The RP3 program visualization environment

by D. N. Kimelman
T. A. Ngo

The performance promised for parallel systems often proves to be somewhat elusive. This paper discusses one important technique for improving the performance of parallel software: *program visualization*—helping programmers visualize the real behavior of an application or system by presenting its state and progress in a continuous graphic fashion. An environment for visualization of program execution is described. Within this *visualization environment*, programmers dynamically establish views of the behavior of a program in execution and watch for trends, anomalies, and correlations as information is displayed. By continually refining the view of the program and replaying the execution of the program, programmers can gain an understanding of program (mis)behavior. This is essential for the debugging, performance analysis, and tuning of parallel software. Design goals for the visualization environment include expandibility, portability, and the ability to accommodate diverse architectures, including highly parallel shared-memory systems and large-scale message-passing systems. Results from visualization of systems and applications running on the RP3, an experimental shared-memory multiprocessor, are presented in the form of color reproductions of typical, useful displays.

Introduction

It is now generally accepted that parallelism significantly compounds the difficulties involved in producing software that is correct and performs well. As a result, debugging and tuning are now widely recognized as crucial aspects of developing software for parallel systems. Unfortunately, support for these activities is often rudimentary at best (although much research has recently been devoted to improving this situation [1, 2]). With parallel debuggers, programmers can suspend execution of a program and examine the state of the system, but debuggers provide only snapshots of small subsets of the total program state. Typically, these snapshots are widely separated in time. Significant events can easily be missed or overlooked. Further, with conventional tracing facilities, programmers can quickly be overwhelmed by volumes of textual output.

Recently, it has become apparent that a more effective way to present program behavior is in a continuous graphic fashion. An important lesson from *scientific* visualization [3] in the computational sciences is that users can deal much more effectively with large amounts of data when the data are presented visually. Information is more rapidly assimilated, and trends and anomalies are more readily recognized. *Program* visualization aims at capitalizing on this human capacity by presenting the results of monitoring a system in a visual fashion [4–9].

Program visualization can be crucial for debugging parallel applications, for tuning these applications, and for adapting them to the variety of architectures on which they may have to run. Visualization facilities can also help

ensure that various models of parallel computation or models for performance prediction do in fact accurately reflect the behavior of an application. In addition, visualization can be used to verify that automatic parallelization is working effectively and to identify regions of code where further manual amendment might yield substantial improvement in overall performance.

This paper describes the RP3 program visualization environment. With this environment, programmers can dynamically establish views of program behavior on a display and have the execution of a program presented in a continuous graphic fashion through these views. At any time during the execution of the program, the programmer can dynamically reconfigure the display by placing additional views on the display, by removing some of the existing views from the display, or by rearranging the views on the display. Program execution may be "replayed" by repeatedly viewing a recorded execution history or by viewing repeated executions of the program as they occur. By continually refining the arrangement of the views and replaying the display of program behavior, programmers can gain an understanding of program (mis)behavior. The work described here is novel in that behavior from *all* levels of a system—hardware, operating system, run-time libraries, and application—are displayed simultaneously for correlation by the user. Furthermore, the system is highly extensible and adaptable to new parallel architectures. Results of the use of this environment are presented in the form of color reproductions of typical, useful displays.

One approach to program visualization

Our general approach to program visualization arises from our perception of common practices in hardware development. In certain respects, tools for analysis of software behavior lag far behind those for analysis of hardware. When hardware under development malfunctions or behaves unexpectedly, a developer rolls up a cart loaded with diagnostic equipment, attaches clips, extenders, probes, indicators, meters, scopes, and analyzers to a circuit board, and rapidly obtains a good graphic characterization of the behavior of the logic.

Our goal is to establish this type of facility in the software domain. For an application, one should be able to create a graphic representation of the program, run it until some condition of interest is encountered, and stop and run it backward, looking for what led to this situation. Then one might change the representation of the program, on the basis of a hypothesis concerning the underlying cause of the situation, and run the application forward again. By continually repeating this process, one could isolate program behavior of interest.

We propose a toolkit approach to program visualization, analogous to the diagnostic tools classically employed for hardware development: a large assortment of standard, generally applicable, "snap-on" components. "Probes" are clipped onto the target system (the system whose behavior is being visualized) to extract information of interest while the system is running. This information is passed to a visualization workstation, where graphic display attachments are connected to the information streams emanating from the various probes.

Examples of information for which a user might probe include the following: at the hardware level—instruction-execution rate, cache misses, memory and interconnection-network traffic, samples of program-counter contents, and physical addresses of instructions and data; at the operating system level—page faults, binding of threads to processors, scheduling of threads, and binding of various pages of a virtual address space to pages at various places within the memory hierarchy; at the level of the programming language run-time library—entering and exiting parallel sections, acquiring and releasing locks, arriving at and departing from barriers, dispatching work from parallel loops, and calling and returning from procedures; and, at the application level—high-level data operations and algorithm state transitions.

The assortment of graphic attachments for displaying the extracted information might include dials, meters, lights with varying brightness and color, scales, bar charts, plots, histograms, timing diagrams, source text, data-structure diagrams, symbol tables, call graphs, control-flow and dependence graphs, synchronization graphs, and system-architecture diagrams.

Support for program visualization must be included in parallel systems at all levels: within the hardware, the operating system, the run-time library, and the application. Facilities must exist for capturing traces—streams of data describing events of interest in the execution of a program. Support must be provided both in the form of a general monitoring mechanism, for controlling trace generation and for handling trace data, and in the form of specific instrumentation—individual pieces of additional code or logic for generating data describing particular events. Monitoring mechanisms must be relatively unobtrusive in order to minimize perturbation of true program behavior. Monitoring must also include the capacity to transfer significant volumes of generated data from the monitored system as quickly as it is being produced, either for long-term storage or for immediate display. Instrumentation code at any of the software levels can either exist permanently within a system and be enabled and disabled dynamically, or it can be dynamically inserted into a system by means of unobtrusive debugging techniques. Hardware assists can be provided to allow instrumentation code to produce trace data in a way that interferes as little as possible with the normal operation of the machine. This can be accomplished with special user-writable registers

and buffers, and separate data paths out of the monitored system.

Development environments must provide means for dynamically enabling selected instrumentation, retrieving the resulting event streams, and dynamically configuring displays of program behavior that are driven by the event streams. Displays can be based on extensible, object-oriented collections of standard graphic components. A windowing system can serve as a framework that handles the creation of new windows, allows "zooming" and "panning" within windows, and facilitates dynamic reconfiguration of windows across multiscreen displays.

An important requirement for this form of system is that it be "open" or "extensible." By clearly defining the architecture of the system and explicitly specifying interfaces and protocols, we can ensure the ability to incorporate custom, user-developed components into the system.

All of this constitutes a *visualization environment*—an architecture and a large collection of standard, generally applicable snap-on components for information capture and display. Within such an environment, programmers can construct an effective graphic presentation of the state and progress of an application in execution. With a set of tools that can be readily applied to fresh problems across a broad range of applications, a programmer can have a quick and easy way to take a fast first look at unexpected system behavior, and have, as well, a facility for more extensive investigation of behavioral phenomena.

By comparison, many first-generation visualization systems are based on statically embedded instrumentation, handcrafted custom graphics, fixed displays, unwieldy user interfaces, and limited control over execution-history replay.

The RP3 monitoring and visualization architecture

- *Hardware monitoring*

The IBM Research Parallel Processor Prototype (RP3) was an experimental, 64-way shared-memory multiprocessor [10]. Each of the 64 processor-memory elements (PMEs) in the RP3 architecture consists of a processor, a floating-point unit, a memory-management unit, an I/O interface, a performance monitor, a cache, and 8 MB of memory. The PMEs are interconnected by a multistage packet-switched omega network. A processor may obtain data from its cache, from its own memory ("local"), and from the memory of some other PME ("global").

The RP3 was designed as a flexible, dynamically reconfigurable system for studying and experimenting with highly parallel architectures, systems, and applications. An important manifestation of the RP3 orientation toward experimentation and analysis is the performance measurement chip (PMC) [11] in each PME. The PMC contains a set of counters for events such as instruction completions, memory references, cache and translation-lookaside-buffer (TLB) misses, local and global memory requests, and requests to the local memory from all other processors. The PMC also contains buffers that can collect virtual- or absolute-address samples, or samples of delays encountered while accessing global data via the network. The counters are incremented and the samples are collected automatically and unobtrusively by the hardware. Whenever a counter overflows, the processor is interrupted. The processor can then unload the counters and buffers, and reset them. Alternatively, the system console and I/O processors can unload the counters and buffers periodically through special access paths, without interrupting the RP3 processors and without involving the operating system. This minimizes perturbation of system behavior.

