

Xunify – A Performance Debugger for a Distributed Shared Memory System

James E. Lumpp, Jr.[†],
Kuppuswamy Sivakumar[†],
Christopher Diaz[‡],
James N. Griffioen[‡]

Department of Electrical Engineering[†]
Department of Computer Science[‡]
University of Kentucky
Lexington, KY 40506, USA
{jel,siva,diaz,griff}@dcs.uky.edu

Abstract

Distributed shared memory (DSM) offers advantages over message passing systems by allowing programmers to use a shared memory programming model. However, distributed shared memory systems present new challenges to performance evaluation and debugging. While programmers write programs using a shared memory model, the processes in the distributed system share information via messages. This abstraction can make it difficult to identify the sources of performance problems. We have developed an instrumentation system and performance evaluation tool called Xunify for monitoring and performance debugging of a DSM system. Xunify provides information in terms of the DSM program level, logical message level, and the transport protocol level.

1. Introduction

Multiprocessor computers once served as the primary platform for high-performance parallel applications. However, the recent development of high-speed system area networks and local area networks combined with the continued increase in speed of general purpose processors has resulted in a trend toward the design and implementation of powerful multicomputers consisting of networks of workstations. Programming models for distributed systems offer the same abstractions as those used on tightly coupled multiprocessor machines and fall into two categories: *message passing pro-*

gramming models and shared memory programming models. Recent distributed object programming models typically build on one of these two underlying computing models.

When a programmer ports shared memory applications to newer distributed shared memory (DSM) architectures, it is possible for them to use the same programming model, however, they will find it more difficult to optimize the performance of the application.

Tuning application performance is complicated by several factors. First, multicomputer systems often lack the uniformity of multiprocessor systems. The workstations that comprise the multicomputer often differ in their processor speeds, memory capacity, available disk space, or architecture. Second, the programming model is largely implemented in software as opposed to hardware (e.g., memory sharing and caching). Also, network communication protocols and routing is primarily implemented in software as opposed to hardware. This can significantly increase the overhead of certain abstractions or features of the programming model (e.g., handling a page fault in a DSM model). Third, network communication is not reliable. Packets can become corrupted in transit, dropped by switches as a result of network congestion, or dropped at receivers as a result of buffer overflow. In multicomputers spanning large networks, additional problems such as high variance in latency, intermittent routing problems and disconnections can

also occur. Network errors typically result in retransmissions which can further congest an already congested network path or host.

Typical performance analysis tools focus on displaying information related to the programming model. They often assume uniformity among processors that communicate over reliable uncongested interconnection networks. However, fine tuning the performance of multicomputer applications requires additional functionality from the performance monitoring tool. Not only must the tool be able to display information that relates to the programming model, but the tool must also provide information about the underlying multicomputer architecture and associated bottlenecks.

This paper describes the design of the *Xunify* performance monitor. *Xunify* is a graphical performance monitoring tool designed to work with the Unify distributed operating system [8]. The tool helps the programmer visualize the performance of an application while it is running and can save trace information for repeated postmortem visualization. *Xunify* is unique in the fact that it provides performance information at three different levels: (1) the DSM programming model level, (2) the logical message send and receive calls, and (3) the low-level transport protocol details. *Xunify* generates trace information that can help programmers discover the cause of a performance degradation without knowing the specifics of how a layer (particularly lower layers) work (for example, summary information might indicate that there is high packet-loss/congestion on the link connecting machines A, B, C, and D). *Xunify* also includes the concept of *abstraction filters* which allow the user to selectively view all the actions related to a particular abstraction such as a shared memory region, a synchronization event, consistency events, etc. Finally, *Xunify* introduces new ways to view the behavior of a running DSM application. Although we target our implementation to Unify, the basic principles can be applied to any DSM or distributed object system running on a multicomputer.

The remainder of the paper is organized as follows. Section 2 describes existing performance modeling tools and the environments they target. Section 3 then gives a brief overview of the Unify distributed shared memory system which *Xunify* is designed to evaluate. Section 4 describes the design of the *Xunify* performance tool, its novel features and the views it supports.

2. Related Work

Several tools have been developed to analyze the performance of programs that execute on networks of machines programmed with both a DSM model and a message passing model. SVMview [2] analyzes the movement of pages on various DSM systems. Like *Xunify*, SVMview provides trace information at the DSM level as well as a level of various services such as synchronization and shared memory allocation. However, *Xunify* additionally provides trace information of the low-level communication. SVMview also provides information to the user about synchronization costs so the user may decide to identify and remove unnecessary synchronization. *Xunify* also provides information on synchronization and provides information for the user to discover synchronization overhead that is caused by problems such as load imbalances and high network latencies.

