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

Although distributed shared memory (DSM) provides a convenient programming model for parallel applications running on a network of workstations, the DSM model presents new challenges for performance evaluation and debugging tools. Although processes communicate and cooperate via shared memory and synchronization primitives, most performance problems arise from the high cost of implementing these abstractions with network messages. Because the DSM abstraction hides the underlying message passing, it is often difficult to identify the reason for poor performance. We have developed an instrumentation system and a performance evaluation tool called Xunify for monitoring and performance debugging of a DSM system. Xunify provides information in terms of the programming-model level, logical-message level, and the transport-protocol level, helping program writers quickly pinpoint the source of inefficiency.

1. Introduction

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

When programmers port 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 on DSM systems 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, and architecture. Second, the programming model is largely implemented in software instead of hardware (e.g., memory sharing and caching) and can add significant overhead. Third, network bandwidth and delays are non-uniform and can be unpredictable. In addition, 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 increase the traffic on an already congested network path or host.

Typical performance analysis tools focus on displaying information related to the programming model. They of-

*This research was supported by the National Science Foundation under Grant CDA-9502645 and the Advanced Research Projects Agency under Grant DAAH04-96-1-0327.

ten 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 for a DSM system. The tool helps the programmer visualize the performance of an application while it is running and can save trace information for 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. The three levels allow application writers to quickly identify inefficiencies due to inappropriate constructs or algorithms at the programming-model level, or alternatively to identify unique bottlenecks that arise from the particular network/runtime environment. 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. Our current implementation is for the Unify DSM system [11], however 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 monitor parallel and distributed systems [21, 1, 8, 19, 20, 9] with the majority of distributed monitering systems targeted towards message passing systems. Each system relies on a monitoring component to gather run-time information, a facility to route this "trace" information to a central location and an analysis and visualization system to display the information to the user. Several tools allow animations of message traffic and processor utilization as well as aggregate statistics on overall program behavior.

For DSM systems, information on the communication and synchronization behavior of the underlying system is still important because it best explains the performance achieved and can be used to find the source of communication-related performance problems. However, in a DSM environment programmers write code using the high-level abstractions of shared memory and consistency. To understand how program constructs correspond to performance problems, the abstractions used by the DSM program must be correlated to the lower-level message passing behavior in the DSM system. The goal of this research to correlate low-level message communication to the high-level constructs of a DSM system.

Tools design for shared memory multiprocessor constructs [17, 10, 15, 12] have been studied and some work has been applied to DSM systems [3]. These systems attempt to provide the user with feedback in terms of the program constructs. However, it is difficult to determine what high-level program behavior causes lower-level communication and synchronization behavior in the DSM system. On a multiprocessor system with uniform processors, uniform bandwidths and latencies, communication costs are often overlooked. However, for the developers of the DSM system with dissimilar machines with varying bandwidths and latencies, information about the relationship between high-level DSM constructs and low-level communication is critical to achieve optimal performance. It is the goal of Xunify to provide a means for programmers and library developers to correlate high-level program behavior to the behavior of the underlying system. Xunify traces the behavior of DSM systems at three distinct levels. Xunify also allows the user to filter the views to display relevant information for a particular performance evaluation or debugging task.

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 implemented Xunify on the Unify multicomputer system, the same techniques can be applied to any DSM system or distributed object system.

Unify [11] 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, uses asynchronous communication to mask latency, and reduces the amount of data being transmitted to achieve the best performance in wide area environments that are typically characterized by high latencies and 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 [18] and MPI [7].

The keys to reducing DSM overhead in Unify are the use of new relaxed-consistency models, support for low-overhead synchronization models, and efficient reliable multicast communication of data (via the TMTP protocol) [22].

The following briefly describes a few of the abstractions and operations supported by the Unify DSM system which are monitored and reported by Xunify.