The PMC also contains three user-writable "annotation" registers. With a single user-mode instruction, an application can write an arbitrary integer value into any one of these registers. The values in these registers are sent with the rest of the PMC data whenever the PMC is unloaded. An application typically deposits an identifying value into one of these registers each time it enters another phase of its computation. The value of one or two important state variables, such as measures of convergence for numerical algorithms or temperatures for simulated annealing, can be deposited into the other registers. In this way, the application annotates, in effect, each segment of the hardware-performance data stream. The annotations provide an indication of the application activity to which each segment corresponds.

The Mach operating system [12] has been ported to RP3 and has been extended to provide control over unique architectural features of RP3 [13]. Support for the PMC has been incorporated into the Mach kernel in the form of a Virtual PMC (VPMC) facility [14]. The VPMC facility provides the illusion of much wider event-counter registers and of PMCs that follow a thread across context switches and from one processor to another.

- *Event logging*

Mach for RP3 has also been extended to incorporate an event-logging facility. As execution reaches various points in the application, run-time, and kernel, instrumentation code generates event records—records containing event-specific data describing the progress of the system. The instrumentation code places these event records into the event log—a buffer maintained in memory by the kernel. Periodically, the kernel unloads all of the PMC data and generates an event record containing these data. At the end of a run, at convenient pauses during a run (e.g., at barrier synchronization points), or at regular intervals, the

event records are retrieved from the log and transmitted out of the system on a communication link. Typically, they are analyzed immediately and displayed on a workstation or stored as trace files on disk.

By convention, all event records contain an event ID followed by arbitrary event-specific binary data. Examples of event-specific data include phase ID (for an application entering another phase of a computation), loop ID and loop index values (for a run-time scheduler dispatching work for a parallel loop), thread ID and some state information (for the operating system scheduling a thread), and current values of the PMC registers (for the operating system handling an interrupt from the PMC). A time stamp and the ID of the processor from which the record was written are automatically added to each event record by the logging mechanism. The time stamp is obtained from a free-running clock in each processor. The clock resolution is 1 ms. A clock-synchronization algorithm running in the kernel keeps the clocks consistent to within 5 ms. This accuracy is more than adequate for the higher-order software behavior that is typically being investigated.

In order to ensure global integrity of the event log, even in the presence of faulty user-provided instrumentation, the event log is managed by the kernel. Instrumentation inside the kernel accesses the log via read and write routines internal to the kernel. Access to the log for user-mode code, such as instrumentation in the run-time library or an application, is provided by a parallel UNIX®-style system call. In cases in which the overhead of a system call is unacceptable, instrumentation code can accumulate event records and place them in the event log as a group. Time stamps can be obtained from the kernel by using an extremely low-overhead system call (approximately 100 μs) as the individual events of the batch are collected. For cases in which such accumulation is unacceptable and guarantees concerning integrity of the log can be forsaken, an event log can be maintained in memory directly accessible to all instrumentation.

The kernel maintains a section of the buffer for the event log in (protected) kernel memory on each PME. Event records are placed in the buffer in the local memory of the processor that created them. This avoids the contention that would occur if there were a single event-log buffer in the memory of one PME, and introduces no additional traffic on the network that interconnects the PMEs.

- *Event collection*

Figure 1 illustrates the configuration typically employed on the RP3 for event collection and collation by the visualization system. A number of processors (at the left of the monitored-system block) are dedicated to running an application. Instrumentation at each level of the system (application, run-time library, kernel, and hardware) generates events, which are fed into the event-log-buffer sections corresponding to these processors. A few of the processors (at the bottom of the monitored-system block) are dedicated to event-handling tasks. We assert that, in many cases, 90% of the RP3 processors and memory are sufficient for realistic runs of an application for experimental purposes. Thus, we can devote 10% of the processors and memory to monitoring and visualization. One of the monitoring processors captures the UNIX standard output of the application and bundles it into event records, which it deposits into the event log. Subsequently, this output can be correlated with the other data in the log. Another processor (at the right of the monitored-system block) collects event records from the event-log-buffer sections corresponding to the application processors, collates the event records according to time stamp, and forwards them out of the system, across a network link to a workstation.

The various collections of instrumentation embedded within the system are enabled or disabled at the beginning of each run of the application. It would be a simple matter to implement a facility that would allow instrumentation to be enabled and disabled dynamically during the run, on demand from a workstation.

- *Analysis and display*

Analysis and display of event streams emanating from the target system is performed by software that runs on a workstation. As illustrated in Figure 1, event records arrive over the network from a running monitored system, or they are read from a stored trace file. The event records are placed in an internal buffer and are then passed through a configuration consisting of couplings of analysis components and display components (at the right of the visualization workstation block) in order to produce a display.

An example of an analysis component is one that accepts a stream of event records containing PMC counter values and produces a stream of single data values. For instance, each time an event record is received, a data value could be produced that gives the instruction execution rate of a processor over the interval between the previous event for that processor and the one currently being processed. An example of a display component is one that accepts a stream of single data values, and displays a bar graph. Each time a data value is accepted, the bar graph is updated to show the current value.

Our intent is to have a small number of types of streams, and for each type of stream, a large number of analysis and display components. This provides a large number of ways to display any piece of information. Further, when a new analysis component is added to the system, it immediately has available to it all of the existing display components that accept the type of stream that it

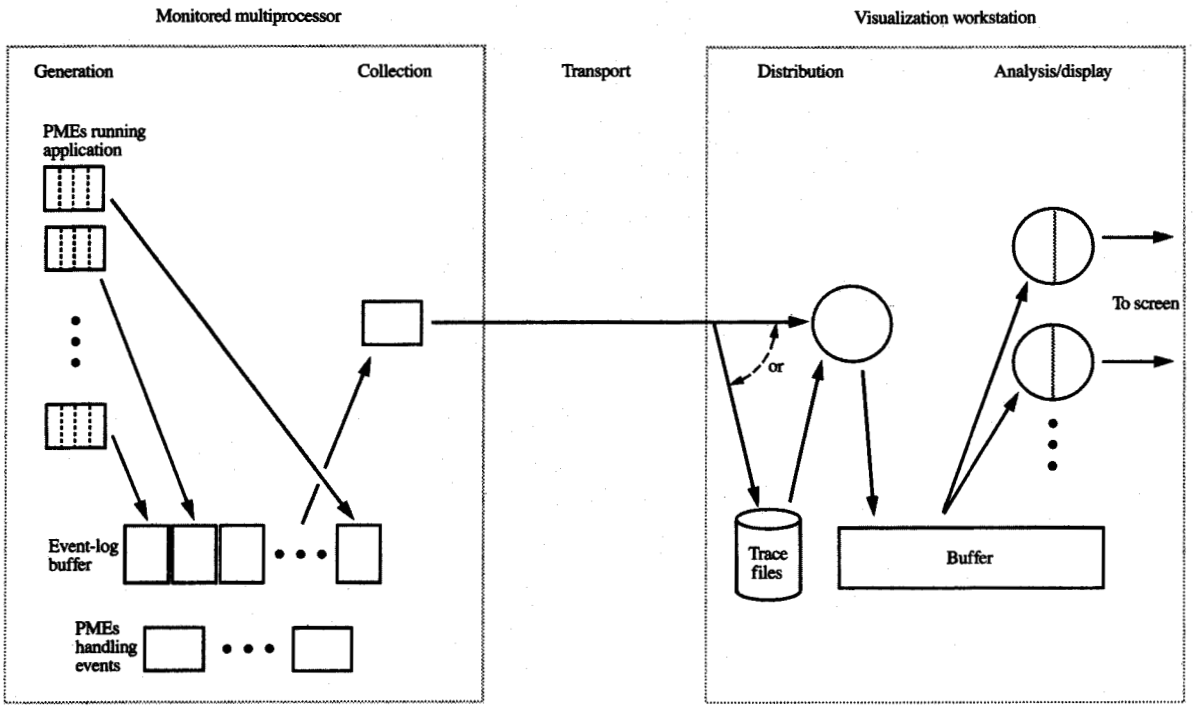


Figure 1

Program visualization environment architecture.

produces. Similarly, when the system is extended with a new display component, the component can be used immediately to display data from any of the existing analysis components that produce the type of stream that it accepts.

Thus, when a user requests that the workstation display be dynamically reconfigured to present additional information, the system must obtain additional analysis and display components of appropriate types, couple them, and connect them to the flow of events from the internal workstation buffer. At that time, if appropriate, a request could be sent to a running target system to enable the instrumentation that provides the information required by the analysis component.

Visualization software structure

The visualization software running on the workstation is an X Window System™ application [15]. The X Toolkit [16] provides a rudimentary object-oriented programming facility, and the Hewlett-Packard Widget set [17] (a precursor of the OSF Motif™ Widgets [18]) constitutes a graphic-object library upon which the user interface is based. Any X window manager may be used to manipulate (reposition, resize, iconify) the windows through which the application produces its displays.