Rajamony and Cox [11] introduce a performance debugger that detects excess synchronization in shared-memory parallel programs. Rather than focusing on graphical displays for trace information, their system analyzes trace information to determine occurrences of unnecessary synchronization and excessive synchronization. Such occurrences are then reported to the user. Like *Xunify*, the performance debugger uses trace information created in a software DSM environment. While *Xunify* does not yet automatically analyze trace information to find excess synchronization, *Xunify* does provide the information for the user to deduce such inefficiencies.

Like *Xunify*, ParaView [13] provides graphical information on shared memory parallel programs to assist the user in identifying performance problems such as load imbalances, contention for data, and unnecessary or inappropriate synchronization. Unlike ParaView, *Xunify* is designed to work with heterogeneous architectures across various local and wide area networks.

PARADE [16] [14] and TAU [10] both provide graphical environments to monitor the execution of a parallel application on a multiprocessor or message passing system, although TAU focuses more on the generation of trace information. Unlike both of these systems, *Xunify* portrays multiple levels of communication detail for the user.

Xunify traces the behavior of DSM systems at three distinct levels providing information that is useful to both application developers and the DSM library developers. *Xunify* also allows the user to filter the views to display rele-

vant information for a particular performance evaluation or debugging task. For example, the user may filter displays to show only activity relating to a particular region of shared memory or to a set of synchronization variables.

3. An Overview of the Unify DSM System

Before describing the Xunify performance analysis tool, we provide a brief introduction to the Unify system for which Xunify was designed. Although we have selected Unify as the multicomputer system for the prototype of our performance monitoring tool, the same techniques can be applied to any DSM system or distributed object system.

Unify [8] is a software DSM system designed to scale across hundreds or thousands of computers potentially spanning large geographical areas. Unify attempts to eliminate communication whenever possible, use asynchronous communication to mask latency, and reduce the amount of data being transmitted because wide area environments are characterized by high latencies and (relatively) low bandwidth. Unify's goal is to minimize the overhead needed for DSM to achieve performance competitive with that of message passing systems such as PVM [15] and MPI [6].

The keys to reducing DSM overhead in Unify is the use of new relaxed-consistency models that relax the consistency requirements on the DSM, support for low-overhead synchronization models so that applications with simple synchronization requirement need not pay the high cost of conventional synchronization abstractions, and efficient reliable multicast communication of data (via the TMTP protocol) [17].

The following briefly describes a few of the abstractions and operations supported by the Unify DSM system which are evaluated by the Xunify performance monitoring tool.

3.1. Unify Memory Model

The Unify memory model provides a single large virtual address space shared by all machines. Memory is allocated in segments. Consistency guarantees differ from segment to segment and are defined when the segment is first allocated. In addition, each segment has a "segment type" that defines how the segment will be used. Unify supports three types of segments: *Random access segments* are randomly addressable. *Sequential access segments* are accessed in a read/front, write/append fashion. *Associative ac-*

cess segments are accessed via <key,value> pairs. Sequential access and associative segments can often be supported with weaker spatial consistency guarantees (described below) that can be implemented more efficiently with fewer high latency messages than random access memory segments. The Xunify tool is aware of a segment's type and consistency requirements and monitors all consistency related activities for each segment as well as accesses and updates made to a segment.

3.2. Unify Consistency Models

Unify allows applications to select appropriate consistency semantics for each segment from a spectrum of consistency protocols, each having a temporal component (defining when the replicas become consistent) and spatial component (defining when the data order is considered consistent).

There are two types of temporal protocols: "application-aided" methods where the application specifies the consistency checkpoints (examples include *release consistency* [5, 1] and *entry consistency* [4]) or "automatic" methods where the operating system automatically ensures consistency. Unify supports a novel automatic consistency model called *eventual consistency* in which the memory becomes consistent after some time lag t . This type of consistency is particularly useful for applications that can detect stale data such as Grapevine [3].

In-order and out-of-order spatial consistency determines the ordering of data in a replica. For many distributed applications that use keyed lookups or sequential access, the order of the data items within a segment is unimportant; only the data values are important. Relaxing the spatial ordering constraint helps avoid the high network latencies and serialization that would otherwise occur when writing to a sequential or associative segment.