3.1. Unify Memory Model

The Unify memory model provides a single large virtual address space shared by all machines. Shared segments are dynamically allocated from the single large virtual address space. 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 accessed. 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 access segments* are accessed via <key,value> pairs.

Sequential access and associative access segments can often be supported with weaker spatial consistency guarantees (see section 3.2) that can be implemented more efficiently with fewer 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 segments.

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* that defines when the replicas become consistent and *spatial component* that defines 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* [6, 2] and *entry consistency* [5]) or "automatic" methods where the operating system automatically ensures consistency (e.g. sequential consistency [14]). Unify supports a novel automatic consistency model called *eventual consistency* where the memory becomes consistent after some time lag *t*. This type of consistency is particularly useful for applications that can detect stale data (e.g. [4]).

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 network latencies and serialization that would otherwise occur when writing to a sequential access or associative access segment.

Xunify monitors all consistency related operations that occur on each segment. In the case of user-aided consistency models, this requires 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 and reports cases of false sharing access detected by Unify. False sharing information includes the writes that initiate false sharing and the time at which concurrent false shared writes are merged together.

3.3. Unify Synchronization

To support low-overhead synchronization, Unify supports an *eventcount* [16] abstraction that uses unreliable multicast communication to update synchronization information on all machines. Unify also provides a migrating ticket master that allocates *sequencers* [16]. 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 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 [22] 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 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.

Performance of DSM systems depends on the 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 monitors events that occur within the communication protocol. The protocol events traced are described in section 4.1.3.

4. Xunify

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

The Xunify system implementation consists of three primary components:

- **Instrumentation Code** added to the Unify operating system to collect trace information

- **A Trace Logging Mechanism** used to send trace information from the various Unify processes to the runtime display where it is merged and logged

- **A Graphical Display** that illustrates an application's behavior either during runtime or during a post-mortem "replay" phase.

The following sections describes the three components of the system and describes how the views can be used to evaluate performance.

4.1 Xunify's Multiple Abstraction Levels

There are several factors that can influence the performance of a distributed application running on a network of workstations. Algorithmic inefficiencies such as excessive synchronization, non-optimal partitioning, 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.

Xunify is novel in the fact that it allows users to view or monitor the abstractions provided by different levels of the system and to correlate the abstraction levels. In particular, Xunify monitors the system at three distinct levels giving users a complete picture of the system's performance. The three level are the:

1. Programming-model level

2. Logical-message level, and the

3. Transport-protocol level

In Xunify, the *programming-model level* monitors DSM-level abstractions such as shared memory segment modifications, synchronization access, and consistency checkpoints. The *logical-message level* records information about communication that may result from the invocation of abstractions at the programming model level. Finally, the *transport-protocol level* monitors events related to the underlying transport communication protocol.

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

- **Create_Seg** events mark the point at which a process allocates a shared segment in the global address space.

- **Get_Update** events mark the point at which a process issues a request for the latest version of a particular segment.

- **Put_Update** events mark the point at which a process explicitly transmits segment modifications to every member in the segment's copy set.

- **Need_Update** events mark the point at which a process issues a request to join the copy set of a particular segment.

- **Dont_Need_Update** events mark the point at which a process issues a request to leave the copy set of a particular segment.

Segment identifiers are recorded with each of the programming model events to allow Xunify to associate events with memory segments. In addition, Get_Updates, Put_Updates, Need_Updates and Dont_Need_Updates events include the associated sequence numbers and issuing process id.

Programming model synchronization events are also monitored and reported.

- **Create_Event** events mark the point at which a process allocates a new 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 programming model 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 is to hide communication from the programmer, this transparency may cause programmers to unknowingly select an inefficient implementation.

To help expose communication bottlenecks, Xunify monitors the communication system at two levels: the logical message level and the transport level. At the logical message level, Xunify records logical communication patterns that corresponds to request/reply style logical messages. At the transport-level, Xunify records packet transmission details such as timeouts, retransmissions and acknowledgments that implement the logical message level.