Within the context of the X Toolkit, objects are referred to as "widgets," and each object automatically has a corresponding window in which it displays its graphics. Widget classes include routines to initialize a new instance, redisplay the graphics of an instance, handle window size changes, and accept new values for object parameters, referred to as "resources." Widgets are created within a dynamic hierarchy, distinct from the static class hierarchy. This dynamic hierarchy facilitates inheritance of values for resources and, to some extent, reflects the nesting of widgets on the display.

The analysis and display components of the visualization software are implemented as widgets. An "event widget" accepts a stream of event records, analyzes the events as discussed in the above subsection on analysis and display, and produces a stream of data, which can be filtered and displayed in any of a number of ways.[1] A "filter widget" accepts a stream of data, performs a function (such as

[1] We use the X Toolkit and widgets uniformly throughout the application for object-oriented programming, even though event and filter objects do not produce graphics.

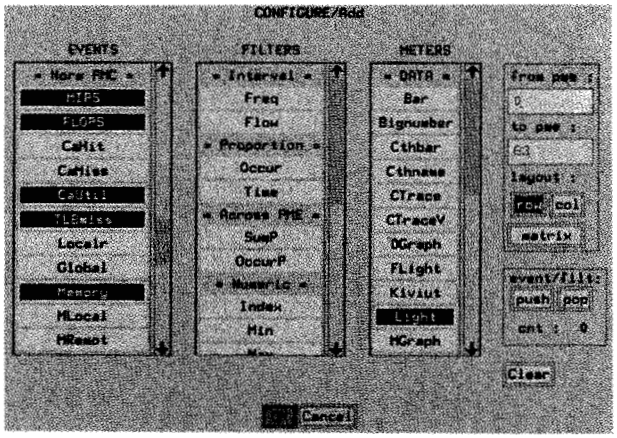


Figure 2

Configuration-dialog box.

scaling values or maintaining a maximum, minimum, or average of the values encountered), and produces a new stream of data. A "meter widget" accepts a stream of data and produces a display. "Trigger widgets" are pseudo-meters which, rather than displaying data, exert control over the system based on the data that they receive. For example, a trigger might be used to suspend the replay of program execution whenever a value exceeds some allowable threshold. Without loss of generality, trigger widgets are omitted from the following discussions.

There is a class description for each kind of event, filter, and meter widget, and there is a global table that lists all of these classes along with various attributes related to their use in configuring visualization displays. When a user requests an addition to the current configuration of widgets, a "dialog box" is derived automatically from the global table and is presented to the user, as shown in **Figure 2.**

The user selects a number of event classes, a number of filter classes, and a meter class from the dialog box (invalid combinations of widgets are disallowed). An instance of each selected event class is created, and each instance is coupled to a chain of filter widgets obtained by instantiating the selected filter classes. Each of the resulting event-to-filter-chain couplings is then coupled to a distinct instance of the selected meter class. In cases where an instance of the meter class can accommodate more than one input, all of the event-to-filter-chain couplings are simply fanned in to a single meter widget. All of this can be replicated for each of a number of processors of the monitored system. In some cases, a single event widget can be coupled to a number of distinct filter chains corresponding to a number of distinct processors. For example, when a user selects MIPS, FLOPS, CaUtil, TLBmiss, and Memory events, as well as the Light meter, for processors 0 through 63, as shown in Figure 2, the resulting widget configuration involves one PMC event widget for each of the selected statistics, each event widget driving 64 Light meter widgets. The resulting display is shown at the top right of **Figure 3.** This display is found in several of the following figures and is discussed in greater detail below. If the user had selected the Average filter, an Average filter widget would have been placed between each event-widget–Light-widget pair.

This user interface protocol for configuring displays is not entirely general, but it does allow simple configurations to be created quickly and easily. So far, in practice, this protocol has not precluded any configuration that our users attempted naturally. As an option, a more elaborate configuration language or a graphic protocol for configuring arbitrary graphs of widgets [19–21] could be provided.

A simple language has been developed for describing configurations, so that once a useful configuration is achieved, it can be saved as a text file and subsequently reloaded.

Extending the system to provide a new kind of display or to accommodate a new form of event involves writing a new event, filter, or meter class, compiling it, making a new entry in the global table, recompiling the global table, and relinking the system. Experience indicates that a programmer familiar with the system and with a clear notion of the function of the new widget can extend the system in this way in one half to two days. Initially, achieving the requisite familiarity may require four to six weeks.

The types of configurations discussed above, consisting of event widgets coupled to filter widget chains and meter widgets, constitute an event-data distribution network superimposed on the normal dynamic widget hierarchy. This network is implemented by a structure incorporated into each widget, which lists the widgets to which it provides data. These structures are similar to, but distinct from, those maintained for manager widgets to list those widgets under their control. The X Toolkit has been extended to include intrinsics for attaching widgets to and detaching them from the event-data distribution network. A common superclass for event and filter widgets provides the additional structure for the widgets. It also provides default (inherited) routines for maintaining this structure when widgets are attached and detached.

The standard toolkit intrinsic for setting resource values could have been used to pass data from one widget to the next along the network. This, however, was found to be excessively slow because of the need to perform conversion for typed values and to interrogate the

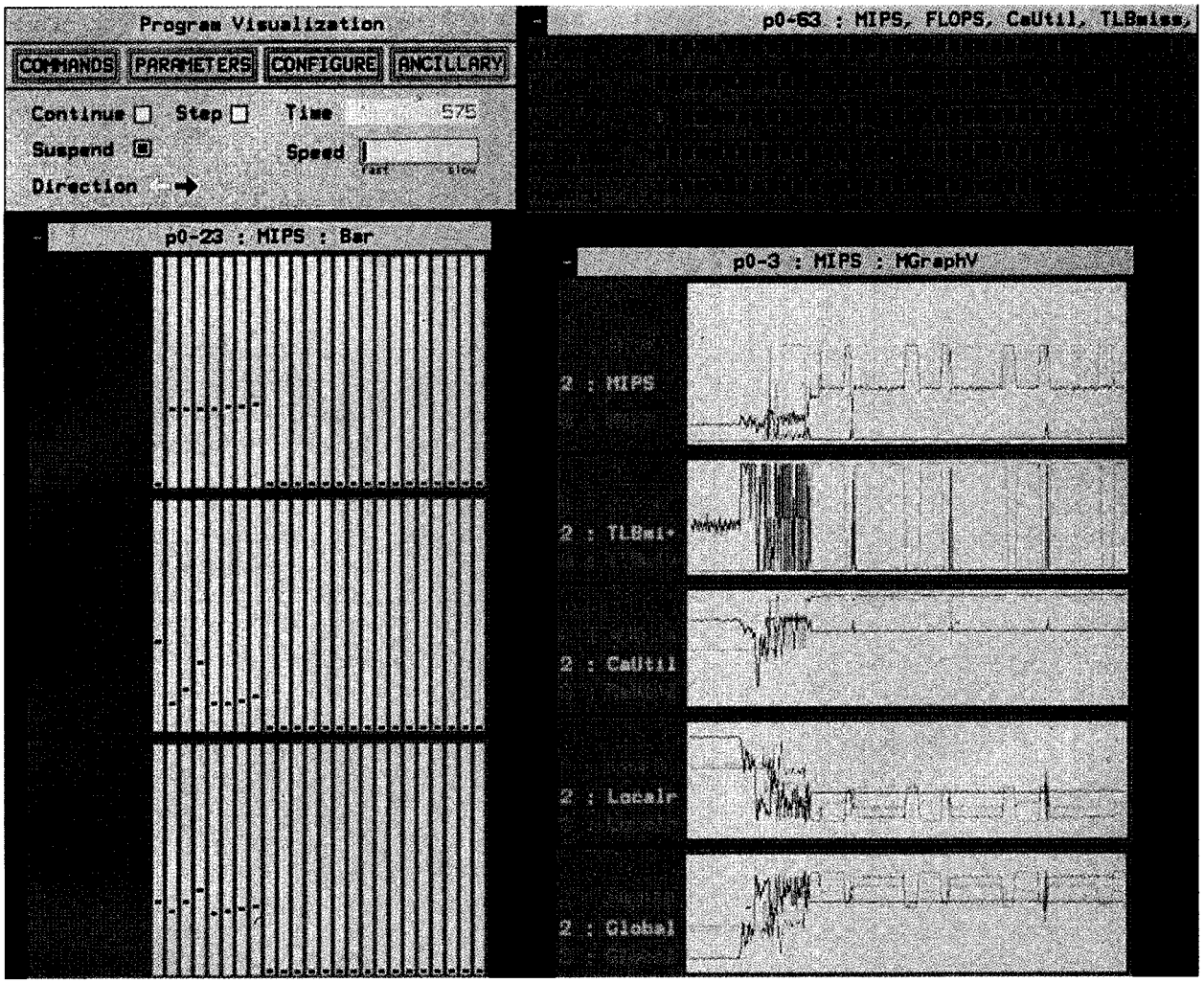


Figure 3

Hardware-utilization display.

superclass chain of the widgets in order to find the routines responsible for handling the resource. Instead, the toolkit has been extended to include an intrinsic for passing untyped values directly to a widget. This yields an improvement of roughly 40% in the performance of the visualization system. Note that the global widget-class table discussed above contains sufficient information to allow the configuration routines to guard against connecting incompatible widgets. The common superclass for event and filter widgets provides a default routine, corresponding to this new intrinsic, which passes received data onward, without change, to all of the widgets to which it is attached. Thus, a fan-out within the distribution network can be implemented by instantiating the common superclass itself.

