Performance Debugging in the Enterprise Parallel Programming System

David Woloschuk
Paul Iglinski
Steven MacDonald
Diego Novillo
Ian Parsons
Jonathan Schaeffer
Duane Szafron

Department of Computing Science,
University of Alberta,
Edmonton, Alberta,
CANADA T6G 2H1

Abstract

Debugging parallel/distributed programs is an iterative process, alternating between correctness debugging and performance debugging. Performance debugging involves identifying bottlenecks in a parallel computation and providing meaningful feedback to the user. The quality of this feedback can play a major role in the quick resolution of performance problems. Many feedback systems provide the user with endless streams of statistics, relying on the user to do the interpretation. This paper discusses the performance debugging facilities in the *Enterprise* parallel programming system. Through visual and aural cues, performance information is conveyed to the user in an intuitive manner.

1 Introduction

Many of the papers on distributed debugging begin with the message, explicit or implicit, that while debugging sequential programs might be characterized as a *difficult art*, debugging parallel programs is a decidedly *painful chore*. The cause of this drudgery is a combination of the special problems encountered in distributed computing and the lack of tools for coping with these inherent difficulties.

The process of debugging a parallel program can be divided into two sub-processes: correctness (logic) debugging and performance debugging. The first process is concerned with modifying a parallel program so that it yields a correct solution to a problem. The second is concerned with modifying a parallel program so that it executes faster than the sequential program, but remains correct. Parallel programming systems (PPSs) must provide support for both kinds of debugging. Unfortunately, the literature concentrates on correctness debugging at the expense of performance debugging. This is unusual when one considers that the major motivation for parallel computing is increased performance.

Unfortunately, the two debugging processes are not independent. In a perfect world, once a correct parallel program is obtained, a series of correctness preserving transformations would be applied to the program to increase performance. In reality, parallel performance improvements are usually obtained by increasing the concurrency, which often does not guarantee that correctness will be preserved. Therefore, the debugging process is an iterative one that shifts back and forth between correctness debugging and performance debugging. To support this process, PPSs must either provide a single debugger that supports both kinds of debugging or an integrated uniform environment that supports a rapid and seamless shift between debuggers.

Enterprise is a PPS which supports the various phases and activities of the software development process: designing, coding, compilation, execution, testing, performance analysis and debugging [6]. The graphical user interface provides a simple, uniform view of a program that reflects the high-level programming model of *Enterprise* [5]. The interface views of performance and correctness debugging are

---

The IBM contact for this paper is Jacob Slonim, Centre for Advanced Studies, IBM Canada Ltd., Dept. 21/894 844 Don Mills Road, North York, Ontario M3C 1V7.

identical and are similar to all of the other views. This makes performance debugging a natural extension of the program development process. It is not an auxiliary activity in *Enterprise*.

This paper focuses on performance debugging in *Enterprise*. Correctness debugging is discussed elsewhere [4]. Section 2 provides an overview of performance tuning. Section 3 provides an overview of the *Enterprise* programming model and environment. Section 4 describes performance analysis in *Enterprise*. Section 5 presents our conclusions and an outline of future research.

2 An Overview of Performance Debugging

Parallel program developers are often interested in the execution behavior of a program as a function of time. The behavior is a result of program events, where an event marks a characteristic state change that the programmer deems interesting. For example, memory location accesses, register changes and message sends between processors are events. The goal of performance tuning is to use the events to identify patterns and trends that arise during execution and to use these patterns trends to modify the program to improve performance.

Performance tuning can be divided into three operations. The first is to *acquire* (capture or record) the interesting events while the program is running. The second is to *analyze* the events to produce information about the run. The third is to *present* this information to the user.

There are two methods to acquire and analyze events. *Real-time analysis* is carried out at run-time as events are captured. In *post-mortem analysis*, events are recorded at run-time, but are analyzed in a post-execution process.

Real-time analysis provides an instantaneous view of the execution of a program and allows users to interact with the program during execution. This offers immediate feedback to tuning operations performed by the programmer. The primary problem with real-time analysis is that the time between events can be small and activity can be bursty. Multiple processors can generate thousands of events per second requiring that the analysis engine not only gather large quantities of data from multiple sources, but also process this data as fast as the events are generated. Even if the hardware and software could cope, it is not clear that the information could be presented to the programmer in such a way that it could be absorbed and understood. A second disadvantage of real-time analysis is that it is difficult to take advantage of long-term patterns and trends. That is, real-time analysis tools usually focus on the current few events.

In post-mortem systems, a program run is used to generate a *trace file* that contains the raw events. Unlike real-time systems, full knowledge of the program run is available and trends that emerge over time can be computed more easily. Post-mortem systems can control the passage of real time, allowing events to proceed as quickly or slowly as the user desires. Presentation can be enhanced, since during uninteresting periods the user can *fast forward* to the next point of interest. If events are happening too quickly to absorb, the user can slow down the trace processing and even step through a trace file one event at a time. The major drawback to this approach is that a full trace must usually be obtained before the user can view program activity and do performance tuning. A full second trace must then be produced before the effects of the tuning can be viewed. This is similar to the difference between program development using language compilers and interpreters. Most PPSs use a post-mortem strategy for two reasons; it is simpler to implement and it provides more user control.

Event acquisition and analysis have been well-studied in the literature (for example, [7]). The key factor in acquisition is to reduce the *probe-effect*, the perturbation of the performance of a program due to the execution of instrumentation code. This intrusion can potentially alter the execution of a program to the extent of altering the results. The *Enterprise* approach to the probe effect is described in [4], but this effect is less important in performance debugging than it is in correctness debugging.