4.1.2. Logical-message Level Monitoring. Monitoring information obtained at the logical message level can be used to identify both algorithmic and resource mapping problems. Consider a program that uses a release consistency DSM model when an entry consistency model would be more efficient. If the users sees messages multicast sent to many machines when only one machine needs the data, the user may decide it is an algorithmic problem and switch to an entry consistency model. Mapping problems can also be identified. For example, viewing the logical messages needed to implement a Put_Updates() in a release consistency model can point out which processes are sharing a segment and should be located on the same network.

There are 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 continue 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. Protocol Level Monitoring. Transport protocol monitoring is useful to determine the characteristics of the communication paths used by an application. Although the typical programmer is not interested in low-level protocol details, summary information of performance at this level can often be invaluable. For example, if a particular network link is highly congested, reports of high packet delays or packet loss rates over that link can help the programmer restructure the application.

In general, programmers do not (and should not) understand the details of the transport layer protocols. However, a visual illustration of the transport level protocol can offer valuable insight into a performance problem, even if the programmer knows relatively little about the protocol. The transport layer monitors and summarizes transport protocol statistics to help point out bottlenecks. 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 and Percent of 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 and Percent of 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, as well as other problems. As a result a programmer might rearrange the assignment of processes to machines, repartition the application, use smaller segment sizes, etc.

To help the advanced programmer, the transport level also records and displays the following events.

- **Frag_Send** and **Frag_Recv** events mark the point at which a process sends and receives a message fragment, respectively.

- **ACK_Send** and **ACK_Recv** events mark the point at which a process sends and receives a cumulative-selective ACK, respectively. **NACK_Send** and **NACK_Recv** events mark the point at which a process sends and receives a 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 re-send attempts issued on the fragment.

4.2 Transmitting Trace Information

It is important that an efficient mechanism be used to send trace data from the DSM processes to the performance evaluation tool. If transmission of 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 we want to show live views of the application as it executes.

For the purposes of our implementation, we decided to build on the functionality already provided by the Parallel Virtual Machine (PVM) [18]. PVM supports real-time tracing of distributed processes, post mortem visualization of trace data and portability. Consequently, we decided to start with PVM and modify it to transmit and display the information we gather from the three levels of abstraction in Unify. Unify only uses the trace facilities provided by PVM and does not use the sharing or synchronization mechanisms provided by 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 a modified version of PVM's evaluation tool called XPVM [8, 13]. 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 trace data handling in PVM to reduce the trace overhead incurred by a Unify application.

To minimize the overhead of sending 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 send trace information when the Unify process is blocked due to synchronization, and (3) added the capability to buffer and periodically transmit trace data. 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 sent. 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 send trace data can be incurred while the process is already blocked. This effectively removes the overhead of transmitting trace data. Finally, to maintain the real-time update capability of displays, the timer feature can force buffered messages to be sent after a predefined time limit expires, bounding 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 performance evaluation tool that is useful for both the development of Unify programs and for the analysis of the Unify libraries. The user interface is based on the XPVM GUI modified to display DSM specific information. Like XPVM, Xunify is capable of displaying live animations of running Unify programs or displaying post-mortem information collected in a trace log file. The interface also provides the capability to select machines and control the execution of Unify applications.

The views in Xunify are specifically designed to allow the user to correlate high-level DSM abstractions with the communication abstraction layers; in particular, the logical message level and protocol packet level. The visual interface allows users to view program activity at individual levels of abstraction or to view combinations of the levels simultaneously. Typically users begin by examining performance at the programming model abstraction level (DSM level). If a potential performance problem is identified, the logical messages used to implement the higher level events can be displayed to reveal the logical communication patterns. In many cases, this is sufficient to identify inefficient sharing patterns or synchronization problems. If the patterns and logical message level appear to be correct, it may be that the performance problem is caused by bottlenecks in the network such as low-bandwidth links, high latencies, or high packet loss rates. In this case, Xunify allows the user to view protocol packet level summary information or even specific packet level events to identify such problems. For both programmers and the DSM library developers, the ability to view these lower-level abstractions are an enormous benefit. Even programmers that are not knowledgeable about message passing or network protocols can gain insight from the graphical display of these layers.

