An Integrated Performance Data Collection, Analysis, and Visualization System

Allen D. Malony*
Daniel A. Reed†

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

Ruth A. Aydt† James W. Arendt†
Dominique Grabas† and Brian K. Totty†

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

Abstract

The lack of tools to observe the operation and performance of message-based parallel architectures limits the user's ability to effectively optimize application and system performance. Performance data collection, analysis, and visualization tools are needed to manage the complexity and quantity of performance data. Furthermore, these tools must be integrated with the machine hardware, the system software, and the applications support software if they are to find pervasive use in program development and experimentation.

In this paper, we describe an integrated performance environment being developed for the Intel iPSC/2 hypercube. The data collection components of the environment include software event tracing at the operating system and program levels plus a hardware-based performance monitoring system used to unobtrusively capture software events. A visualization system, based on the X window system, permits the performance analyst to browse and explore interesting data components by dynamically interconnecting new performance displays and data analysis tools.

1 Introduction

Despite continued technical advances, parallel system design remains *ad hoc*, an art form practiced by a small cadre of experienced, highly valued designers. No known, general purpose methods can predict the performance of a proposed system design. Moreover, seemingly minor perturbations of parallel architecture, system software, or application algorithms can induce large changes in observed performance. In numerical analysis, such problems are called ill-conditioned — small changes in the input x of a function $f(x)$ yield large changes in $f(x)$ (i.e., $f(x+\epsilon) \not\approx f(x)$). Clearly, approximating $f(x)$ by $f(\overline{x})$, by $f(\underline{x})$, or by $\max_x f(x)$, is dangerous, if not wrong. Yet, peak performance ratings in MIPS (millions of instructions per second) or MFLOPS (millions of floating point operations per second) are precisely such an approximation.

In reality, the performance of a parallel computing system is the complex product of its component interactions. A complete performance analysis requires both *static* and *dynamic* characterizations. Static or average behavior analysis may mask transients that dramatically alter system performance. By analogy, biological researchers have long recognized the importance of both *in vitro* and *in vivo* measurements. Laboratory measurements of isolated cells or biological molecules often differ from similar measurements in natural environments. Similarly, the performance of parallel system components depend on the frequency and types of their interactions; these interactions often cannot be predicted, but they can be measured.

Despite the manifest need for dynamic performance instrumentation and data capture, its efficient implementation is non-trivial. Instrumentation, no matter how unobtrusive, introduces performance perturbations, and the degree of perturbation is proportional to the fraction of the system state that is captured — volume and accuracy are antithetical. The degree of uncertainty manifested depends on the programming paradigm, the system software, and the underlying hardware. Some systems are more conducive to instrumentation than others.

Message passing systems, such as the Intel iPSC/2, pose particularly acute instrumentation problems. First, detectable events occur locally at each processor. Identifying global events requires associating two or more events from different processors. The obvious approach imposes a total order on events, based on event timestamps, and identifies global events from temporally proximate event groups. Unfortunately, the second problem complicates solutions to the first: most message passing systems, including the Intel iPSC/2, lack a globally synchronized clock to create event timestamps — this clock is needed to maintain event causality. Given a global time reference, the distributed event data still must be collected for analysis and presentation.

Given the existence of a minimally intrusive performance instrumentation system, a message passing system can quickly generate vast quantities of performance data. This data must be presented in ways that emphasize important events while eliding irrelevant details. Just as visual presentation of scientific data can provide new in-

*Supported in part by NSF Grants NSF MIP-8410110 and NSF DCR 84-06916, DOE Grant DOE DE-FG02-85ER25001, Air Force Office of Scientific Research Grant AFOSR-F496200, and a donation from IBM.

†Supported in part by the National Science Foundation under grants NSF CCR86-57696, NSF CCR87-06653 and NSF ANTI TAPESTRY 1-5-30035, by the National Aeronautics and Space Administration under NASA Contract Number NAG-1-613, and by donations from Digital Equipment Corporation and Central Data Corporation.

sights, a performance visualization system would permit the performance analyst to browse and explore interesting data components by dynamically interconnecting new performance displays and data analysis tools.