This paper focuses more on the presentation of data than on its acquisition or analysis. Presentation is responsible for displaying information to the user in a meaningful, unambiguous and (sometimes) entertaining manner. Limited only by the creativity and imagination of designers, presentation tools often employ clever and colorful means to convey information. Animations and other graphical displays not only grab attention, but are ideal in capturing the dynamic nature of parallel programs. Well-designed graphical displays with simple visual cues and familiar symbolic representations enhance a user's innate ability to process visual information (for example, [7, 1]).

Despite the power of graphical visualization, there are several limitations to such presentation systems. First, if too much information is displayed at one time, or if displayed information is changed too rapidly, the user will miss fine details that may be of interest. Second, there are also practical limitations due to the hardware of such systems. Graphical operations are computationally expensive and are often severely handicapped while executing on anything less than the fastest of machines. Display monitor screens are restricted in size and resolution, limiting how much information can be displayed at any time. These limitations are particularly troublesome in systems where processors number in the thousands. However, graphics is not the only way of presenting information to the user in a non-textual manner. Some interesting research has been done to investigate auralization (sound) to characterize the behavior of a program [2].

3 An Overview of Enterprise

Rather than provide a complete introduction to *Enterprise*, this paper describes only the salient features of its architecture and programming model, with enough environment features to understand the performance tuning tools.

3.1 The Enterprise Architecture

The *Enterprise* system consists of four components: a pre-compiler, the *Enterprise* executive, a communication manager and the object-oriented user interface. The user provides the remaining components: an *Enterprise* program consisting of a modularized C program, together with any user-required libraries and a meta-program for describing the parallelism desired.

A uniform graphical user interface provides the mechanisms for modularizing the components of a user's program and selecting the desired parallelization techniques. All the necessary communication and synchronization code for parallel execution on a selected group of workstations is automatically inserted into the user's code by the pre-compiler when the program is compiled. The communication manager, currently PVM [3], is transparent to the user. The *Enterprise* executive takes care of all the process management and synchronization. A conventional C programmer need not learn any new language, language constructs, or libraries of special functions for handling inter-process communication or heterogeneity concerns. This, in itself, is a major step towards reducing programming errors which result from the use of unfamiliar language constructs or specialized library calls. A simple menu-based graphical meta-programming model for expressing parallelism is really the only new symbolic construct for the programmer to learn.

The *Enterprise* model provides the user with templates to implement various parallelization techniques. As such, it can be categorized as a template-based PPS. Arbitrary communication patterns among concurrent processes are not allowed. The process communication graph is determined by the selected templates.

3.2 The Programming Model

In distributed systems with multiple agents working together towards a common goal, the method of communication is message passing. The traditional programmer's view of message passing involves four steps: pack the data, send the message, receive the message, and unpack the data. The *Enterprise* approach to message passing is to make inter-process communication look like standard C function calls, but without the programming effort and synchronous semantics of remote procedure calls. *Enterprise* executes designated function calls concurrently. Consider the following C code:

```
result = func( x, y );
/* other C code */
a = result;
```

If `func` is to be executed concurrently, then *Enterprise* modifies the procedure call to pack the parameters `x` and `y` into a message and sends it to the processor that executes `func`. The caller continues executing and only blocks and waits for the function result when it is needed (`a = result`). A pending result that allows concurrent actions has been called a *future*.

By letting the compiler generate the code to do data packing and unpacking, programming errors related to message packing are entirely avoided. Any heterogeneity concerns, such as byte-ordering schemes of the communicating processors, are handled automatically. For example, off-by-one errors in arrays are eliminated.

3.3 The Meta-Programming Model

There are no explicit references to parallelism in the user's code. The parallelism is described orthogonally in a meta-program constructed using the graphical user interface. The meta-programming model employs an analogy with a business organization to represent different types of parallelization techniques. There is an intrinsic similarity between the units of a business organization, all ultimately composed of individual workers, and the groups of individual processes executing in a distributed computing environment. This analogy provides us with consistent terminology and a metaphor for building hierarchical parallel structures.

Every program entity in *Enterprise*, be it an individual module or an integrated group of modules, is called an *asset*. Each asset is one of: Enterprise, Individual, Service, Receptionist, Representative, Line, Department or Division. Lines, Departments, and Divisions are composite assets representing different parallelization techniques. The user draws a simple hierarchical asset diagram to represent a parallel program. A new *Enterprise* program starts with an Enterprise asset containing a single Individual. If the entire program code is placed in this Individual, the program will simply execute sequentially, analogous to a one-person business. Four basic operations are used to transform this sequential program into a parallel one: asset expansion, asset transformation, asset addition and asset replication. Using the analogy, the simple business grows incrementally into a complex organization.

Figure 1 shows the meta-program for a simple program called *CubeSquare*. It consists of a Department with a Receptionist, `CubeSquare`, and two Individuals, `Square` and `Cube`, each replicated twice. `CubeSquare`, `Square` and `Cube` are C-code procedures. `Square` and `Cube` are simple routines that contain a randomized sleep to imitate a computationally intensive function which takes a non-determinate amount of time to execute. `CubeSquare` calls `Square` and `Cube` in a loop and then sums these results in another loop. If `Cube` and `Square` were indeed computationally intensive functions, *Enterprise* would deliver speedups. Clearly, even without the randomized sleep, the message-passing and process-management overheads in this minimalist example overwhelm any parallelization gains. This trivial example is used for illustration purposes throughout this paper.