Xunify monitors all consistency related operations that occur on each segment. In the case of user-aided consistency models, this involves recording all calls to update shared memory copies. For automatic methods, it requires monitoring all segment transfer operations performed by the Unify operating system. In addition, Xunify monitors all cases of false sharing of segments that are detected by Unify, including the precise moment the false sharing begins and the time at which the writes to a common segment are merged together.

3.3. Unify Synchronization

To support low-overhead synchronization, Unify supports an *eventcount* [12] abstraction that uses unreliable multicast communication to update synchronization information on all machines. Unify also provides a migrating ticket master that hands out *sequencers*. Sequencers and eventcounts can be used to efficiently implement other synchronization primitives such as locks, semaphores, and barriers.

The use of unreliable multicast to update information improves performance but means that synchronization updates may be lost and thus some hosts may not have up-to-date synchronization information. The Xunify tool monitors the synchronization state of the system and is able to detect when such inconsistencies arise.

3.4. Unify's Communication Protocols

Unify employs an efficient reliable multicast protocol called TMTP [17] for large scale dissemination of shared data. To provide reliability, TMTP uses a combination of sender and receiver initiated approaches. The sender initiated component uses a novel window-based and rate-based flow control mechanism with positive cumulative acknowledgments, timeouts, and retransmissions. The receiver initiated component uses negative acknowledgments with NACK suppression to achieve quick recovery from lost messages. All retransmissions use a "restricted scope" multicast message to limit the retransmission to the area where the error occurred. As a result, retransmissions are handled locally in a timely fashion, thereby avoiding retransmissions to the entire Internet. The use of localized NACKs with nack suppression ensures quick response to lost messages with minimal overhead.

Performance of DSM systems depend on the network location of the workstations that make up the multicomputer and their communication patterns. Even if an application is written correctly, the way in which the problem is partitioned and allocated to hosts can result in network congestion that severely affects performance. To detect these situations, Xunify also monitors all events that occur within the communication protocol.

4. Xunify

Xunify provides a variety of graphical displays to help analyze programs. High-level views of program execution help application programmers identify algorithmic performance problems in applications and lower-level views help programmers and DSM developers correlate program level constructs to message activity and protocol behavior.

The Xunify system consists of three primary components:

- **instrumentation code** added to the Unify operating system to generate the necessary trace information;
- **trace routing mechanism** used to route trace information from the Unify processes to the run-time display, and;
- **graphical display** that illustrates an application's behavior either during runtime or during a post-mortem phase via a trace file.

The following sections describes the three components of the system and show how the information displayed by the Xunify tool can be used to improve application performance.

4.1 Trace Instrumentation

There are several factors that can influence the performance of a distributed application running on a multicomputer comprised of network of workstations. Algorithmic inefficiencies such as excessive synchronization, non-optimal partitioning, "hot" data, false sharing and other problems can severely affect application performance. The way the application is mapped onto the resources of a multicomputer can also severely affect an application's performance. For example, if two processes that communicate frequently are placed on distant machines, network delay will limit performance. Alternatively, if the two processes must communicate over a congested link, packet loss will affect performance. In most cases, an alternate mapping of the problem onto the multicomputer resources can significantly enhance performance. Xunify monitors the system at three distinct levels to capture a complete picture of the system's performance: (1) the programming model level, (2) logical message level, and (3) transport protocol level. In Xunify, the "programming model level" monitors application-level abstractions such as shared memory

segment modifications, synchronization access, and consistency checkpoints. The “logical message level” records information about the any communication that may result from the invocation of abstractions at the programming model level. Finally, the “transport level” monitors events related to the underlying transport communication protocol.

4.1.1 Programming Model Monitoring. Monitoring information recorded at the programming model level is particularly useful for identifying algorithmic performance problems such as excessive synchronization or improper partitioning. To identify such problems, Xunify monitors the following programming model level events.

- *Create_Seg* events mark the point at which allocates a segment out of the global address space.
- *Put_Updates* events mark the point at which a process explicitly transmits updates of a segment to every member in the segment’s copy set.
- *Get_Updates* events mark the point at which a process issues a request for updates to a particular segment.
- *Need_Updates* events mark the point at which a process issues a request to join the copy set of a particular segment.
- *Dont_Need_Updates* events mark the point at which a process issues a request to leave the copy set of a particular segment.