To provide insight into dynamic system performance, we are developing an integrated data collection, analysis, and data visualization system for the Intel iPSC/2 hypercube. In §2, we begin with a system overview. The data collection components of the environment, discussed in §3 and §4, include software event tracing at the operating system and program level plus a hardware-based performance monitoring system used to unobtrusively capture software events. In §5, we describe a visualization system, based on the X window environment, that permits dynamic display and reduction of performance data. Finally, §6 summarizes our experience and development plans.

2 Environment Organization

Integration and flexibility are the twin keys to an effective performance analysis environment. If the interactions among environment components are awkward or inefficient, the performance analyst will seek simpler tools. Similarly, if the environment does not permit diverse approaches to performance data reduction and analysis, including addition of new environment components (e.g., data filters and displays), its functional lifetime will be limited. Given the implications of integration and flexibility and our experience with an earlier environment design [1], we established several specific environment design goals. Not all of these goals were simultaneously realizable, but they provided a framework for system design.

- The individual analysis and visualization components should be **easy to build** for many different application types and system software environments.
- The environment should be **fast**, preferably fast enough to process bursts of real-time data.
- It should be possible to **dynamically configure** data analysis and visualization components, allowing the performance analyst to change data perspectives during execution.
- Finally, the environment should be **portable** to different systems. Although the mechanisms for performance data capture are inherently system dependent, performance data reduction (e.g., computation of sliding window averages) and visualization are largely system independent.

Given these design goals, Figure 1 shows the organization of our performance environment. The environment's fundamental performance measure is an *event*. Given a "complete set" of event types, a timestamped event trace suffices to construct general performance measures. As the figure suggests, our performance instrumentation includes event tracing at both the program and operating system levels.

At the program level, the performance analyst can direct a modified version of the GNU C compiler to automatically generate code to create a timestamped log of procedure entries and exits; this requires no modification to the application source code. Additional program performance events can be generated by inserting calls to event tracing routines; these include marking the entry and exit to code sections and marking the occurrence of a user specified event [2, 3]. In our current implementation on the Intel iPSC/2, all application program events are passed to the hypercube operating system NX/2. NX/2 has been modified to record both these application events and operating system events corresponding to message transmissions, process state changes, and system calls.

The events produced as a result of this instrumentation are transmitted from each node to the hardware monitor via additional signal lines on the iPSC/2 backplane.[1] Because the hardware monitor generates event timestamps and can accept simultaneous events from all nodes, causality is assured. The resulting event stream can be either stored in an event trace file or, if the event frequency is low enough to permit real-time processing, sent to a set of data filters and displays.

Clearly, the number of generated events is potentially enormous; the event data must be presented in ways that emphasize important primitive events (i.e., those generated during execution) and that reflect aggregate system behavior (i.e., by synthesizing compound events). The environment includes a set of data filters that process the event data, either by eliding irrelevant events or by computing dynamic statistics (e.g., sliding window averages).

Given the diversity of performance data and possible statistics, a variety of performance displays (including meters, plots, histograms, event graphs, dynamic call graphs, and topological views) are needed to display the dynamics of system performance. These displays are implemented as X window widgets [4], providing display portability across a variety of vendor workstations. Because the display filters isolate the semantics of performance data, the same data can be displayed in multiple ways (e.g., histograms and meters) without embedding the data semantics in each display. Via an environment control, the binding of data filter and display can be changed dynamically, allowing the user to select the filters and display formats best suited to the data. The remainder of this paper discusses the hardware and software implementation of this environment, including examples of its use.

3 Software Instrumentation

There are many levels in the hierarchy of performance instrumentation, including hardware, system software, and application program. The answer to the oft-asked question, "How fast is it?" depends on the intended use of the performance data. Operating system instrumentation can capture the interplay of hardware and system software, but it cannot identify the performance bottlenecks in an application program. Consequently, our instrumentation system provides both, correlating system and application performance.

[1] Until the hardware monitor is complete, these events are timestamped and recorded in the memories of individual nodes [3].