The use of X as the graphics system on which the visualization software is based results in a number of benefits in addition to graphics functionality. Convenience and familiarity for users result from employing standard X operating procedures and window managers. The system is extensible, in the sense that analyzing new forms of events or displaying data in a new way require only that new widgets be written in the standard fashion. The system is portable, in that the visualization software can be run on any UNIX workstation supporting X. Finally, the network transparency and device independence of X allow the visualization display to be run at remote sites on any of a number of different workstations. We have demonstrated this a number of times, by running an application on RP3 at Yorktown Heights, New York, having the event stream

sent over a local area network to the visualization software running on an IBM RT® workstation in Yorktown Heights, and having the graphics sent over a network to a Sun workstation at sites such as Syracuse, New York, or Calgary, Alberta.

Separating the monitored system from the visualization system by the use of standard network protocols allows the visualization system to be easily adapted to new monitored systems. Accommodating a new monitored system requires producing an event stream from that monitored system by whatever means are available and converting the event stream into the format understood by the visualization system.

Replay control

Comprehensive control over the replay of execution history from trace files (or live presentation of displays as execution proceeds) is provided by the control panel, shown at the top left of Figure 3. The user can suspend replay, resume it, step it one event at a time, or run it forward or backward at varying speeds. In the future the system will allow a user to "drive" the replay from the keyboard (in order to be more responsive).

While such control might at first seem frivolous, it is in fact essential for the intended mode of operation of the system. As discussed earlier, users typically run the visualization system until a situation of interest arises, then step or run the system back and forth slowly, isolating the precise moment the behavior of interest occurs. At the same time, the user looks for correlations between the various display components that might suggest the underlying cause of the given behavior. One example of such a correlation might be a dramatic decrease in processor speed when departing from a particular state or accessing particular data (due, for example, to thrashing in a set-associative cache as a result of unfortunate memory-access patterns or layouts of data in memory). Note that, while temporal correlations can be transformed into spatial correlations by displays that incorporate time along one axis, not all forms of display have an obvious or practical means for incorporating time.

One measure of the importance of this degree of control over replay is the amount of frustration experienced by users who became accustomed to this style of interaction, and later had to deal with some other system in which this degree of control had not yet been implemented.

Selected results

This section illustrates both the type of displays that have been developed and found useful, and the type of information that can be gained from these displays concerning program behavior. Figures 3 through 9 are images of workstation display screens captured during RP3 program-visualization sessions.

- *Hardware and application behavior*

Figure 3 is a hardware-oriented display. A user might configure such a display in order to provide a general view of data being obtained from the PMCs.[2]

The trace viewed in Figure 3 was obtained using the system console and I/O processors, as discussed earlier in the subsection on hardware monitoring, rather than the event-logging facility. In this manner, with extremely low overhead and little perturbation of the true system behavior, execution history was captured from the beginning of the operating system boot through the end of the execution of a hydrodynamics application. The trace consists of only PMC event records generated once per second for eight processors. The trace file covers 20 minutes of execution time and occupies 860 KB of disk storage.

The control panel displayed at the top left of Figure 3 indicates how far the replay has progressed (575 seconds) and the rate (fast) at which replay is proceeding. The control panel also provides pull-down menus for configuring the system and controlling its operation.

The display at the top right of Figure 3 is the highest-level view of hardware activity. It consists of a number of color dots (lights), with one column for each PME and one row for each statistic of interest: instructions executed per second, floating-point operations per second, cache utilization, TLB misses per second, and memory activity. Note that only eight processors are active. Each light varies in color from blue (cold) for low values through red (hot) for high values. To a "seasoned observer," the various patterns of color in the lights indicate the kind of activity taking place, the phases through which a computation is moving, and where difficulties are arising. Problems such as memory contention and thrashing in caches stand out dramatically. For example, at the time represented by this display, it is clear that processor 3 is incurring far more TLB misses than any of the other active processors.

Since there is a fixed amount of screen space, the user is continually faced with the trade-off between the number of different kinds of information that can be displayed and the amount of detail that can be displayed for each kind of information. The three bar charts down the left side of Figure 3 give more detailed information with finer resolution than the array of lights, but for fewer performance characteristics. Each chart consists of a bar (actually, a floating dot) for each PME, the height of the bar indicating the value of a given statistic for that PME. The top chart shows instructions per second, the middle

[2] Please note that color is used extensively in these displays. It highlights the structure and organization of the various displays, it distinguishes among kinds of data being plotted on a single display, and it provides another dimension for plotting data values. Unfortunately, the full range of color in the displays may not be reproducible by the printing process. Hence, some of the displays shown here may prove slightly difficult to comprehend.

chart shows references by each processor to its local memory (as a percentage of the maximum possible rate), and the bottom chart shows activity of the PME memory from all sources. Spikes and dips in these charts reflect an uneven load on the system. Spikes such as those for processor 3 in the middle and bottom charts often result from heavily accessed code or data being concentrated in a particular PME. It is possible, for example, that a small amount of shared data is being accessed repeatedly by all of processors 1 through 7 and that, by virtue of interleaving, the data lie mostly in the memory of PME 3 and partly in the memory of PME 2. In this case, the references made by processor 3 to these data are mostly local references, whereas the references made to these data by the other processors are mostly global. Further, because it holds the data being accessed by all of the processors, the memory in PME 3 sees greater activity than the other memory units.

The five graphs down the right side of Figure 3 give information for fewer PMEs, but include history. Each graph consists of a line for each PME, showing the value of a given statistic for that PME over time, and each PME is assigned a different color (corresponding to the order of the label colors). Corresponding points in time on the different graphs are aligned vertically. At the left section of these graphs, only PME 0 is active, and the rest of the PMEs are idle except for periodic clock handling. We conclude that this must be the time during which the operating system (running on PME 0) loads the application. This is followed by a flurry of TLB miss activity by PME 1 (green), concurrent with low processor activity (MIPS chart) by PME 1, moderate processor activity by PME 0, and high processor activity by the remaining PMEs. This must be the phase in which the master application processor (PME 1) is initializing the application address space and incurring numerous page faults, which are being serviced by the operating system. The remaining processors are spinning at a barrier synchronization point, waiting for the master to complete the initialization. (Spinning is the only way that they can achieve the instruction rates shown.) After this flurry of activity, the application processors settle down to a regular pattern of behavior. There are flat stretches at a moderate instruction rate, during which they are doing useful work, and spikes at a high instruction rate, where they are spinning while waiting at a barrier. The first barrier of each pair of barriers ends the first phase of a two-phase iteration, and the second barrier ends the second phase. Note how the various phases of a computation can be recognized readily just from hardware-level information, even without any displays of higher-level function.