Each of the programming model level events include the associated segment identifiers. In addition, *Put_Updates*, *Get_Updates*, *Need_Updates* and *Dont_Need_Updates* events include information about the associated sequence numbers.

The following events are instrumented to analyze the high-level synchronization behavior of Unify programs:

- *Create_Event* events mark the point at which a process allocates an event.
- *Advance* events mark the point at which a process issues an advance on an event.
- *Await* events mark the point at which a process waits for an event to reach a certain value.
- *Await_with_Timeout* events mark the point at which a process waits for either an event to reach a certain value or for a specified time period, whichever occurs first.

- *Event_Sync* events mark the point at which a process requests to bring the local count of a particular event into a consistent state.
- *Get_Ticket* events mark the point at which a process issues a request for a ticket.

Each of the high-level synchronization events include the associated event or sequencer identifier. In addition, the *Advance*, *Event_Sync* and *Get_Ticket* events include the associated sequence number. All of this information is available to the programmer in various views with Xunify.

Unfortunately, the information gathered at the programming model level often cannot identify communication inefficiencies. Although a goal of DSM systems is to hide communication from the programmer, this transparency may cause programmers to unknowingly select an inefficient implementation. This is particularly true of multicomputers, which also hide the nonuniform network speeds and bandwidths from the programmer. Such information is useful to obtain optimal performance.

Xunify monitors communication information at two levels: the logical message level and the underlying transport level. At the logical message level, Xunify records logical communication patterns that corresponds to request messages and reply messages. At the transport-level, Xunify records the details of reliable transmission (timeouts, retransmissions, acknowledgments, etc.) used to implement the logical message level.

4.1.2. Monitoring Logical Messages. Monitoring information obtained at the logical message level can be used to identify both algorithmic and resource mapping problems. For example, viewing the logical messages produced at the logical message level might clarify the inefficiency of using an in-order sequential-access segment instead of an out-of-order sequential-access segment (algorithmic problem). Alternatively, viewing the logical messages needed to implement a *Put_Updates()* might point out which processes are sharing a segment and should be located on the same network.

There are basically four events monitored at the logical message level:

- *Msg_Send_Begin* events mark the point at which a process begins sending out a logical message.
- *Msg_Send_End* events mark the point at which the send operation unblocks, allowing the application to con-

tinue its processing. If the send operation is asynchronous, `Msg_Send_End` may occur before the data arrives at the destination. If the send operation is synchronous, `Msg_Send_End` indicates the data has arrived at the destination.

- `Msg_Receive_Begin` events mark the point at which a process begins waiting for a logical message to arrive.
- `Msg_Receive_End` events mark the point at which a process receives a complete message. Note that a `Msg_Receive_End` event may occur without an associated `Msg_Receive_Begin` event if the application was not actively waiting for a message to arrive.

Each of these events includes the message type, a sequence number and context information, such as a segment identifier, to differentiate similar messages from one another. Possible message types include *data messages*, *request messages*, and *reply messages*. Data messages result from operations such as `Put_Updates()` and `Advance()` operations. Request messages result from operations such as `Get_Updates()` and `Get_Ticket()`. Reply messages are sent in response to request messages and contain the requested information or redirection information (e.g., hints regarding a segment's current location).

4.1.3. Monitoring the Transport Protocol. Transport protocol monitoring is useful to determine the characteristics of the communication paths used by an application. The logical message level gives a logical view of the messages exchanged between workers but does not display the actual messages exchanged across the network by the transport protocol. For example, a logical message may have been implemented by breaking the message into fragments that are each transmitted and acknowledged with selective acknowledgments. If some of the fragments are lost, they are retransmitted via retransmission timer expirations or selective negative acknowledgments. None of these transmission details appear at the logical message level.

In general, programmers do not (and should not) understand the details of the transport layer protocols. However, even without understanding the details, a visual illustration of the operation of the transport level protocol can offer valuable insight into a performance problem, even if the programmer knows relatively little about the protocol. These views are invaluable to the DSM library developer in tuning the performance of the DSM system itself. The

transport layer monitors all the transport protocol related events including:

- `Frag_{Send/Recv}` events mark the point at which a process sends/receives a message fragment.
- `ACK_{Send/Recv}` and `NACK_{Send/Recv}` events mark the point at which a process sends/receives a cumulative-selective ACK or selective NACK, respectively.
- `Retrans_Send` events mark the point at which a process retransmits a fragment due to a retransmission timer expiration or a NACK specification.
- `Duplicate_Frag` events mark the point at which a process receives a duplicate fragment.
- `Packet_Discard` events mark the point at which a process discards a packet.