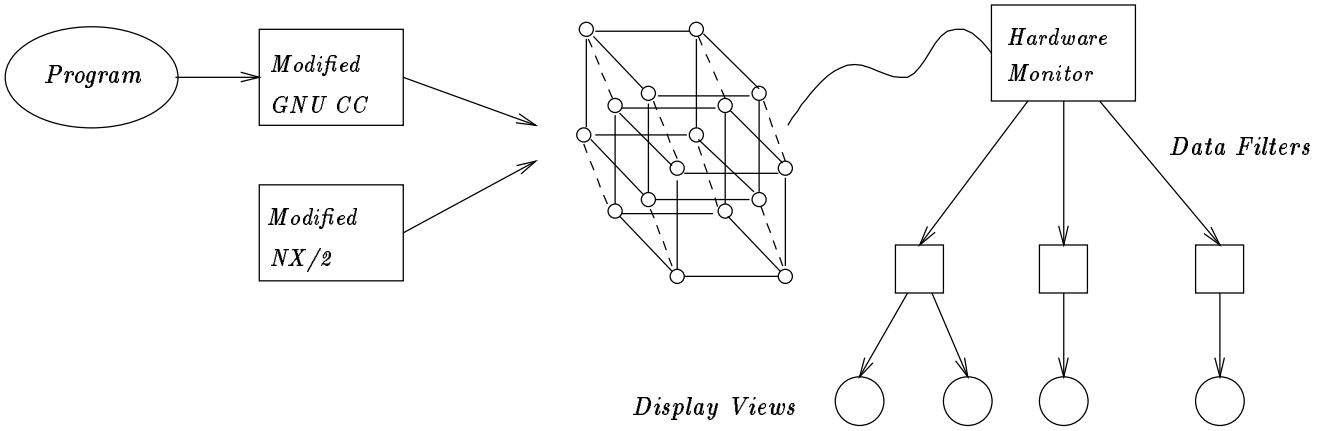

Figure 1: Environment Organization

3.1 Application Instrumentation

If extensive source code modifications are required to capture application performance data, analysis of many competing application algorithms becomes infeasible. Thus, the great appeal of compile time instrumentation is its flexibility and ease of use; no source code modifications are required, yet execution traces or profiles can be generated.

To capture application performance data, we have modified the Free Software Foundation's GNU C compiler [5] to emit instrumented code for the Intel iPSC/2. Not only is the source to the GNU C compiler readily available, it generates code whose quality is competitive with commercial compilers.[2]

Compiler command line options enable generation of instrumented code. If only selective instrumentation is desired, the user can specify either the list of procedures to be instrumented or, equivalently, those not to be instrumented. As each procedure is compiled, calls to monitoring functions are inserted in the procedure's prologue and epilogue. These calls are inserted in the intermediate RTL representation used internally by the compiler, not the generated assembly code. Thus, the instrumentation is machine independent, and given the requisite support, allows instrumentation of other machine architectures. At execution time, the monitoring functions invoked by each procedure's prologue and epilogue pass the procedure's name and entry or exit events to NX/2, the iPSC/2 node operating system. This creates a timestamped trace of procedure entry and exit events for each hypercube node.

If additional application trace events are needed (e.g., the occurrence of a condition or the execution of a particular code section), the user can manually insert calls to monitoring functions. Like the automatic instrumentation, these calls generate events that are passed to NX/2 and embedded in the event trace. From this trace, the visualization subsystem can construct program execution profiles and dynamic displays of program activity.

[2]Benchmarks show little difference between GNU C and the Greenhills C provided with the Intel iPSC/2.

3.2 Operating System Instrumentation

As just noted, application performance events are passed to NX/2. These application events are merged with three classes of operating system events: message, process, and system call. These operating system events are captured by extensive instrumentation of the NX/2 operating system source code [3]. For message transmissions, NX/2 captures the parameters of the application program's `csend` or `isend` call and the time that physical transmission of the message began and completed. NX/2 captures corresponding events for message receipt, permitting matching of message sends and receives on different nodes. In addition, the NX/2 instrumentation captures context switches, including the identity of the processes, and all system calls.

Because the hardware monitor is not yet operational, NX/2 currently records both application and operating system events in a portion of each node's memory. After the application program completes, the individual node traces are transmitted to the iPSC/2 host for post-processing and display. The hardware monitor will permit NX/2 to record each event by writing to an I/O port in the Intel 80386; the monitor will capture and timestamp each such write.

4 Hardware Monitoring

As mentioned at the outset, the lack of a global clock on the Intel iPSC/2 exacerbates the already large data capture problems that exist in a distributed memory system. To circumvent the data capture and event ordering problems, we are developing a hardware-based monitoring system, called HYPERMON, for the Intel iPSC/2. Below, we describe the components of the HYPERMON and HYPERMON's relationship to the remainder of the analysis and visualization system.