Figures 4 and **5** are views of activity from earlier times in the same trace. They show various phases through

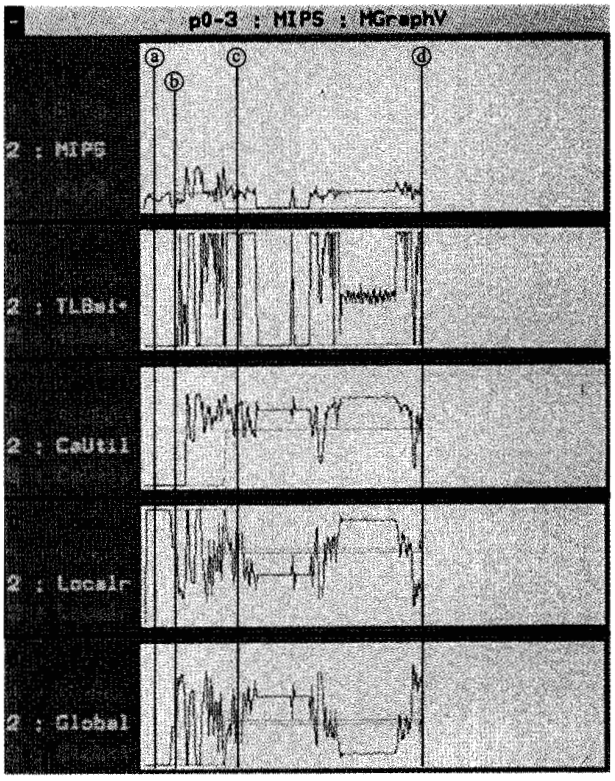


Figure 4

Graphs showing hardware-utilization history.

which the operating system progresses during the boot process. Initially, all PMEs clear their local memories. Each executes a small piece of code from its local memory, with address translation and cacheing disabled. As shown by the bar charts in Figure 5(a), all references are local. Further, because of disabled caches, the processors run quite slowly and memory is (uniformly) quite busy. Next [Figure 5(b)], PME 0 (the master system processor) copies the operating system kernel instructions into global interleaved memory, while the other PMEs (slave processors) busy-wait (wait by executing a small loop repeatedly). As shown in Figure 5(b), PME 0 no longer makes local references exclusively. Further, as a result of its references to global memory, its processor runs even more slowly, and its memory has less activity. Interleaving distributes the global memory activity evenly across the other PMEs, and this activity causes an imperceptible increase [compared with Figure 5(a)] in the memory activity of the other PMEs. This increase is seen in the minute increase in the height of the corresponding memory bars. Once the kernel code has been copied to global memory, all processors enable address translation and cacheing and begin executing from this single copy of

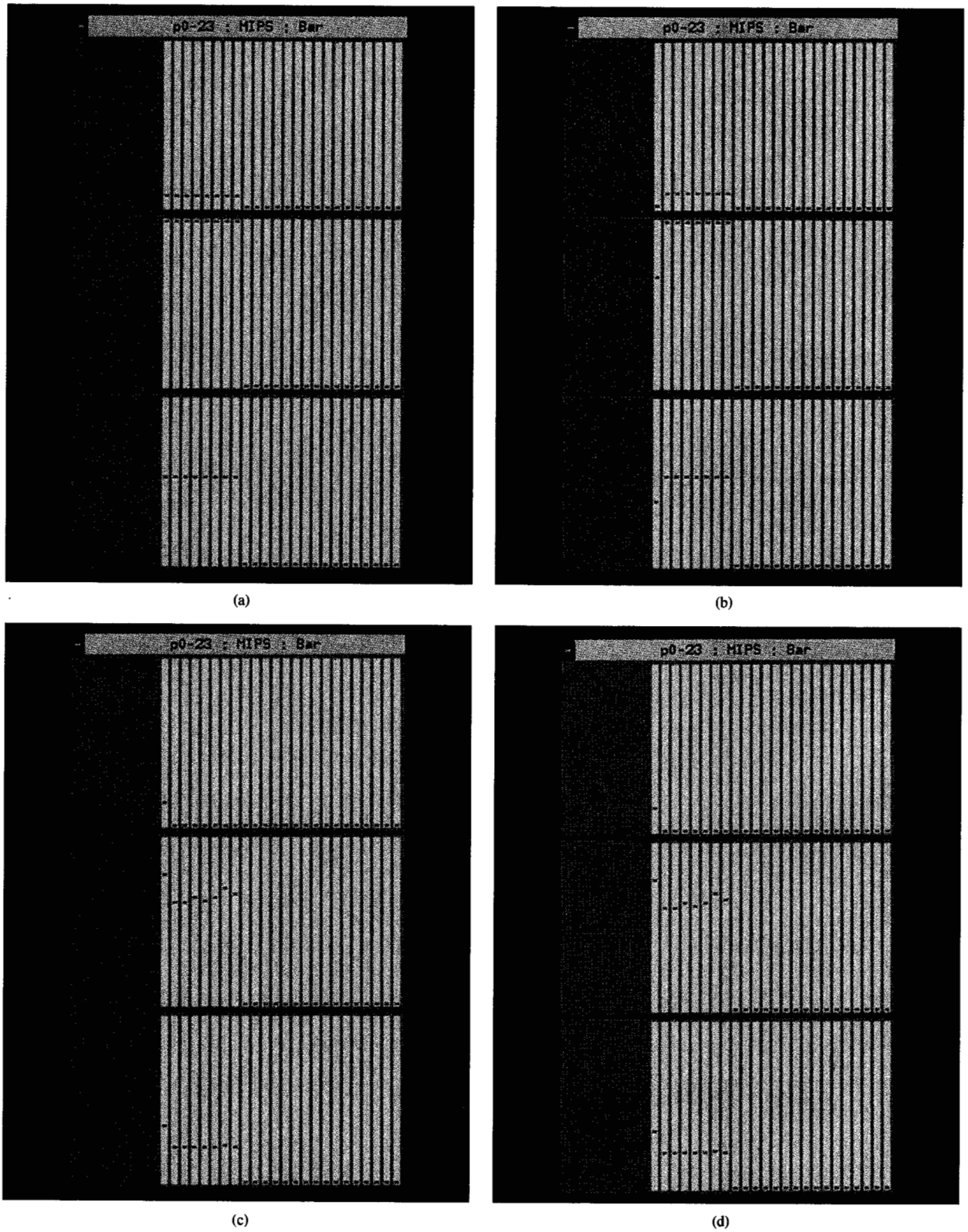


Figure 5

Bar charts corresponding to instants in Figure 4: (a) Kernel boot—clearing local memory. (b) Copying kernel code to global memory. (c) Initialization of kernel data structures. (d) Beginning an application.

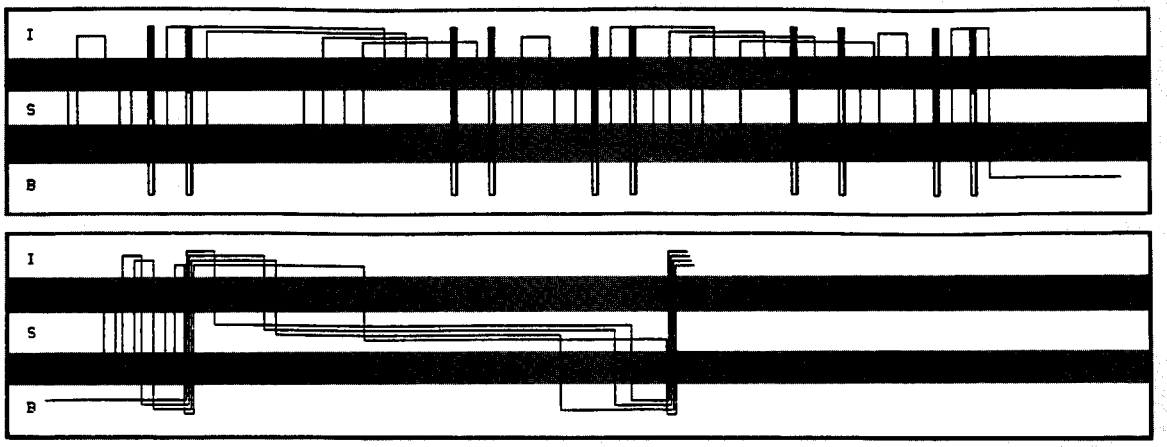


Figure 6

Thread state-history display.

the code. The third graph in Figure 4 shows some of the resulting cache activity, and the bar charts show a corresponding reduction in demand on memory. The second graph in Figure 4 also shows TLB miss activity for PME 0 as the master begins traversing its address space while initializing kernel data structures. Processor speed for the slaves has dropped to nearly zero, because all of them have executed wait instructions rather than busy-waiting for the master.[3] Figure 5(d) shows the slave processors "waking up" as an application begins to run. Each slave experiences a number of TLB misses (Figure 4) as it initially traverses the address space. Local memory access and instruction rate vary widely because the slaves proceed at different speeds through the initialization code. They may incur TLB misses or page faults at different times, and resolution of these situations may take different amounts of time. Once the application begins its main work, conditions stabilize, resulting in patterns similar to those in Figure 3.

- *Thread behavior*

Figure 6 is a thread-oriented display (from an early prototype of the visualization environment) that depicts the state history of each of a number of application threads in the form of a timing diagram, with one line for each thread. The trace in Figure 6 results from instrumentation at a number of different levels. Application-level instrumentation provides information concerning access to various regions of data, as well as information concerning transitions between algorithm phases, by generating event-records containing the relevant data. Run-time-library-level instrumentation provides information concerning thread dispatching and synchronization activity. PMC data were collected to provide hardware-level information. The trace file covers six minutes of execution time and occupies 187 KB of disk storage. Companion displays shown in [6] illustrate access by the application to matrix data, and the corresponding hardware utilization.