Each of the transport level events include information about the associated sequence and fragment numbers. In addition, the `Retrans_Send` event includes the number of resend attempts issued on the fragment.

Although the details of communication can be helpful in debugging a performance problem resulting from too much communication or inefficient communication patterns, we have found that transport level *summary information* provides the most useful information to the typical programmer. Thus Xunify gathers the following summary information:

- *Transmission Rates* record the average throughput observed between processes.
- *Round Trip Times* measure the average latency between processes.
- *Packet Loss Rates* between processes is calculated from the number of packets sent and the number of packets received. These can be broken down into data packet loss rates, ACK packet loss rates, and NACK packet loss rates.
- *Number of and Percent Retransmissions* measure the frequency in which packets were resent. These figures are broken down by retransmissions that result from timeouts and retransmissions that result from NACK messages.
- *Number of and Percent Duplicates Messages and Duplicate Fragments* measure the frequency in which duplicate packets were received.
- *Retransmission Levels* records the number of fragments retransmitted once, twice, three times, etc.

Transport level summary information is particularly useful for identifying low bandwidth or high latency communication links, links with high packet loss, multiple routes between hosts, slow machines incapable handling data bursts and other problems. As a result a programmer might rearrange the assignment of processes to machines, repartition the application, use smaller segment sizes, etc.

4.2 Gathering Trace Information

It is important that an efficient mechanism be used to route trace data from the DSM processes to the performance evaluation tool. If routing trace data consumes too much of the resources of the processor or network, the behavior of the application can be affected. In addition, it is not possible to buffer all trace information locally until the execution is complete because of our goal to provide views as the applications runs.

The Parallel Virtual Machine (PVM) message passing system offered several advantages as the transport mechanism for our purposes.

PVM was developed to support message passing programming on a network of machines [15]. Due to its portability, ease of use and support for instrumentation, PVM was chosen as the mechanism to transmit trace data from each of the Unify tasks to Xunify. Although Unify does use the trace facilities provided by PVM, Unify does not use the sharing or synchronization mechanisms provided by PVM during the execution of a parallel application.

The other advantage of choosing PVM to route trace messages is that PVM 3.3 includes a tracing facility that provides support for tracing PVM applications. PVM 3.3 includes a framework for users to dynamically enable and disable the monitoring of classes of events, the ability to capture task output and route it back to the analysis tools, and provides a standard format for trace messages.

While we are not interested in monitoring PVM, by adopting the PVM trace data format for our DSM trace messages we are able to seamlessly integrate PVM instrumentation into the Unify DSM system and are able to use DSM trace data to drive an existing performance evaluation tool called XPVM [7, 9]. By using XPVM as a basis for the development of our graphical interfaces the development time for the prototype tool was greatly reduced. While modifications to the XPVM visualization tool were required to tailor it to the task of analyzing Unify programs, no modifications

to PVM were required. However, we did optimize the way trace data is handled in PVM to reduce the trace overhead incurred by a Unify application during run-time.

To minimize the overhead of routing trace data while still maintaining timely update of the displays, we took three steps: 1) Added buffering support to buffer trace messages; 2) Added the capability to route trace buffers when the Unify process is blocked due to synchronization, and; 3) Added the capability to initiate the routing of trace buffers after a specific time limit has expired. The first two capabilities reduce the overhead of tracing, while the third guarantees timely updates of the displays during program execution. There is a large amount of overhead incurred for each message routed. By buffering several smaller messages, this overhead is amortized over all the messages. In addition, there are times when processes are blocked on synchronization. If these times can be identified, the overhead to route trace data can be incurred while the process is already blocked. This effectively removes the overhead of routing trace data. Finally, to maintain the real-time update capability of displays, a timer feature was added to force buffered messages to be sent after a predefined time limit had expired. This puts an upper bound on the amount of time a trace message will be buffered.

4.3 The User Interface

The goal in the design of Xunify is to develop a trace based performance evaluation tool that is useful in both the development of Unify programs and in the analysis of Unify libraries. To achieve this, graphical views that provide feedback on both the behavior of the applications and the performance of the Unify system are required. Xunify provides a variety of graphical views that illustrate the run-time behavior of an application. The interface can display trace information saved in a trace file as part of an iterative debugging process. Additionally, the interface can display trace information as the program runs in order to track the progress of the computation and the performance achieved.