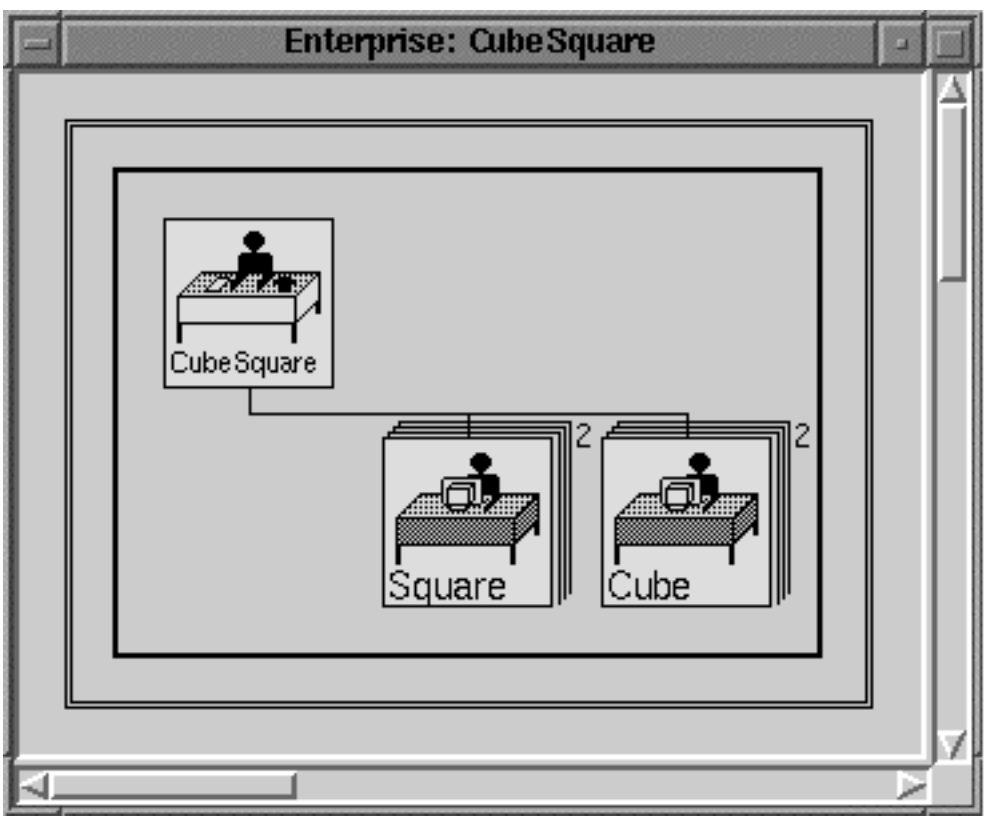

Figure 1. The meta-program for `CubeSquare`.

4. Performance Debugging in Enterprise

Enterprise supports three techniques for performance debugging. The first is a post-mortem animation of the execution of a program [5]. The second is a set of real-time performance monitoring graphs [8]. The third is the ability of *Enterprise* to automatically allocate resources based on load. The first two will be discussed in this section and the third will be described in a future paper.

4.1 Event Acquisition in Enterprise

There are three acquisition modes that can be selected at run-time. The first is a *quiet* mode where no event information is gathered and no analysis is performed. This mode is used for production versions of the program where performance tuning and program debugging are no longer an issue. The second mode, *performance logging*, is used to acquire real-time performance data including event type, the ids of the collaborating processes and an event time stamp. The third mode is *event logging* and it is used to capture complete events to a trace file for program replay, animation, debugging and post-mortem analysis. The event information gathered during event logging is a super-set of the information gathered during performance logging. The additional information includes the contents of messages (user parameter values).

The data acquisition mechanism is shown in Figure 2. At run-time, if either logging mode is enabled, each *Enterprise* asset sends a special event logging message to the *Enterprise* executive process whenever a message is sent between assets. If event logging is enabled, the executive process writes the gathered event messages to an event log file. This file is used in the animation and replay of a given program run. If performance logging is enabled, the executive process also forwards the event logging message to the *Enterprise* interface process via a UNIX™ (trademark of Unix Systems Laboratory, Inc.) pipe. The interface then parses the input from the pipe to obtain the raw event data.