In addition to the providing the ability to view and correlate the multiple abstraction levels, we also added the capability to filter the views at all levels of abstraction to show only those activities corresponding to specific DSM segments or synchronization primitives (event-counters). This allows the user to focus on a particular component of the performance (e.g., to only view events related to a specific shared segment in their program data or important synchronization event in their algorithm). Finally, we added a new utilization view that shows the utilization achieved by each process.

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 have been modified to allow users to start Unify programs and specify the information to be traced.

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 represent the architecture. As the program executes, the colors of the machine icons change corresponding to state changes of the DSM processes on those machines (for example, from a computation phase to a communication phase). Similarly, as messages pass between hosts, the links between the hosts change color and width to represent the bandwidth utilized.

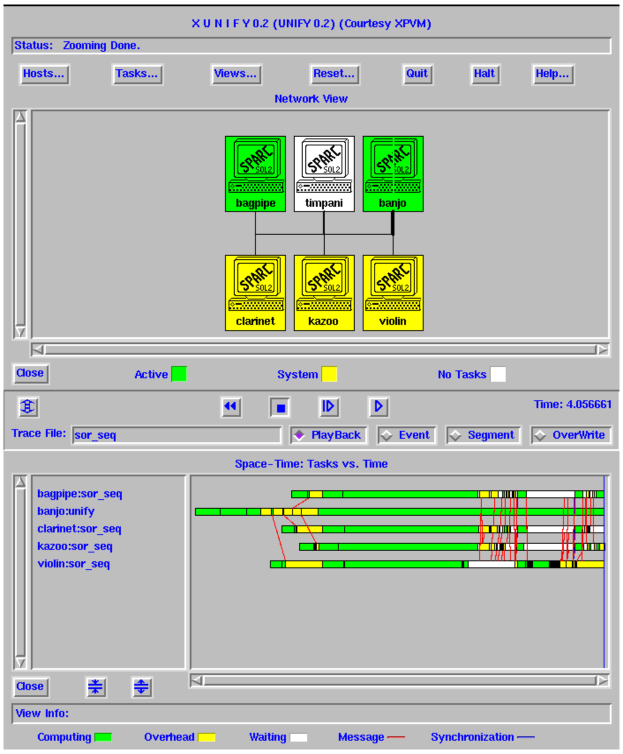

Figure 1. Main Xunify Display

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). The process bars also 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 programming model 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 obtain 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 can "zoom" 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.

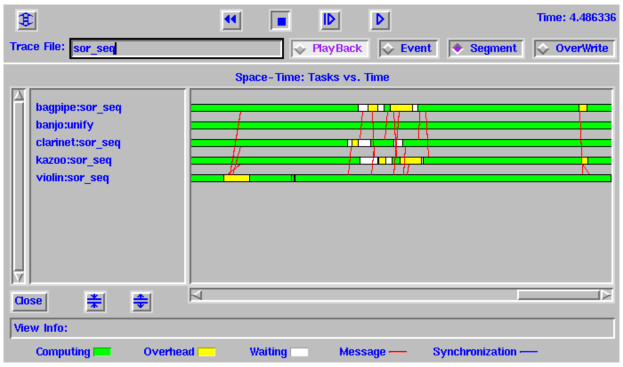

Figure 2. View showing operations on a set of shared segments

4.5 Utilization Views

Xunify provides two ways to view the utilization of processors. The standard utilization view from XPVM shows the instantaneous utilization over time as shown in Figure 3. The horizontal axis represents time and the the vertical axis shows the number of tasks operating in computation phases, in overhead operations, or blocked via synchronization. This view gives a measure of the degree of parallelism achieved over time, however, it is difficult to identify individual processors that are not utilized efficiently.