4.1 Hypermon Architecture

As Figure 2 suggests, the HYPERMON architecture includes four primary components: event capture, event processing, performance data storage, and a system interface. Each iPSC/2 node independently sends event data to HYPERMON via backplane connections. HYPERMON captures

Figure 2: HYPERMON Architecture

these events, generates global timestamps, and stores the resulting event data in internal memory buffers. Because HYPERMON contains two internal processors, preliminary analysis of the event data (e.g., format conversion) can occur at the point of capture. The resulting performance data can be stored on HYPERMON's disk or transferred via a network to a workstation for further analysis and presentation.

There are two major requirements for the HYPERMON system. First, HYPERMON must accept events from the hypercube nodes at their natural generation rate. The design guideline allows each node to generate an event every 100 microseconds on average, and supports bursts of up to 256 events with inter-event times as small as 10 microseconds. Second, HYPERMON must support the transfer and storage of event data. Assuming the representation of each event requires at most 10 bytes, approximately 1.5 MBytes/second of event data must be transferred to disk or across a network.

The prototype HYPERMON implementation is built on a MULTIBUS II platform; see Figure 2. All components are standard MULTIBUS II cards except the custom event capture board described in §4.3.

4.2 iPSC/2 Event Visibility

An Intel modification of the iPSC/2 hypercube makes possible external access to software events on each node processor. Five bits from an I/O port on each hypercube node board are routed via the system backplane to an external connector in the system cabinet. The ECB event capture board attaches to this connector. Up to 16 iPSC/2 nodes can be supported by one ECB.

Because the NX/2 operating system generates event data by writing to an I/O port, one bit must be reserved as a strobe to signal the ECB that valid event data are present. Thus, software event generation proceeds in two phases:

1. WRITE: Event.strobe = 0, Event.data = *undefined*
2. WRITE: Event.strobe = 1, Event.data = *new event data*

The 80386 microprocessor in the Intel iPSC/2 requires approximately six cycles to complete an I/O write operation. Thus, a minimum of 12 cycles are needed to write a software event. Up to sixteen events can be uniquely represented by four bits of data. If additional events are needed, or if data is associated with an event, multiple I/O write operations are required. The HYPERMON processor can reconstruct the software events from this data.

Although the four bit event field may seem limiting, hardware constraints limit the number of available backplane signals. Despite this apparent limitation, the potential software event rate remains substantial. Moreover, higher rates would distort system performance.

4.3 Event Capture

Figure 3 shows the functional design of the event capture hardware. The event data for each hypercube node is placed in a FIFO buffer that is clocked by the corresponding event strobe signal. All events occurring within a given *event time window*, defined by the rate of the *timestamp* clock (approximately 4 MHz), are combined to form an *event frame*.[3] Each event frame is then placed in the event frame FIFO for transfer across the MULTIBUS II to the HYPERMON processor or memory.

All events present within a time window are given the same timestamp. Because the event signals are generated from processors with asynchronous clocks, the event strobes for each node must be synchronized with respect to the window to determine event presence. An event frame is constructed for a time window only if one of the nodes produces an event during the window. The strobe signals are captured as part of the frame to indicate which nodes generated events.

Each event frame consists of four 32-bit words; see Figure 4. A 32-bit timestamp is saved with each frame. As mentioned above, the strobe vector identifies which of the event data are valid for the time window represented by the frame. Four event data bits from each node FIFO are always placed in the frame. However, only those FIFO's with valid data for this time window will be shifted into the event frame; the other event data fields in the frame are undefined. Once an event frame is constructed, it is saved in the ECB's frame FIFO.

4.4 Event Data Management

The ECB supports a simple interface to a frame multiplexing board that provides MULTIBUS II transfers to the processor and memory from (potentially) several ECB's; the prototype system has only one ECB.

The interface board contains an Intel 80186 microprocessor, 512K bytes of memory, a DMA controller and some

[3] The event data FIFO's provide data buffering during this process.