The display in Figure 6 follows four threads. It is divided into two horizontal bands. Time increases from left to right and wraps from the right edge of the top band to the left edge of the bottom band. The left edge of each band is annotated to indicate five higher-level states through which a thread progresses: *I*—getting work; *F*—doing one kind of work; *S*—doing another kind of work; *L*—waiting for, and then holding, a lock that allows the thread to complete its work; and *B*—waiting at a barrier synchronization point. (See [23] for a discussion of barrier synchronization.) The thick vertical bars appearing occasionally along the diagram indicate that time passed without any state changes. The height of the hollow bar inside the dark bar indicates the amount of time that passed; a full-height hollow bar indicates time equal to the full width of the display.

The first barrier synchronization occurs just after the start of the bottom band. It is interesting to note that one thread seems to have arrived at the barrier a good deal

[3] A more detailed description of the workings of the RP3 kernel can be found in [22].

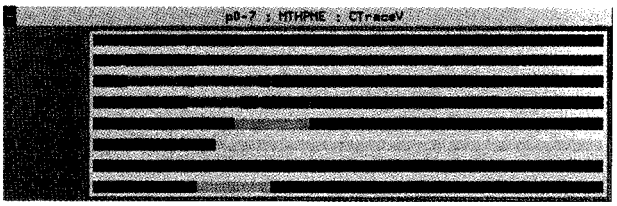


Figure 7

Thread scheduling by the operating system.

earlier than all of the others (near the end of the top band). This might indicate a load-balancing problem, the need for dispatching finer-grain work, unfortunate input data, or higher or lower performance of the particular processor on which the thread was running. Some of these hypotheses can be confirmed or refuted by studying the correlation of the state-history diagram with other displays that show, for example, hardware performance or data-access patterns. On the other hand, when we consider the elision bars, we see that the amount of time by which the one thread precedes the others to the barrier is actually quite small compared to the time required to do a single piece of work (time spent in the *F* or *S* state), and may not be a cause for concern.

A second barrier synchronization occurs just past the halfway point in the bottom band. Judging by the difference in appearance of the thread behavior between the top band and the bottom band, one may reasonably conclude that the work taking place between the first and second barriers must be the second phase of (at least) a two-phase computation. In fact, if we examine more of the trace, we realize that this is an iterative computation with a two-phase iteration. It is interesting to note that the second phase takes much less time than the first, that no locking is required in the second phase, and that there are only enough data to provide one piece of work for each thread. Thus, dynamic structure of an application, or high-level patterns of behavior, which would be extremely difficult to deduce or discern by any conventional method, can be readily recognized with a visualization system, even without detailed knowledge of an application.

- *Operating system behavior*

Figure 7 is a display oriented toward operating-system-level scheduler activity. It illustrates the migration history of a set of threads. The trace includes both operating-system-level information and run-time-library-level information, in addition to the hardware-level information from the PMCs. Operating system instrumentation provides information concerning context switching by processors from thread to thread. C Threads run-time library [24] instrumentation provides information concerning creation, naming, and destruction of C Threads, as well as dispatching C Threads to Mach threads, and operations on lock and condition variables. The trace file covers two minutes of execution time and occupies 136 KB of disk storage.

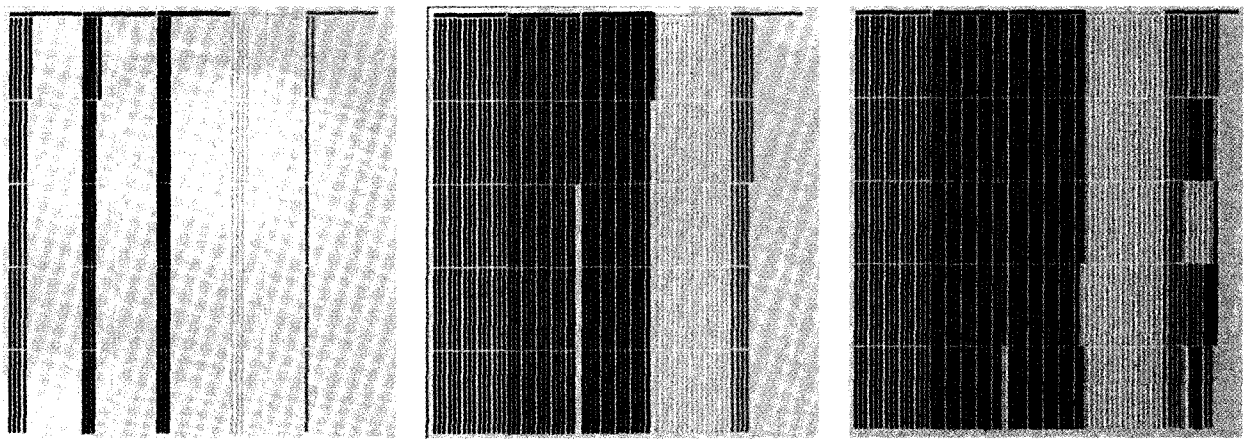
The eight horizontal color strips in Figure 7 show the scheduling of threads onto processors, over time. There is one strip for each processor, from the top strip for processor 0 to the bottom strip for processor 7. Time increases from left to right. The strips scroll to the left as time passes. Each user thread is assigned a distinct color; the darkest blue is used to represent all kernel threads. The strips are colored according to the threads that are running on the corresponding processors at each instant. For example, the display shows that processor 5 initially runs one or more kernel threads (most likely the "idle thread") for a while and then runs the magenta thread briefly. The magenta thread then runs briefly on processors 0 and 1 before settling on processor 1. The cyan thread begins on processor 3, runs there briefly, migrates in quick succession to processors 0, 4, and 7, and then settles on processor 2, where it runs to completion. Note that the red thread runs periodically for brief intervals, alternating between processor 6 and processor 0. This thread was alluded to earlier in the section on event collection: It is the process that collects event records from the event log, collates them, and forwards them over the network. Writing to the network requires a UNIX system call, which can be executed only on processor 0. Thus, the thread is forced to processor 0 for each system call and then returned to processor 6 to continue processing.

- *Language-level behavior*

Figures 8 and **9** are displays of run-time-library-level scheduler activity. They illustrate dynamic load balancing in the scheduling of nested parallel loops (loops in which iterates can be executed in parallel) for PTRAN, a parallel FORTRAN system [25].

The trace viewed in Figures 8 and 9 includes both the hardware-level information from the PMCs and run-time-library-level information generated by instrumentation embedded within the PTRAN scheduler. As a parallel loop is executed, this instrumentation provides information concerning which "chunks," or sub-sequences of the iterations of the loop, are assigned to which threads (each thread is bound to a distinct processor). Also provided are the times at which execution of the chunks begins and ends and the times during which threads are idle. The trace file covers 38 seconds of execution time and occupies 320 KB of disk storage.

Figure 8(a) represents the iteration space of the parallel loop currently being executed. Either a single parallel loop



(a) (b) (c)

Figure 8

Run-time scheduling of nested parallel loops: (a) Beginning of the execution of a parallel loop. (b) Near completion of the work of most threads. (c) Dynamic load balancing.

or a nested pair of parallel loops can be displayed.[4] The x-axis represents the iterates of the outer loop, and the y-axis represents the iterates of the inner loop, if an inner loop exists. As execution proceeds, regions in the display of the iteration space are colored according to the thread that is executing the corresponding chunk of iterations. For example, in Figure 8(a), the green thread has taken the first chunk of the outer loop—the green horizontal block, and the magenta thread has taken the second chunk of the outer loop—the magenta horizontal block. (A window can be popped up to reveal that in this case the green thread corresponds to processor 1. Additional displays can be configured to show, for instance, that the chunk taken by the green thread is iterations 1 through 15 and that the chunk taken by the magenta thread is iterations 16 through 30.) Further, in Figure 8(a), for each of the individual iterates of its chunk of the outer loop, the green thread is progressing, a chunk at a time (the green vertical blocks), through the iterations of complete instances of the inner loop. At the time of this display, it is executing iterations 1 through 15 of the inner loop for iteration 5 of the outer loop.

Note that for loops with a large number of iterations, each pixel of the display could represent a group of iterates rather than a single iterate. As well, the user could be allowed to zoom in and out and pan back and forth across the iteration space. As new loops are encountered, the entire display could be scrolled to the left. In cases in which the parallel loop indices are simply used as subscripts of an array, the display could also be regarded as an illustration of which areas of data are being accessed by which threads or processors.

At the time of the display in Figure 8(b), most of the threads are nearing completion of all of the instances of the inner loop for all of the iterates of their chunks of the outer loop, but the blue thread seems to lag far behind the others. In Figure 8(c), dynamic load balancing is apparent: The yellow thread has helped the magenta and dark blue threads finish their work; now all of these threads are undertaking work that would originally have been expected to have been performed by the dark blue thread.