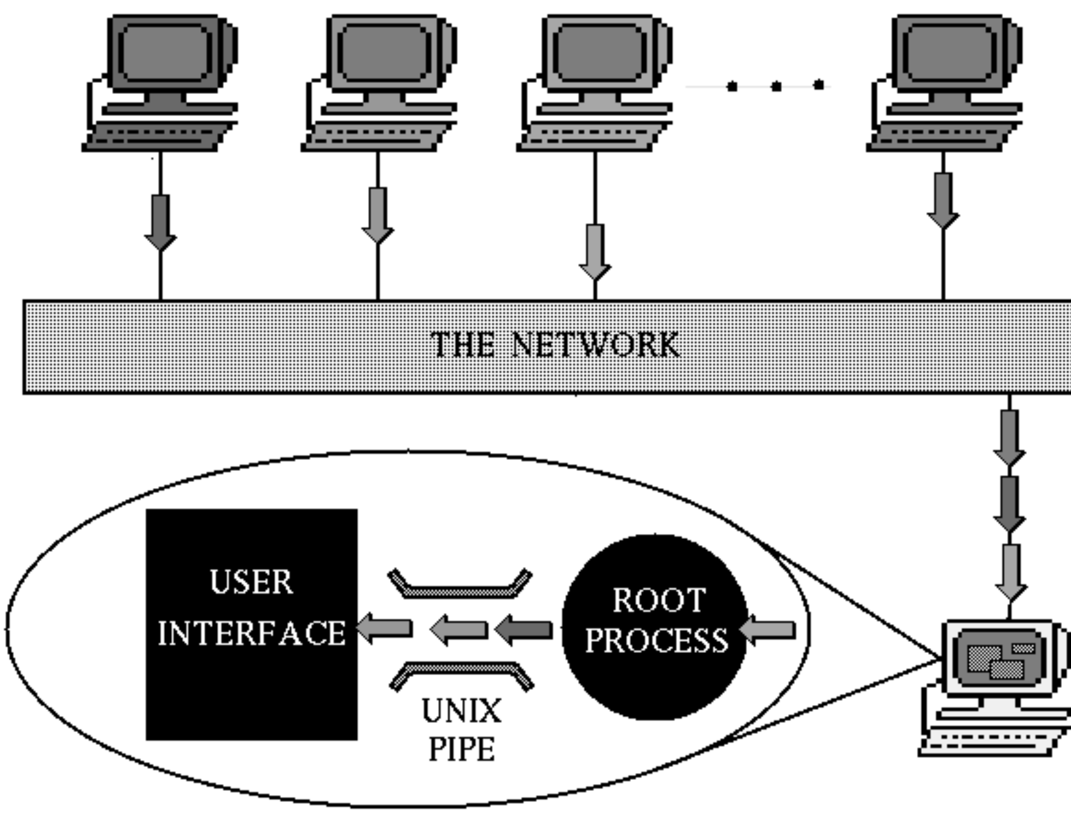

Figure 2. The Data Acquisition Mechanism.

Unlike event logging where event data is written to a file, the performance logging mechanism manages the data as it arrives in one of two user specified modes: *ordered* or *chaotic*. Ordered mode processing requires that the user specify a delay time measured in seconds. In this case, a buffer holds events taken from the pipe for the delay time before processing them further. This buffering compensates for mis-ordered events that may arise as a result of uneven processor-to-processor message delays introduced by a congested network. The end result is that the analysis will have a minimum delay time before displaying its results. In the chaotic mode, events are processed as they are taken from the pipe. If events from different processors arrive out of order, a warning message is displayed, but no further action is taken and event processing continues. Chaotic mode is sometimes useful since it reduces the overhead associated with buffering and ordering events and may be safely used in the presence of a fast reliable network with low congestion.

4.2 Performance Analysis in Enterprise

Analysis is based on asset state information and the history of each message sent between assets during a program run. At any time during the execution of a program, an *Enterprise* asset can be in one of four states: *busy* doing user's work, *idle* waiting for work, *blocked* waiting for a future to be returned, or *dead* when it is finished all of its work. In the best-case scenario, an asset will be busy for most of its life. The history of a message consists of which asset sent the message, which asset received the message, the amount of time the message was queued before it was processed, and the amount of time required to process the message. This simple information can provide a great deal of insight into an *Enterprise* program.

The processing method is essentially the same for post-mortem and real-time analysis with the chief difference being the source of the raw events. For real-time processing, the events are obtained from the UNIX pipe linking the executive process and the interface. For post-mortem processing, the events are obtained from a trace file generated during an execution run. In either case, the first step of the analysis process is to register the raw events. Each raw event is accepted by the interface and its effects are computed and registered in an internal data structure that maintains statistics for each asset and each message sent by an asset.

In real-time processing, an attempt is made to register events at speeds matching actual execution of the program. However, such a scenario is not always possible. There are times when raw event processing must be slowed, such as when messages experience unusually long delays over a network. In an effort to preserve some consistent model of time, the performance analysis tool maintains two timers. The first, *real time*, records the true wall-clock execution time of the program in seconds. The second, *virtual time*, records the time based on event processing. If events can be registered at a speed comparable to execution time, real time and virtual time will proceed at an equivalent rate. However, should the system be unable to process raw events quickly enough, the rate at which virtual time passes will be slowed by some factor. For example, virtual time may flow at half real time or one quarter real time. This approach preserves relative time spans in virtual time and preserves the appearance of how the program actually executed.

The second step of the analysis process is to take *snapshots* of the system state for real-time display purposes. Although each event is registered as it arrives at the interface, it may be too expensive to perform a detailed analysis of each event and present the results graphically, especially if bursty activity overloads the analysis mechanism or the display mechanism. Instead, the user can specify a sample interval in seconds such that a snapshot is taken once each interval. For example, if the interval is set to one second, the state of each asset will only be updated each second, even if it changed state several times during this period.

4.3 Post-mortem Presentation: Animation

An *Animation View* allows the user to animate the message-passing behavior of a program based on a captured event file from a previous execution. This view provides the simultaneous macroscopic perspective of the program together with a microscopic perspective of the state of individual assets and messages.

Once an event file has been created, the animation view can be used for both performance debugging and correctness debugging. The captured events are animated as a sort of time-lapse movie which shows messages moving between assets, entering and leaving message queues, and assets changing state. Figure 3 shows an animation view of the program *CubeSquare*.

