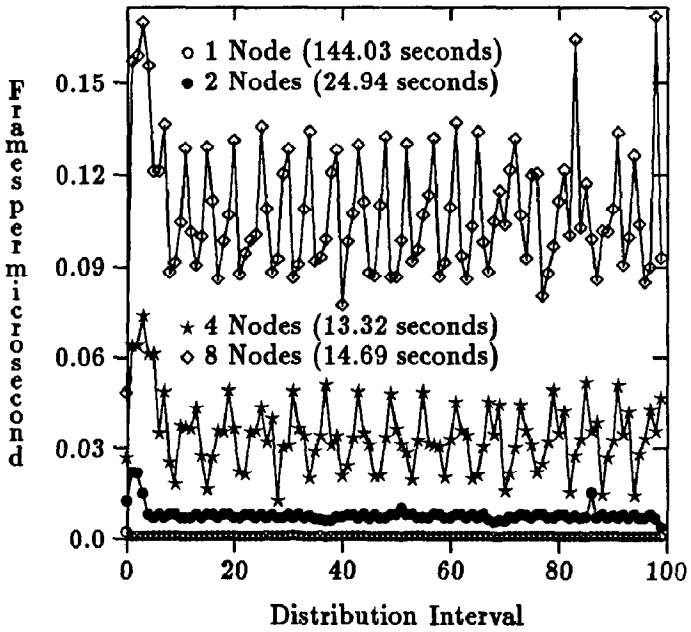


Figure 10: *Simplex* Event Frame Rate Distribution

[5]Recall that the HYPERMON interface to the PBX I/O node includes a counter of the number of buffered event frames. This count defines the number of events read in each "group." Due to PBX node memory limits, only the first 100,000 event rate distribution samples were recorded.