Instead of redeveloping a graphical interface, XPVM, an existing graphical tool developed to animate message passing programs, was used as a starting point. To support the unique problems involved in monitoring and tuning DSM programs, capabilities are added to XPVM, including:

- A mechanism for filtering animations to show the DSM messages that correspond to a specified set of

virtual memory segments.

- Views of the high-level synchronization behavior of the program.
- A mechanism for filtering the synchronization view to show the activities of specific eventcounts in the program.
- A process-specific view of the CPU utilization.

Figure 1 shows the main Xunify interface. The top most part of the display is identical to the XPVM tool, except that the pull-down menus can be used to configure the Unify instrumentation system and to start Unify programs.

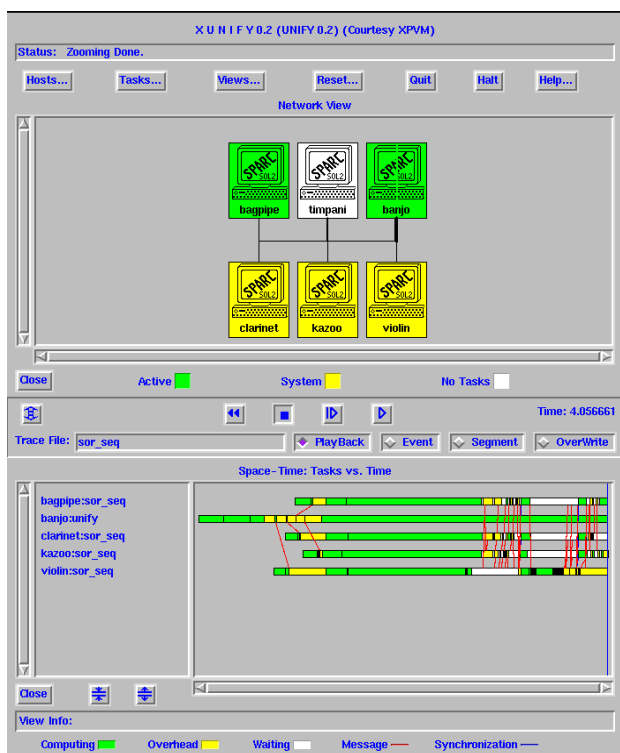


Figure 1. Main Xunify Display

The top window shows a view of the hosts that are in the current configuration connected by a logical bus topology (irrespective of the actual underlying network topology). This is virtually identical to the view in XPVM, however, the animation of the hosts corresponds to activities in the DSM system. The hosts are represented by icons that include the architecture. As the program executes, the colors of the machine icons change as the states of the hosts change (for example, from a computation phase to sending a message). Similarly, as messages pass between hosts, the links between the hosts change color and width to represent the

bandwidth utilized.

The main animation appears in the lower window in Figure 1. This “Space-Time” view represents the execution of the application. The horizontal axis corresponds to time and the vertical axis shows the tasks (represented by horizontal bars) along with the name of the host machines on which the workers run. The process bars change color as the processes change state. Green represents computation, yellow indicates overhead in the Unify library functions and white indicates that the worker is blocked on synchronization. The DSM level communication between processes (e.g., to maintain consistency of a shared segment) is shown with red arrows directed from the sender to the receiver.

While this view provides an overview of the computation, it can also be used to provide detailed information about the Unify events that occur in the application. The horizontal bars actually consist of a series of small rectangles that correspond to Unify events. Clicking on any of these bars or any of the message arrows between bars provides information about the type of event. Total and elapsed time are also available and the view supports zooming in and out to locate performance problems.

4.4 DSM Segment Based Animation

To make Xunify more useful to Unify programmers, it is desirable to correlate activity in the Unify system with abstractions in the user’s application. For example, a programmer may want to view the activities that relate to a given shared data object in the application. To provide this capability, Xunify uses a filter mechanism for the space-time animation to show activity that relates to a specific subset of segments in the application. For example, Figure 2 illustrates the exchange of border segment updates among workers at the beginning of an iteration in the sequential-access segment implementation of a SOR (successive over relaxation) application. This type of view allows the programmer to focus in on performance problems by eliminating from the view information that is not pertinent.