The reason for the blue thread lagging so far behind the others is revealed in Figure 9, in which additional displays have been configured and execution history has been replayed. Having learned from a pop-up window that the blue thread is running on processor 3, we can see from the array of lights that the processor for the blue thread is incurring an inordinately large number of TLB misses. Thus, a significant fraction of the capacity of this processor is lost to the "overhead" of servicing the TLB misses, rather than being devoted to the true work of the application. The graph at the bottom right of Figure 9 provides a further indication that processor 3, the blue processor in this display, consistently takes far more TLB misses than the other processors.

The graph at the bottom left of Figure 9 provides an indication of the parallelism actually achieved by the application. Time increases from left to right. At any instant, a vertical slice through the graph can have a red component, a green component, and a yellow component, one above the other. The height of the red component indicates the number of threads currently working, the

[4] A new scheme is currently under consideration for representing the iteration space in a way that could accommodate arbitrarily deep nesting of loops.

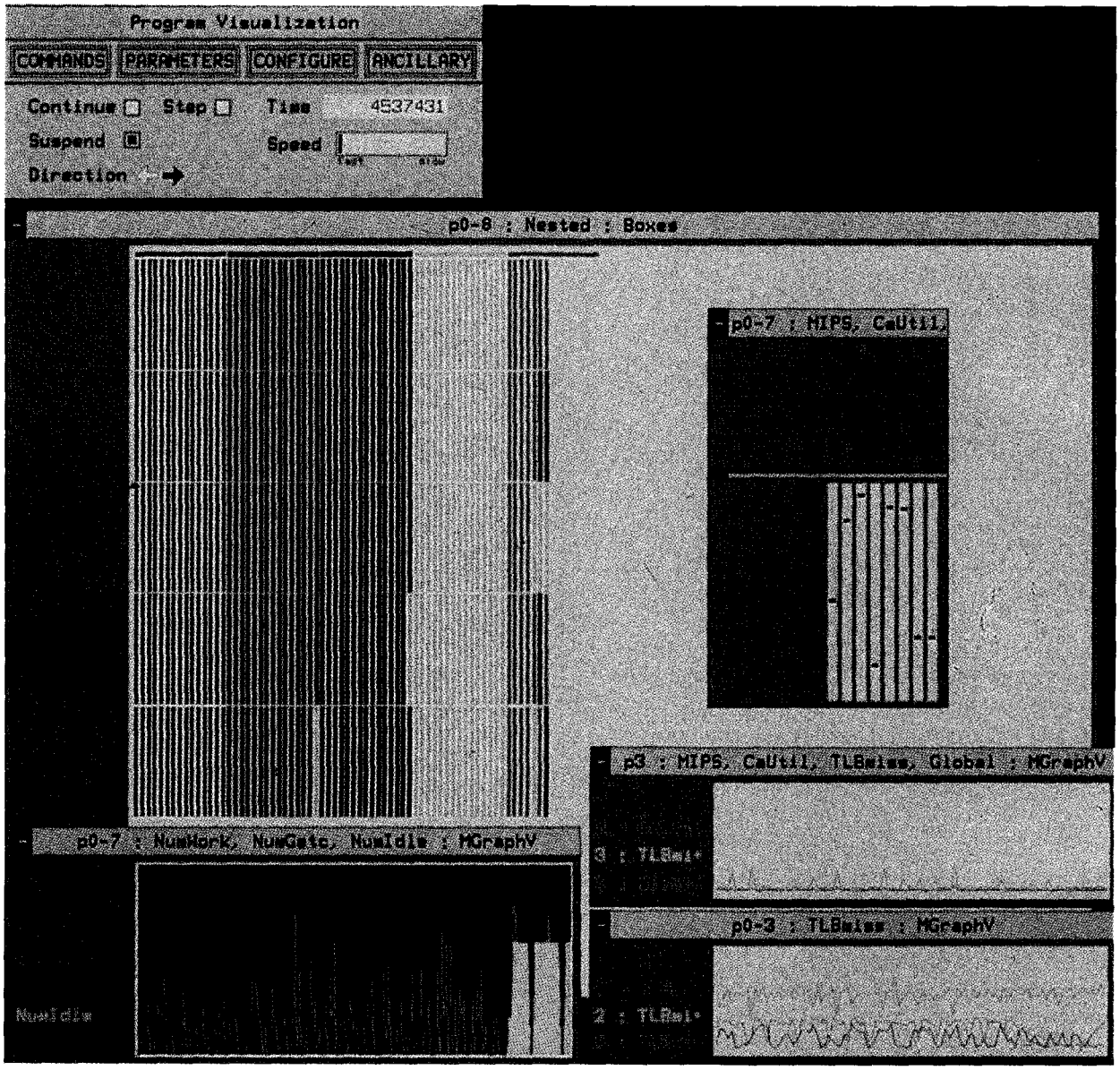


Figure 9

Correlation of loop scheduling with hardware utilization (based on same trace as Figure 8).

height of the green component indicates the number of threads currently getting another chunk of work to do, and the height of the yellow component indicates the number of threads currently idling. A thread is counted as idle only when it is ready for another chunk but no chunks are available to be assigned to it from the queue. Thus, the upward green spikes in the early part of the graph represent threads finishing one chunk and picking up another one. The broad upward yellow spikes at the right of the graph are threads going idle. By replaying execution, we realized that threads begin to be idle at about the time the other threads in the computation come to the aid of the blue thread. This corresponds to the time at which contention for a work queue first becomes a possibility. However, rather than contention, the more likely cause for the idle threads is the fact that there is barrier synchronization at the end of each instance of the inner loop, and the fact that there are only five chunks of work per instance of the inner loop. Unfortunately, the blue thread always picks up one of these chunks. As

threads finish their single chunk of an instance of the inner loop, they must become idle while waiting for all other threads to complete their chunks. Thus, it is highly likely that some number of threads will become idle at the end of each instance of the inner loop, waiting for the (slower) blue thread to finish.

Figure 9 also led to the observation that the green thread never joins the other threads in completing the work of the blue thread. This turned out to be an opportunity for improvement of the design of the scheduler. In cases in which an inner loop has sufficient work to keep a number of threads busy, this lack of participation in dynamic load balancing by the green thread could have a significant impact on the performance of the system.

The excessive idling and the loss of a thread are situations that seem to present opportunities for important improvements to the scheduler. These phenomena were apparent almost immediately from the visualization displays but would almost certainly have gone unnoticed for quite some time in a more conventional environment.

This section has described the display of information from a number of different levels within the system. Although many of the displays were initially developed as examples, they have, in fact, proved to be highly effective in examining and understanding system behavior. As was noted in each case above, a relatively small amount of trace data was required to produce these displays. Trace data rates were often less than 10 KB per second. Overhead, measured as variation in total run time, typically was between 5 and 10 percent. Perturbation was minimal. Most of the problematic behavior that was being investigated was reproducible and remained reproducible after instrumentation was enabled.

Future directions

A number of directions are possible for further work in this area. First, comprehensive, documented case studies are sorely needed to demonstrate the broad applicability, viability, and effectiveness of program visualization. As well, displays for systems with hundreds, thousands, and tens of thousands of processors will soon be required. Some of the displays presented here will generalize quite well to significantly larger numbers of processors; others will not. Further, even for smaller-scale systems, there is a need to consider more abstract displays with the potential to provide more revealing views of program behavior at all levels within the system.

New displays are required for more direct correlation of system behavior with program source. For example, a display of source code could be presented, and colored to show the location of each processor based on program counter samples. Source code could also be colored to show the location of cache misses or page faults. Subroutine-call graphs could be colored to show profile information such as how much time is being spent in each routine.

Design goals for the RP3 visualization environment include

- *Extensibility*, so that custom components specific to particular application domains or classes of machines can be added easily.
- *Portability*, so that the visualization software can be run on any workstation providing X and UNIX.
- *Retargetability*, to accommodate architectures as diverse as highly parallel shared-memory systems, large-scale message-passing systems, and multiprocessor systems with a few very powerful vector processors.

Efforts are currently under way to accommodate traces from an application running on Victor [26, 27], a 256-node mesh-connected transputer-based message-passing system. A few additional event widgets and one or two additional meter widgets, combined with the existing widget set, should provide a powerful facility for visualizing the behavior of message-passing applications running on this system.

Plans are also in place for porting the visualization environment from the Mach IBM RT workstation on which it was developed to newer UNIX workstations. Once the environment is ported to one or more newer workstations, a standard base set of widgets can be completed. The environment can then be deployed for use by application programmers. Field trials and experimental evolution will provide valuable feedback concerning the effectiveness of the various components of the present facility and insight into the visualization requirements of different application domains and different classes of systems. Eventually, perhaps, a central catalog or repository can be established, to which the user community can contribute successful widgets, event generators, operating system modifications for event collection, and interesting application traces. This would allow new users to rapidly tailor their environments to their tastes or requirements, using collections of proven components applicable to their particular context.