Figure 3: HYPERMON Event Capture

Unused		Strobe Vector	
Timestamp			
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

Figure 4: HYPERMON Event Frame

custom logic. Events are transferred via DMA from the ECB to the memory of the interface board. The interface board's processor provides the flexibility to do limited event preprocessing. Events are buffered on the interface board and transferred in blocks across the bus to the primary HYPERMON memory. Although this buffering increases the efficiency of bus transfers, a timeout mechanism is needed to insure that event transfers continue when the effective event arrival rate is low.

All events are accessible to the primary HYPERMON processor, a 20 MHz 80386 that supports Unix System V. This processor can be used to compress or preprocess the event trace. Finally, the HYPERMON architecture provides two external interfaces: a SCSI Winchester disk interface and a high-speed external channel. The disk interface includes a DMA control, established by the primary processor, to transfer data between the HYPERMON memory and disk. The disk is used primarily for storing event data. The high-speed external channel provides an interface from the HYPERMON system to a workstation. The channel design depends on the interconnect options available with the workstation; the prototype system uses Ethernet.

5 Data Analysis and Visualization

As discussed in §2, flexibility and dynamic reconfigurability were primary design goals for the data analysis and visualization environment. Thus, the environment infrastructure permits addition of new data analysis functions and data views.

5.1 Infrastructure Design

The data analysis and visualization system contains a *user interface*, an *event preprocessor*, a set of generic data analysis *filters*, a set of filter-display interfaces called *strainers*, and a set of display *views*. The event preprocessor converts the event stream produced by both the software instrumentation and the hardware monitor into a standard format for use by the event filters.[4]

The event filters accept the trace events and maintain an internal event summary for each node. At present, these include

- message counts and message volume, ordered by message type, message size, and source and destination node,
- processor state, including utilization, context switches, and system calls, and
- program state, including current procedure and execution profile.

Although the semantics of each filter's internal state differ, these semantics need not be known by the display tools. By isolating semantic issues, different filter data can be displayed in multiple ways using standard views (e.g., bar charts and meters). Thus, each filter has an associates set of event strainers that, via access to the internal filter state, create a view specific data representation.

Given the diversity of data filters, a correspondingly rich set of performance views are needed. In addition, standard view interface mechanisms must be provided to permit rapid construction of new views, based on specific parallel system requirements. Although the most useful set of views can be determined only from experience, we have constructed a prototype set of views using the widgets of the X window system [4]. These include dials, bar charts, LEDs, Kiviat diagrams, matrix views, 3-dimensional perspective plots, and a general purpose graph display used for both procedure call graphs and hypercube topological views. Each display is configurable, via the environment control and the X window manager, and is capable of displaying data from a variety of sources. In addition, the portability provided by X permits the display environment to execute on a wide variety of vendor workstations.

The user interface allows the performance analyst to configure and manage the filters, strainers and views. Via this interface, the user can change the attributes of performance views, open new views or close existing ones, and

[4] This isolates idiosyncrasies of the hardware event format and permits compact data representations for network transfer and disk storage.

change the binding of view and data filter (e.g., by replacing the strainer for a bar graph of processor utilization with a strainer for a utilization meter).

The environment defines several interface standards, allowing addition of new filters, strainers and views. These include standards for module initialization, termination and user customization. The builder of a standard module (i.e., filter, strainer, or view) need only meet these interface standards. The environment intrastructure then provides intermodule communication. Early experience suggests that this design promotes extensibility and design flexibility.

5.2 Examples

TO BE ADDED.

6 Conclusions

TO BE ADDED.

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. Central Data Corporation graciously donated a MULTIBUS II interface card and provided invaluable hardware configuration advice. Without their help, the design of the iPSC/2 hardware monitor would not have been possible.

References

[1] MALONY, A. D., AND REED, D. A. Visualizing Parallel Computer System Performance. In *Instrumentation for Parallel Computer Systems*, I. Bucher, M. Simmons, and R. Koskela, Eds. Addison-Wesley Publishing Company, 1989.

[2] RUDOLPH, D. C. CRYSTAL: A Performance Evaluation Tool for the Intel iPSC/2. Master's thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, May 1989.

[3] 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).

[4] SCHEIFLER, R. W., AND GETTYS, J. The X Window System. *ACM Transactions on Graphics 5*, 2 (Apr. 1986), 79–109.

[5] STALLMAN, R. M. Using and porting GNU CC. Tech. rep., Free Software Foundation, Inc., Cambridge, MA, Dec. 1988.