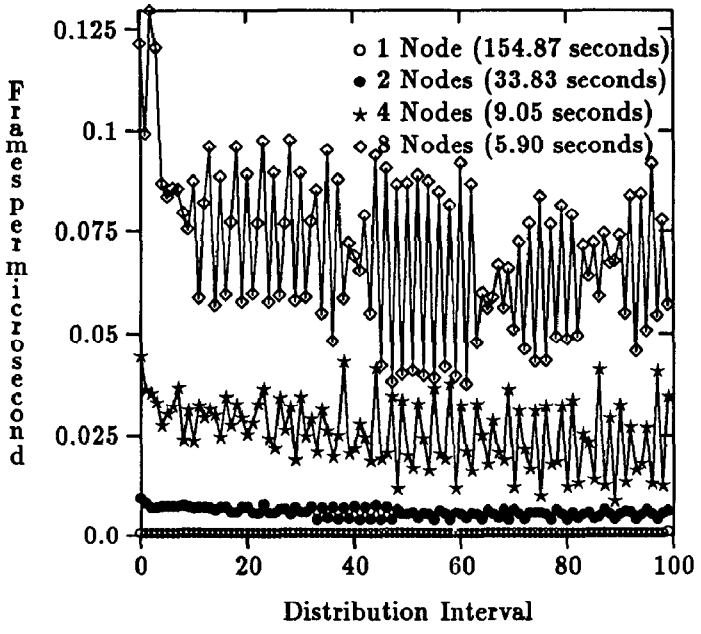


Figure 11: *Life* Event Frame Rate Distribution

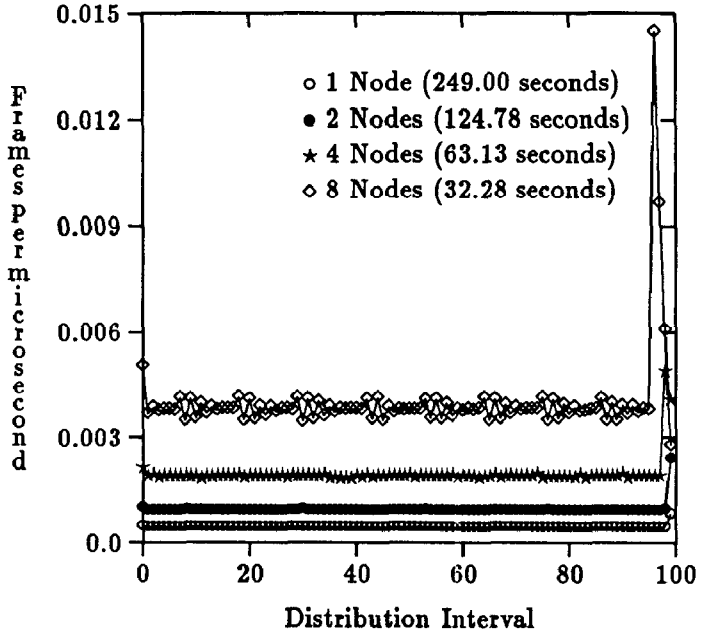


Figure 12: *MatMul* Event Frame Rate Distribution

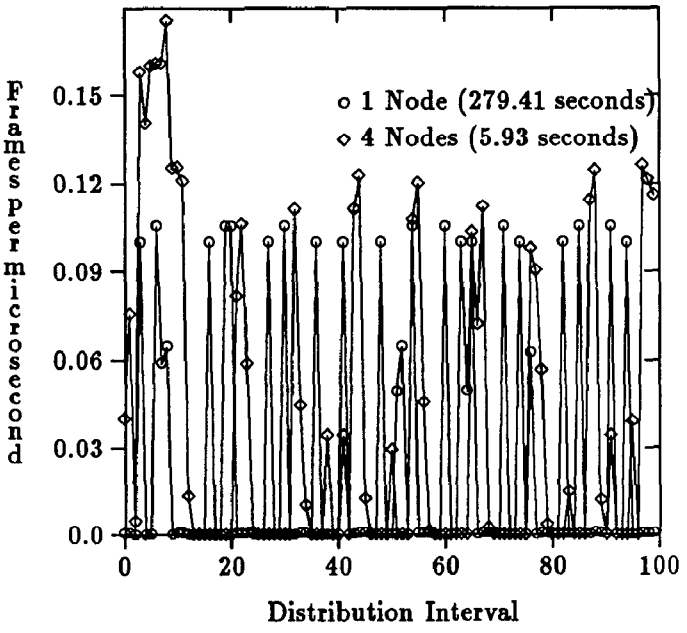


Figure 13: *Place* Event Frame Rate Distribution

lar cycle of computation and global data exchange [11]. Communication generates a burst of operating system instrumentation events (e.g., context switches, message buffering, and message transmission) [6], and this is reflected in the event frame rate. As the number of nodes increases, the ratio of communication to computation increases and the event frame rate increases commensurately.

Unlike the *Simplex* or *MatMul* codes, where the amount of computation on each node varies little during successive computation cycles, the *Life* code updates a grid of cells whose sparsity changes over time. Figure 11 shows that the data recording requirements of such codes can change substantially as the computation load balance changes.

The *MatMul* event frame rate distribution in Figure 12 reflects the simple structure of the application code. The computation first distributes the matrix to the nodes, where they compute independently until returning their partial results to the host. The initial matrix broadcast is not shown in Figure 12, but the transmission of sub-matrices to the host is clearly visible. Because no communication occurs during the computation phase, most recorded events are node time slice context switches. Finally, the event rates for *Place* application, shown in Figure 13, are random, bursty, and high. In §6, we describe the underlying reasons for this behavior and the implications for hardware event data recording.

As Figures 10–13 show, the dynamics of event frame rates are closely tied to application behavior and can vary widely across application types. This disparity in burst rates has important implications for capture hardware design, the subject of the next section.

6 Lessons Learned

The design of HYPERMON was subject to the engineering constraints imposed by the iPSC/2 system: the 4-bit I/O event data interface, the physical separation of event regeneration from event capture, and the PBX I/O node interface. Although the larger overhead for recording event data via HYPERMON was expected, and we knew that many applications exhibited cyclic communication behavior, we did not foresee all the implications of bursty event data rates.

The lesson regarding decreased execution time perturbations with hardware data recording is clear. External interfaces used to record event data via hardware should have sufficient bandwidth to avoid delaying the computation processors. Ideally, the access time to the interface should be no larger than that needed to write the event words to memory (i.e., hardware event recording should have less overhead than that for software buffer management and data recording).

Regarding bursty event data rates, further investigation of event data bursts using software event traces reveals significant variances in event data rates during the lifetime of most computations. Figure 14 shows the event data volume generated by our NX/2 operating system instrumentation [8] in one millisecond intervals for the *Place* application on four, eight and sixteen nodes. Although the average event data rates are 82 Kbytes/second, 276 Kbytes/second, and 629 Kbytes/second, respectively, event data bursts significantly exceed these rates. In particular, the data rate for the sixteen node *Place* execution can reach two to three Mbytes/second in twenty millisecond bursts. To support this type of software performance instrumentation, a hardware data recording system must be designed with sufficient buffer capacity to accommodate event data bursts. Analysis of software event traces can be instrumental in defining buffer requirements. At present, we are using the software traces as input to simulation models of monitor designs to understand dynamic buffering requirements.