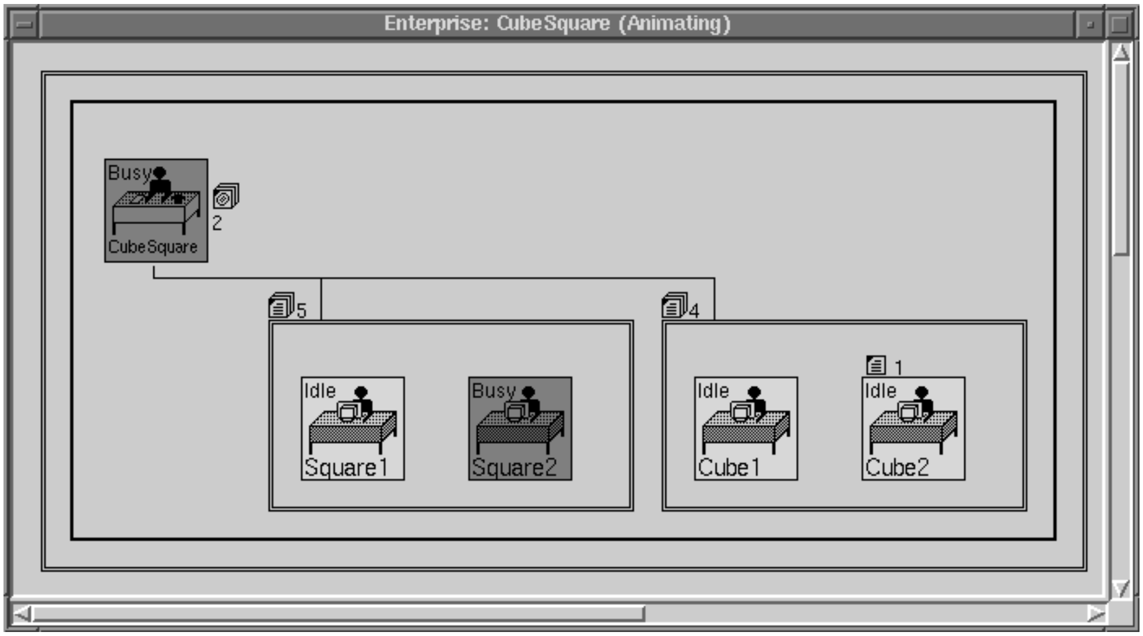

Figure 3. `CubeSquare` animating.

In the animation view, the user can, in effect, simulate a particular execution without actually re-running the program. The post-mortem analysis of a program in this view can reveal important performance characteristics and help to detect various programming errors. By viewing the animation, the user can dynamically observe the degree of parallelism, the relative states of processes, the buildup of messages in message queues, and the values of logged parameters in the messages.

The animation view displays the assets of the program in their fully expanded form, expanding all replicas, or with assets selectively collapsed to hide uninteresting detail. Each asset is labeled by the state that it is in (IDLE, BUSY, BLOCKED or DEAD) and every replica is uniquely named.

In addition, each asset has two message queues: an input queue at the left of its top edge for call messages and a reply queue at the middle of its right edge for return messages. If the queues are empty, nothing is displayed. During animation, messages move along paths on the screen between assets and enter the message queues, which are then visibly displayed by a message icon and a number designating the number of messages in the queue.

At any time, the user can stop, resume, single step or restart the animation. When an animation is stopped, a window with a Message Queue List (Figure 4) for any of the visible message queues can be opened by using the menu associated with the message queue icon. A message selected from the list in the top pane of the window has its captured parameters displayed in the bottom pane. Messages in the list are briefly identified by their sender and a tag. To illustrate how logged arrays are displayed, the *CubeSquare* code was slightly modified to include an array of ints in the parameter list for `Square`. Here, the user had decided to log `y[1]`, `y[2]`, `y[3]`, and `y[5]` (but not `y[0]` or `y[4]`) when the asset `Square` was called. These Message Queue Lists can be kept open when animation is resumed. They will update automatically as messages enter and leave the queues. The ability to inspect parameter and return values in these message lists is a valuable debugging aid.

Replicas of an asset are displayed within a bordered rectangle which represents a manager process for the replicas. The manager receives all the calls to the asset and then looks for an idle replica to respond to the call. Each of these replicated asset managers has its own input queue, but replies from the replicas animate directly back to the original caller. The name and state of a manager are not displayed.

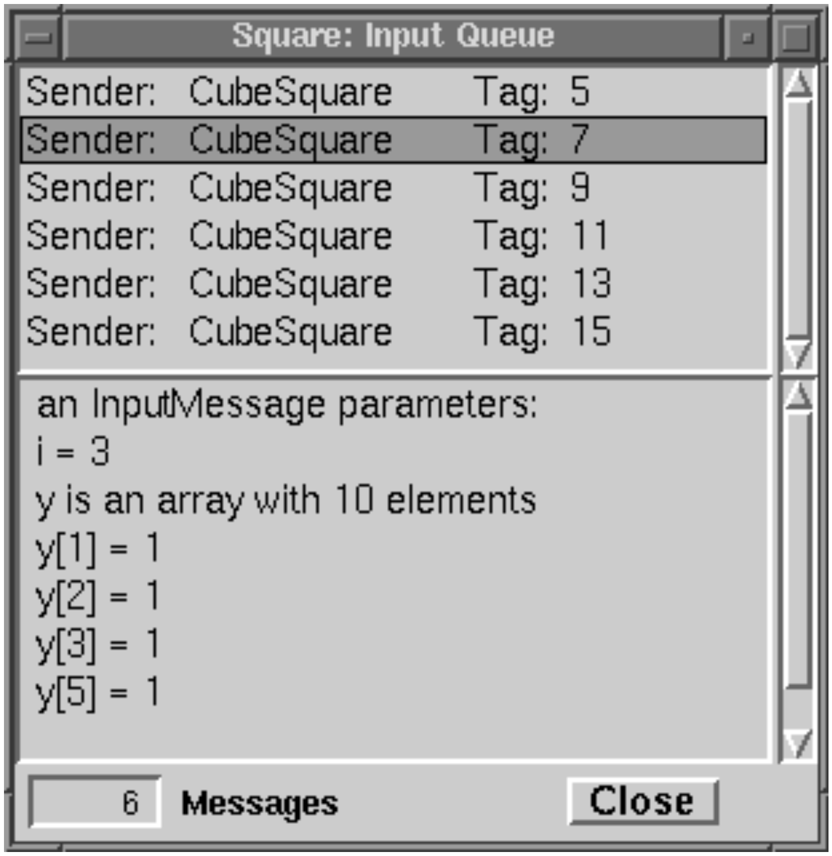

Figure 4. Message Queue List.