4.5 Utilization Views

Xunify provides two ways to view the utilization of processors. The standard utilization view shows the instantaneous utilization over time as shown in Figure 3. The horizontal axis represents time and the vertical axis shows

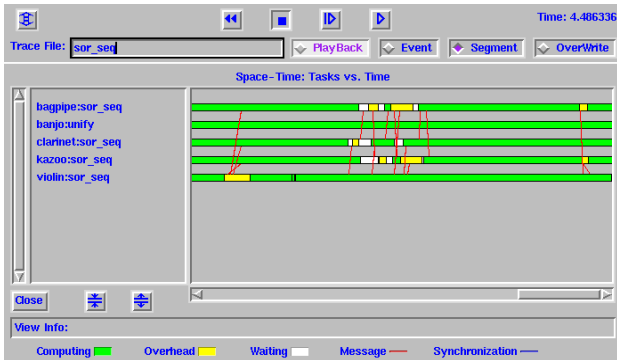


Figure 2. View showing operations on a set of shared segments

the number of tasks operating in computation phases, in overhead operations, or blocked via synchronization. This view gives an measure of the degree of parallelism achieved over time. However, the view does not provide any easy way to identify individual processors that are not utilized efficiently.

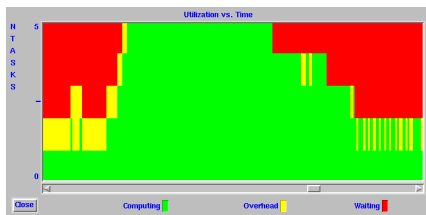


Figure 3. Utilization View

To identify if there are specific processes that are performing poorly, we provide a “per-process” view of utilization. The “per-process” view of utilization is represented by a bar for each task, and each bar is colored to show the percentage of time that each process spent in computation, overhead, and blocked on synchronization. An example of this view is shown in Figure 4. Using this view, the programmer can quickly identify processes that are excessively blocked. This information, combined with the synchronization views provided by Xunify, allow users to optimize programs by restructuring them to avoid excess blocking.

4.6 Synchronization View

Efficient synchronization is essential to the performance of a parallel program. In addition, synchronization errors are a common source of bugs in parallel programs. Xunify

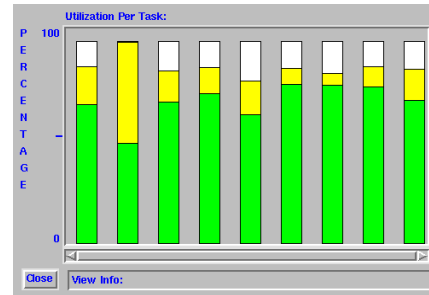


Figure 4. Process Utilization View

includes specific displays to help programmers analyze the synchronization behavior of Unify programs.

Unify’s primary synchronization mechanism is based on scalable eventcounts. Other popular synchronization primitives such as locks and barriers can be implemented on top of eventcounts. Unify provides three calls to manage eventcounts: `Create_Event()`, `Advance()` and `Await()`. Each of these calls is instrumented to record the eventcount id, the eventcount value and the time at which the call occurs.

The synchronization view shows all three calls as rectangles on the space-time view. However, when an `Advance()` causes one or more `Await()` calls to proceed, an arrow is drawn from the `Advance` event to all corresponding `Await` events. `Advance` events that do not cause any `Await` events to continue are only drawn with rectangles. The synchronization view can be superimposed over the standard space-time view or can be displayed as a separate view. In the space-time view, the arrows that represent synchronization are displayed in a different color than the messages for data sharing.

In the same way the user can filter the space-time view to show only messages that correspond to a subset of shared segments, the user can also filter the synchronization view to show only the `Create`, `Advance` and `Await` events that correspond to a specific eventcount or set of eventcounts. This capability allows the user to debug synchronization patterns in their program in terms of the specific event counters. Figure 5 shows an example of some of the synchronization behavior in a matrix multiply benchmark.

Xunify contains several other views from XPVM, including windows that show standard output of each process and one that shows message queues for each process. By leveraging an existing message passing monitoring tool, it was possible to implement Xunify without re-implementing basic trace handling functions and common views. Instead,

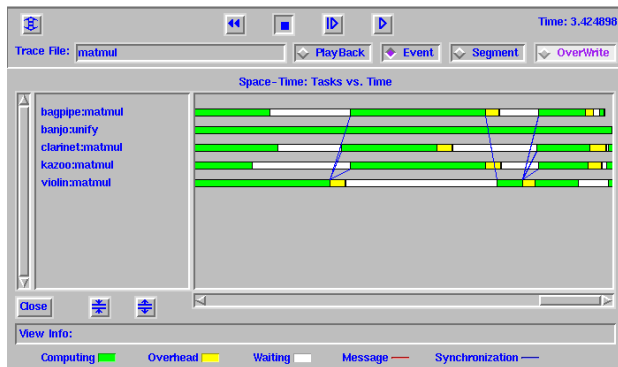


Figure 5. Example Synchronization View

attention was focused on the new capabilities and functionality required to support DSM systems. The resulting tool provides performance information for both Unify application programmers and Unify library developers.