An important decision in the HYPERMON design was to treat each 4-bit event datum as a potentially unique event. This determined timestamp generation and motivated the notion of event frames to amortize timestamp overhead. In practice, our NX/2 operating system instrumentation produced logical events composed of multiple event data nibbles. Significant reductions in the volume of data recorded by HYPERMON would have been possible had we chosen to timestamp larger data units (e.g., 32-bit quantities). In this case we would accumulate a 32-bit word on each 4-bit input port before storing it in the event data FIFO. The size of timestamped quantities should be chosen so that only a small fraction of the available bandwidth is lost. Ideally, there should be support for selective timestamping of event data such that timestamps are produced only when directed by the software.

The experiments conducted with the instrumented NX/2 operating system, described in §5.2, represent HYPERMON stress tests. Clearly, there exists a spectrum of data recording and data analysis alternatives. No reduction of event data occurred in our experiments prior to writing data to HYPERMON. The use of parallel, on-the-fly data reduction, possibly in the form of periodic statistical summaries, would eliminate many of the problems encountered during our stress tests of HYPERMON operation. Although improvements in the HYPERMON design can extend its operational range, there are many performance experiments that can take advantage of the current prototype's real-time monitoring capabilities.

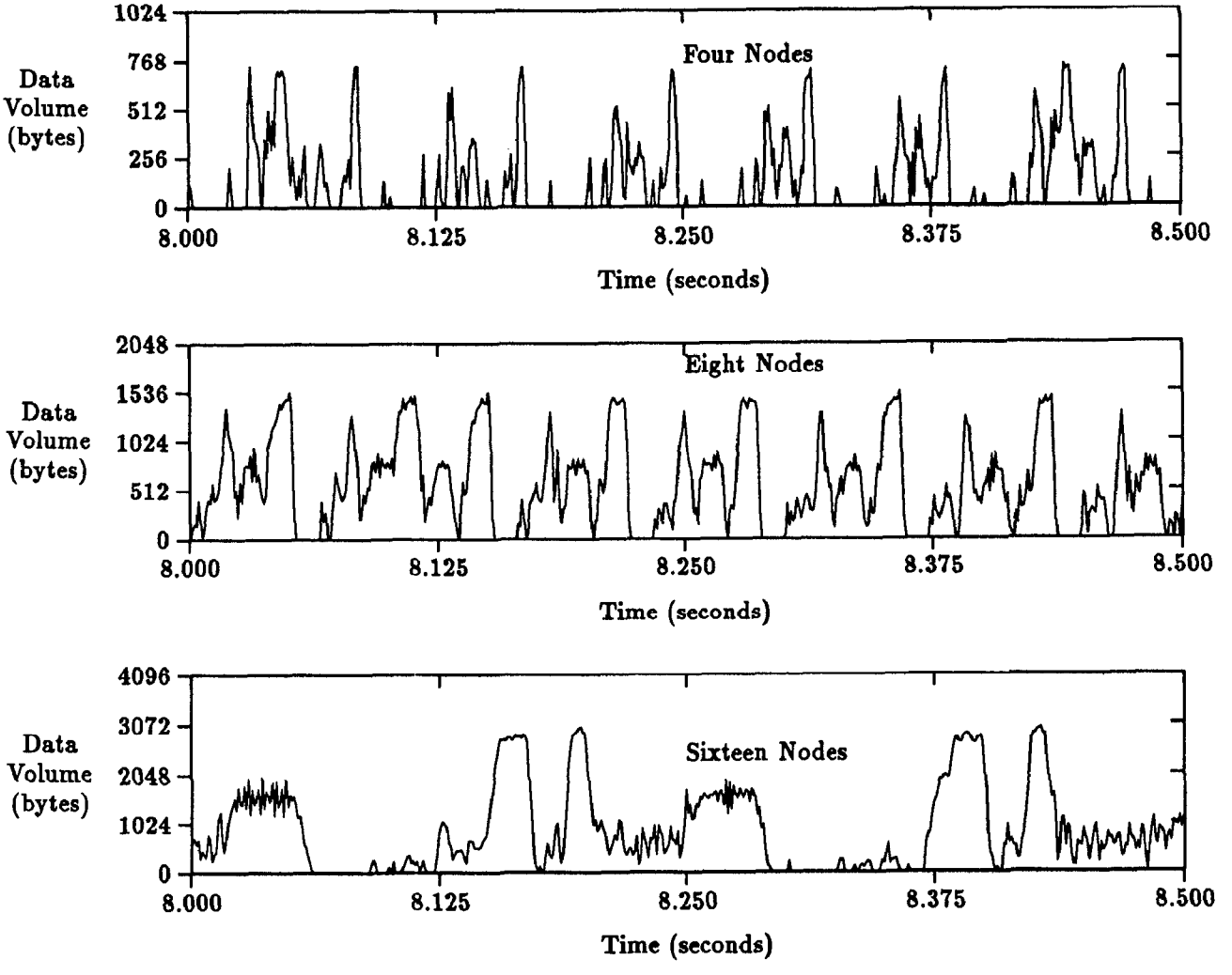


Figure 14: Place Event Data Volume (One Millisecond Intervals)

7 Conclusions

Despite the manifest need for dynamic performance instrumentation and data capture, their efficient implementation is non-trivial. HYPERMON was designed in response to the iPSC/2 hardware interface for capturing software event traces. In contrast to software-based recording in individual node memories, HYPERMON uses external memory for trace storage and generates globally-synchronized timestamps automatically.

In addition to the considerable development effort for the HYPERMON prototype, our initial experience clearly indicates the need for careful analysis of the interactions with the iPSC/2's hardware interface. For example, the 4-bit I/O interface from each node has obvious performance limitations; only a wider I/O port will alleviate the instrumentation perturbations when HYPERMON is used.