4.4 Real-time Performance Presentation in Enterprise

Real-time information from the analysis phase is displayed in *Enterprise* using interactive graphical displays that provide different views of the processor utilization and communication trends, as gleaned from events gathered during an execution run. Each graphical display is updated once every sample interval during real-time processing. Once execution is complete and the events have been registered, it is possible to alter sample rates or play events in reverse order. It is also possible to use these views for post-mortem analysis.

The *Asset Utilization View* shown in Figure 5 displays the relative amount of time spent in each of the three working states from the execution start to the time indicated in the *Current Time* field. Red indicates time spent waiting, green indicates busy time and yellow indicates time spent idle. The numerical values beside the asset represent the number of messages waiting in the input queue (upper number) and output queue (lower number) of the asset.

The *Message View* in Figure 6 shows, for each asset, the message passing history of the asset. Each asset is represented by a time-line with a link being drawn from the asset sending the message to the asset that receives the message (similar to other systems, such as [7]).

The *Annotation View* shown in Figure 7 is a highly configurable display which allows the user to select special situations to be identified. Whenever one of these special situations occurs, a visual annotation appears on a graphical time-line display of the execution run. The annotation pinpoints exactly when the event occurred. This annotation can be selected from the display to provide more information. In addition, the user can specify that all graphical views *jump* to the time as indicated by the annotated event to provide further insights.

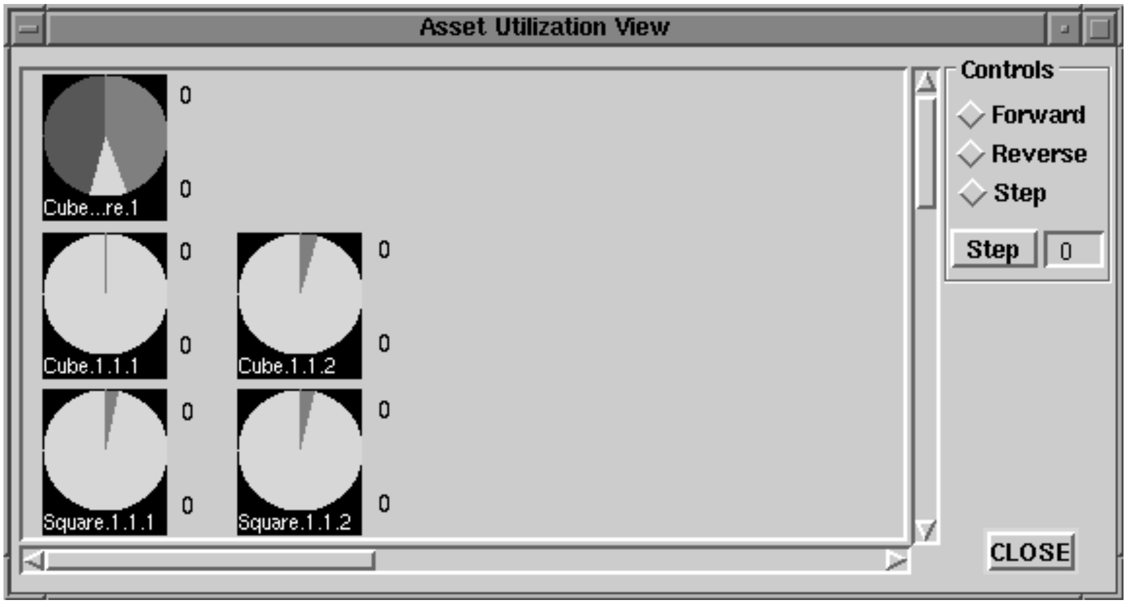

Figure 5. The Asset Utilization View.