Conclusion

The development of a visualization environment for RP3 has been completed. The environment provides an architecture and an extensible collection of generally applicable snap-on components for information capture, information analysis, and information display. These tools can be readily applied to fresh problems across a broad range of applications. Using the environment, programmers dynamically configure views of programs in execution and watch for trends, anomalies, and

correlations in the information that is displayed. By continually refining the view of the program and replaying execution, programmers can gain an understanding of program (mis)behavior.

We have demonstrated remote operation of the system over a network and achieved system-independent display on workstations from a number of different vendors. Working at sites far removed from the RP3, we were able to start an application, and we were able to get better feedback concerning its execution than most users could get sitting in the machine room. The displays shown in this paper have exposed behavior that would have taken much longer or been much more difficult to uncover by conventional means.

We are highly encouraged by the effectiveness of even the simplest visualization tools, and we are optimistic about the potential of extensible environments for the visualization of program execution. Certainly, some problems will resist this form of attack, but even if we can resolve just the "easy" bugs and performance problems with these new methods and then revert to older, more time-consuming methods for the remainder of the problems, these environments will have been well worth the time spent on their development.

Acknowledgments

Many thanks to Jim Arendt, who implemented the displays of parallel-loop activity. Henry Chang developed the operating system support for the PMC and the event-logging facility. Bryan Rosenburg designed and implemented the clock-synchronization algorithm, and Edith Schonberg instrumented the PTRAN scheduler. Tim Guinther refined the user interface and is currently adding components to the visualization system to accommodate a message-passing monitored system. Finally, the referees provided valuable suggestions for improvement of the paper. This work was sponsored in part by the Advanced Research Project Agency under Contract No. N00039-87-C-0122.

References

1. C. E. McDowell and D. P. Helmbold, "Debugging Concurrent Programs," *ACM Computing Surv.* **21**, No. 4, 593–622 (December 1989).
2. S. Utter and C. M. Pancake, "A Bibliography of Parallel Debuggers," *ACM SIGPLAN Notices* **26**, No. 1, 21–37 (January 1991).
3. G. M. Nielson, B. D. Shriver, and J. Rosenblum, *Visualization in Scientific Computing*, IEEE Computer Society Press, Washington, 1990.
4. A. D. Malony and D. A. Reed, "Visualizing Parallel Computer System Performance," *CSRD Report No. 812*, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, May 1988.
5. T. Lehr, Z. Segall, D. F. Vrsalovic, E. Caplan, A. L. Chung, and C. E. Fineman, "Visualizing Performance Debugging," *IEEE Computer* **22**, No. 10, 38–51 (October 1989).
6. D. N. Kimelman, "Environments for Visualization of Program Execution," *Performance Instrumentation and Visualization*, M. Simmons and R. Koskela, Eds., Addison-Wesley Publishing Co., Reading, MA, August 1990, pp. 135–146.
7. *Performance Instrumentation and Visualization*, M. Simmons and R. Koskela, Eds., Addison-Wesley Publishing Co., Reading, MA, August 1990.
8. Special Issue on Software Tools for Parallel Programming and Visualization, L. M. Ni and K. C. Tai, Eds., *J. Parallel & Distr. Computing* **9**, No. 2 (June 1990).
9. M. T. Heath, "Visual Animation of Parallel Algorithms for Matrix Computations," *Proceedings of the Fifth Distributed Memory Computing Conference*, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 1213–1222.
10. G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, "Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proceedings of the 1985 International Conference on Parallel Processing*, August 1985, pp. 764–771.
11. W. C. Brantley, K. P. McAuliffe, and T. A. Ngo, "RP3 Performance Monitoring Hardware," *Instrumentation for Future Parallel Computing Systems*, M. Simmons, R. Koskela, and I. Bucher, Eds., Addison-Wesley Publishing Co., Reading, MA, 1989, pp. 35–47.
12. M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development," *Proceedings of the Summer 1986 USENIX Conference*, USENIX Association, Atlanta, July 1986, pp. 93–113.
13. R. M. Bryant, H.-Y. Chang, and B. S. Rosenburg, "Operating System Support for Parallel Programming on RP3," *IBM J. Res. Develop.* **35**, No. 5/6, 617–634 (1991, this issue).
14. W. C. Brantley and H. Y. Chang, "Support Environment for the RP3 Performance Monitor," *Performance Instrumentation and Visualization*, M. Simmons and R. Koskela, Eds., Addison-Wesley Publishing Co., Reading, MA, August 1990, pp. 117–134.
15. R. W. Scheifler and J. Gettys, "The X Window System," *ACM Trans. Graphics* **5**, No. 2, 79–109 (April 1986).
16. J. McCormack and P. Asente, "An Overview of the X Toolkit," *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software*, October 1988, pp. 46–55.
17. D. A. Young, "X Window Systems Programming and Applications with Xt," Prentice-Hall, Inc., Englewood Cliffs, NJ, 1989.
18. *OSF/Motif Programmer's Guide*, Open Software Foundation, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1990.
19. D. A. Reed, R. D. Olson, R. A. Aydt, T. Madhyastha, T. Birkett, D. W. Jensen, B. A. A. Nazief, and B. K. Totty, "Scalable Performance Environments for Parallel Systems," *Research Report No. UIUCDCS R 91-1673*, University of Illinois at Urbana-Champaign, March 1991.
20. C. Upson, T. Faulhaber, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam, "The Application Visualization System: A Computational Environment for Scientific Visualization," *IEEE Computer Graph. & Appl.* **9**, No. 4, 30–42 (July 1989).

21. D. S. Dyer, "A Dataflow Toolkit for Visualization," *IEEE Computer Graph. & Appl.* **10**, No. 4, 60–69 (July 1990).
22. H. Y. Chang and B. Rosenburg, "Experience Porting MACH to the RP3 Large-Scale Shared-Memory Multiprocessor," *Proceedings of InfoJapan '90—Information Technology Harmonizing with Society*, October 1990, pp. 347–354.
23. J. M. Mellor Crummey and M. L. Scott, "Algorithms for Scalable Synchronization on Shared Memory Multiprocessors," *ACM Trans. Computing Syst.* **9**, 21–65 (February 1991).
24. E. C. Cooper and R. P. Draves, "C Threads," *Technical Report CMU-CS-88-154*, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, March 1988.
25. F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, "An Overview of the PTRAN Analysis System for Multiprocessing," *J. Parallel & Distr. Computing* **5**, No. 5, 617–640 (October 1988).
26. W. Wilcke, D. Shea, R. Booth, D. Brown, M. Giampapa, L. Huisman, G. Irwin, T. Ma, T. Murakami, F. Tong, P. Varker, and D. Zukowski, "The IBM Victor Multiprocessor Project," *Proceedings of the 4th International Conference on Hypercubes*, Los Altos, CA, April 1989, pp. 201–207.
27. D. G. Shea, W. W. Wilcke, R. C. Booth, D. H. Brown, Z. D. Christidis, M. E. Giampapa, G. B. Irwin, T. T. Murakami, V. K. Naik, F. T. Tong, P. R. Varker, and D. J. Zukowski, "The IBM Victor V256 Partitionable Multiprocessor," *IBM J. Res. Develop.* **35**, No. 5/6, 573–590 (1991, this issue).

Received November 29, 1990; accepted for publication May 21, 1991

Douglas N. Kimelman *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598.* Dr. Kimelman received his Ph.D. in computer science from the University of Manitoba, Canada, in 1986. Prior to joining IBM, he served as a faculty member at the University of Manitoba and as a consultant to the electronic publishing industry. Dr. Kimelman joined the Thomas J. Watson Research Center in 1988 as a Research Staff Member. Since joining IBM he has worked in the area of operating systems and environments for parallel systems. He currently leads a group investigating program visualization for parallel systems. His other research interests include debuggers for parallel systems, software development environments, and operating systems. He is a member of the ACM and IEEE.

Ton A. Ngo *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598.* Mr. Ngo received a B.S.E.E. from the Georgia Institute of Technology in 1982 and an M.S.E.E. from Florida Atlantic University in 1987; he is currently pursuing the Ph.D. program in computer science at the University of Washington. From 1982 to 1987, he worked in the IBM System Products Division on the design of a microprocessor. He has been with the IBM Thomas J. Watson Research Center since 1987, and has been involved with the design and integration of the RP3 machine and the development of the Program Visualization environment. Mr. Ngo is interested in issues in parallel architecture, including exploiting data locality, parallel machine simulation, and parallel program debugging.