The experiments conducted with the instrumented NX/2 operating system, described in §5.2, represent HYPERMON stress tests. The spectrum of data recording and data analysis alternatives is vast. The use of parallel, on-the-fly data reduction, possibly in the form of periodic statistical summaries, rather than the detailed operating system performance instrumentation used in our stress tests, seems the best match to the 4-bit I/O interface and HYPERMON's buffer requirements.

Acknowledgments

Justin Rattner (Intel Scientific Computers) first suggested implementing a performance monitor via signals from the iPSC/2 backplane. Since that time, Paul Close (Intel Scientific Computers), has provided technical information and support. Without their help, the design of the iPSC/2 hardware monitor would not have been possible.

References

[1] R. Arlauskas. iPSC/2 System: A Second Generation Hypercube. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, Volume I*, pages 38–42, Pasadena, CA, January 1988. Association for Computing Machinery.

[2] R. J. Brouwer and P. Banerjee. A Parallel Simulated Annealing Algorithm for Channel Routing on a Hypercube Multiprocessor. In *Proceedings of the International Conference on Computer Design*, pages 4–7, Rye Brook, NY, October 1988.

[3] Paul Close. The iPSC/2 Node Architecture. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, Volume I*, pages 43–50, Pasadena, CA, January 1988. Association for Computing Machinery.

[4] Martin Gardner. Mathematical Games. *Scientific American*, pages 120–123, October 1970.

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

[6] Allen D. Malony, Daniel A. Reed, James W. Arendt, Ruth A. Aydt, Dominique Grabas, and Brian K. Totty. An Integrated Performance Data Collection Analysis, and Visualization System. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 229–236, Monterey, CA, March 1989. Association for Computing Machinery.

[7] Allen D. Malony, Daniel A. Reed, and Harry Wijshoff. Performance Measurement Intrusion and Perturbation Analysis. Technical Report CSRD No. 923, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, October 1989.

[8] Daniel A. Reed and David C. Rudolph. Experiences with Hypercube Operating System Instrumentation. *International Journal of High Speed Computing*, 1990. to be published.

[9] D. C. Rudolph and Daniel A. Reed. CRYSTAL: Operating System Instrumentation for the Intel iPSC/2. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, pages 249–252, Monterey, CA, March 1989.

[10] David C. Rudolph. Performance Instrumentation for the Intel iPSC/2. Master's thesis, University of Illinois at Urbana–Champaign, Department of Computer Science, July 1989.

[11] C. B. Stunkel and D. A. Reed. Hypercube Implementation of the Simplex Algorithm. In *Proceedings of the Third Conference on Hypercube Computers and Concurrent Applications*, pages 1473–1482, Pasadena, CA, January 1988. Association for Computing Machinery.