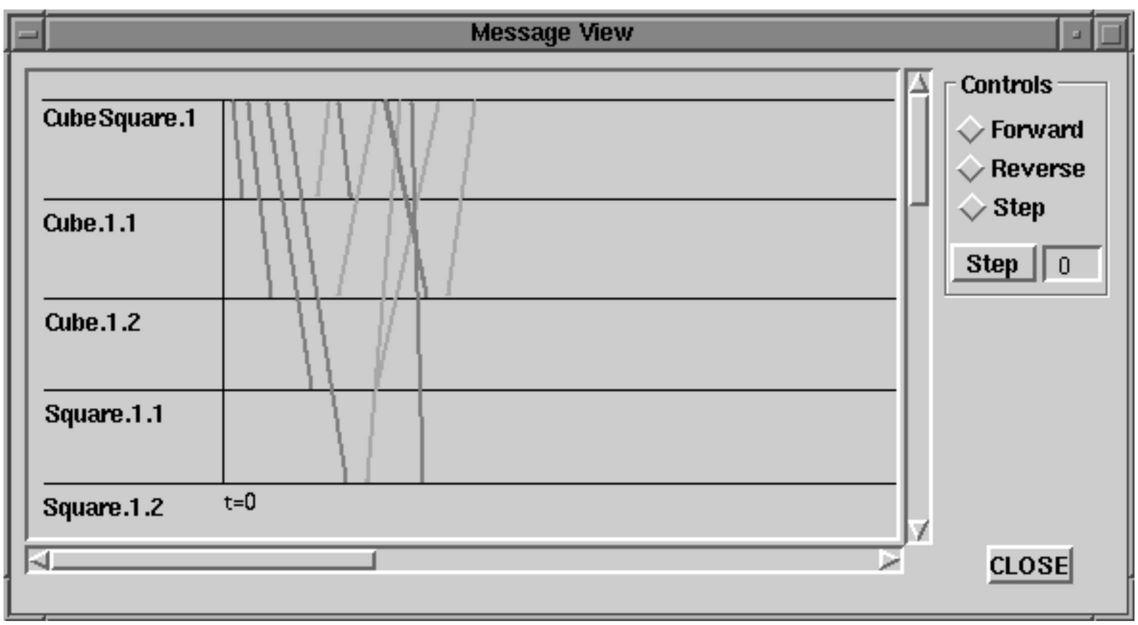

Figure 6. The Message View.

For example, in Figure 7 the Road Runner icon shows where the program achieved a good speedup and the Coyote icon shows where a slowdown occurred. Each of these events can be annotated with a comment indicating the probable cause of the event and an indication of where in the user's code the performance problem occurred. Thus the user knows exactly where potential problems lie. The situation choices are obtained from a menu that includes such utilization statistics as *register on first speed-up*, *register on all speed-up* and *register on first slow-down*. It also includes such aggregate operations as *asset under-utilized*, *asset over-utilized*, *overloaded message queue* and *excessive message processing time*.

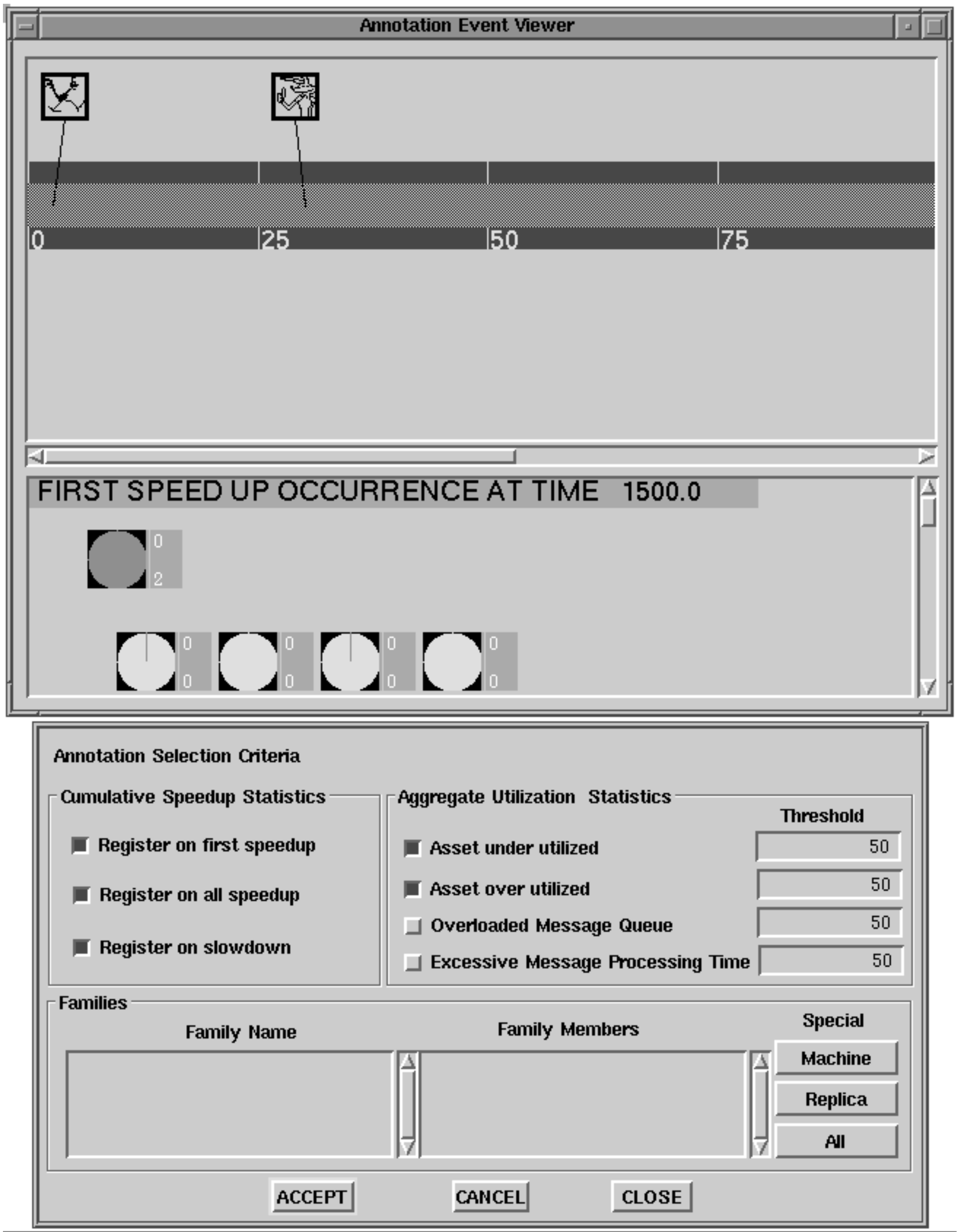

Figure 7. The Annotation View.