5. Conclusion

This paper introduced the Xunify performance monitoring tool which was designed to work with the Unify DSM operating system. Xunify provides feedback in terms of the application level constructs, the application level send and receive calls that implement the higher level DSM constructs, and the low-level transport protocol details. It displays this information in various graphical views to help application programmers and DSM library developers understand the behavior of applications. The availability of such performance evaluation and debugging tools greatly enhances the usability of DSM systems by providing a means for users and developers to tune applications in an environment where it can be difficult to correlate program level constructs to low level process behavior. The first public release of Xunify is planned for summer 1997 to coincide with the first public release of Unify.

References

- [1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.
- [2] D. Badouel, T. Priol, and L. Renambot. Svmview: a performance tuning tool for dsm-based parallel computers. Technical report, IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France, Nov. 1995.
- [3] A. D. Birrel, R. Levin, R.M. Needham, and M.D. Schroeder. Grapevine: An Exercise in Distributed Computing. *Communications of the ACM*, 25(4), 1982.
- [4] M. J.Zekauskas B. N.Bershad and W. A. Sawdon. The midway distributed shared memory system. In *Proceedings of COMPCON*, 1993.
- [5] J. B. Carter, A. L. Cox, S. Dwarkadas, E. N. Elnozahy, D. B. Johnson, P. Keleher, S. Rodrigues, W. Yu, and W. Zwaenepoel. Network multicomputing using recoverable distributed shared memory. In *Proceedings of COMPCON '93*, pages 519–527, Feb. 1993.
- [6] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical Report Computer Science Department CS-94-230, University of Tennessee, Knoxville, TN, 1994.
- [7] P. Papadopoulos G.A.Geist, James Kohl. Visualization, debugging and performance in pvm. In *Proceedings of Visualization and Debugging Workshop*, Oct. 1994.
- [8] J. Griffioen, R. Yavatkar, and R. Finkel. Unify: A scalable approach to multicomputer design. *IEEE Computer Society Bulletin of the Technical Committee on Operating Systems and Application Environments*, 7(2), 1995.
- [9] J.A. Kohl and G.A.Geist. The pvm 3.4 tracing facility and xpvm 1.1. In *Proceedings of 29th Annual International Conference on System Sciences*, Jan. 1996.
- [10] B. Mohr, D. Brown, and A. Malony. Tau: A portable parallel program analysis environment for pc++. Technical report, University of Oregon Department of Computer and Information Science, Eugene, Oregon 97403.
- [11] R. Rajamony and A. L. Cox. A performance debugger for eliminating excess synchronization in shared-memory parallel programs. Technical report, Rice University Depts. of Electrical Engineering, Computer Engineering and Computer Science, Houston, TX 77251.

- [12] D. Reed and R. Kanodia. Synchronization with event-counts and sequencers. *Communications of the ACM*, 22(2):115–123, Feb. 1979.
- [13] E. Speight and J. K. Bennett. Paraview: Performance debugging of shared-memory programs. Technical report, Rice University Department of Computer and Electrical Engineering, Houston, TX 77251-1892, Mar. 1994.
- [14] J. T. Stasko. The parade environment for visualizing parallel program executions: A progress report. Technical Report GIT-GVU-95-03, Georgia Institute of Technology College of Computing, Atlanta, GA, 1995.
- [15] V.S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4), Dec. 1990.
- [16] B. Topol and J. T. Stasko. Integrating visualization support into distributed computing systems. Technical Report GIT-GVU-94-38, Georgia Institute of Technology College of Computing, Atlanta, GA, Oct. 1994.
- [17] R. Yavatkar, J. Griffioen, and M. Sudan. A Reliable Dissemination Protocol for Interactive Collaborative Applications. In *The Proceedings of the ACM Multimedia '95 Conference*, pages 333–344, Nov. 1995.