The aggregate operations are based on using threshold-variance techniques defined over a user specified group, or *family*, of assets. After specifying which assets constitute a family, the user specifies a variance threshold such that should any asset vary from the group average by more than the threshold value the situation is flagged on the display. This selection process is fully interactive and may be done at any time. This form of performance analysis provides an elegant mechanism for a user to be provided subjective performance information at a high level in a way that is quick to recognize.

We introduced audio feedback into our performance presentation by playing a happy tune for busy assets, a sad tune for idle and blocked assets and a somber tune for dead assets. Unfortunately, the additional overhead slowed down the presentation too much. However, we are continuing to look at audio techniques, since they provide identifiable feedback on a different channel than the often overloaded visual channel.

5. Conclusions and Future Directions

Enterprise provides several views of an executing program (snapshots, animation, auralization, performance highlights). The combination is a powerful tool set for quickly identifying performance problems. These tools are at a high-level. The user is given direct feedback as to which assets need attention and the lines of code where beneficial changes can likely be made. The tools are also seamlessly integrated into the environment so that the user does not have to learn an *add-on* performance debugging tool.

Another powerful performance enhancement is being incorporated into *Enterprise*: *automatic* performance tuning. In *Enterprise*, each process has all of the code necessary to perform the tasks of any asset and to execute the code either sequentially or in parallel. For example, we can execute a recursive call in a division either sequentially or in parallel. By gathering data, an *Enterprise* asset can decide whether to actually launch an asset call as a process or to execute it sequentially if the granularity is small. Such dynamic performance tuning can be used to correct user parallelization errors, such as choosing a granularity that is too small.

Acknowledgments

This research was supported by the Natural Sciences and Engineering Research Council of Canada and by a Ph.D. fellowship for Ian Parsons from the Center for Advanced Studies, IBM Canada.

About the Authors

David Woloschuk is completing a M.Sc. program at the University of Alberta. His main interests are in the analysis and visualization of distributed parallel programs. He received his B.Sc. from the University of Alberta. His Internet address is davidw@cs.ualberta.ca.

Paul Iglinski is a programmer-analyst at the University of Alberta. He received a B.A. (Math) from the University of South Florida, an M.A. (Chinese) from Stanford University, and a B.Sc. and M.Sc. from the University of Alberta. The focus of his research was distributed parallel debugging and object-oriented user interfaces for parallel computing. His Internet address is iglinski@cs.ualberta.ca.

Steve MacDonald is in the M.Sc. program at the University of Alberta. His main interest is run-time systems for distributed parallel programming environments. He received a B.Math. from the University of Waterloo. His Internet address is stevem@cs.ualberta.ca.

Diego Novillo is in the Ph.D. program at the University of Alberta. His main interest is distributed shared memory for parallel programming environments. He received a B.Sc. from CAECE University, Buenos Aires, Argentina. His Internet address is diego@cs.ualberta.ca

Ian Parsons is in the Ph.D. program at the University of Alberta. His main interest is in programming environments for distributed parallel applications. He received a B.Sc. and M.Sc. from the University of Alberta, and a B.Sc. (Chemistry) from the University of Western Ontario. His Internet address is ian@cs.ualberta.ca.

Jonathan Schaeffer is a Professor of computing science at the University of Alberta. His research interests include parallel computing (programming environments and algorithms) and artificial intelligence (heuristic search). He received a Ph.D. and M.Math. from the University

of Waterloo and a B.Sc. from the University of Toronto. His Internet address is jonathan@cs.ualberta.ca.

Duane Szafron is an Associate Professor of computing science at the University of Alberta. His research interests include object-oriented computing, programming environments and user interfaces. He received a Ph.D. from the University of Waterloo and a B.Sc. and M.Sc. from the University of Regina. His Internet address is duane@cs.ualberta.ca.

References

[1] A. Beguelin, J. Dongarra, G. Geist, R. Manchek and V. Sunderam. Graphical Development Tools for Network-Based Concurrent Supercomputing. *Supercomputing '91*, pp. 435-444, 1991.

[2] J. Francioni and J. Jackson. Breaking the Silence: Auralization of Parallel Program Behavior. *Journal of Parallel and Distributed Computing*, vol. 18, pp. 179-194, 1993.

[3] G. Geist and V. Sunderam. Network-Based Concurrent Computing on the PVM System. *Concurrency: Practice and Experience*, vol. 4, no. 4, pp. 293-311, 1992.

[4] P. Iglinski. An Execution Replay Facility and Event-based Debugger for the Enterprise Parallel Programming System. M.Sc. thesis, Department of Computing Science, University of Alberta, 1994.

[5] G. Lobe, D. Szafron and J. Schaeffer. The Enterprise User Interface. *TOOLS (Technology of Object-Oriented Languages and Systems) 11*, R. Ege, M. Singh and B. Mayer (editors), pp. 215-229, 1993.

[6] J. Schaeffer, D. Szafron, G. Lobe and I. Parsons. The Enterprise Model for Developing Distributed Applications. *IEEE Parallel and Distributed Technology*, vol. 1, no. 3, pp. 85-96, 1993.

[7] D. Taylor. A Prototype Debugger for Hermes. CASCON '92, pp. 29-42, 1992.

[8] D. Woloschuk. Analysis and Display of Parallel Program Performance Information within the Enterprise System. M.Sc. thesis, Department of Computing Science, University of Alberta, 1995. In preparation.