A "per-process" aggregate view of utilization makes it easy to identify specific processes that are performing poorly. The Process Utilization View consists of a bar for each task with each bar 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 blocked excessively. This information, combined with the synchronization views, allow users to optimize programs by restructuring them to avoid blocking.

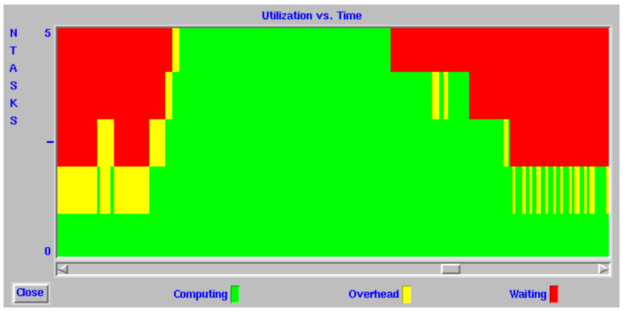

Figure 3. Instantaneous Utilization View

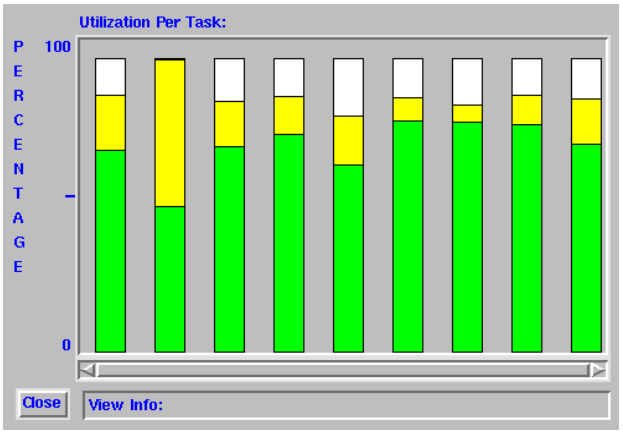

Figure 4. Process Utilization View

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 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 with 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.

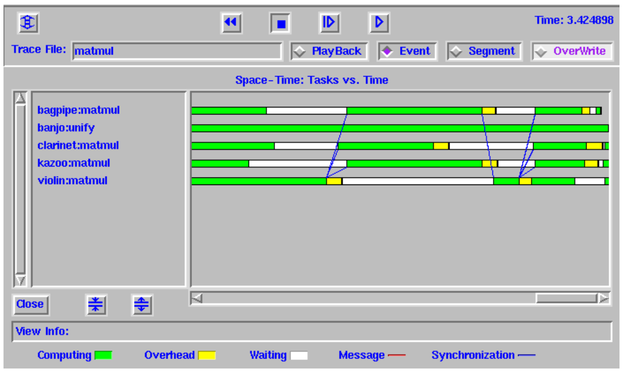

Figure 5. Example Synchronization View

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, attention was focused on the new capabilities and functionality required to support a DSM system. The resulting tool provides performance information useful 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 programming model level constructs, the logical send and receive calls that implement the programming model level 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.

References

[1] V. S. Adve, J. M. Crummey, M. Anderson, K. Kennedy, J. C. Wang, and D. A. Reed. Integrated compilation and performance analysis enviroment for parallel programs. In *Proceedings of Supercomputing '95*, December 1995.

[2] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[3] Didier Badouel, Thierry Priol, and Luc Renambot. Svmview: a performance tuning tool for dsm-based parallel computers. Technical report, IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France, November 1995.

[4] 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.

[5] Matthew J.Zekauskas Brian N.Bershad and Wayne A. Sawdon. The midway distributed shared memory system. In *Proceddings of COMPCON*, 1993.

[6] John B. Carter, Alan L. Cox, Sandhya Dwarkadas, Elmootazbellah N. Elnozahy, David B. Johnson, Pete Keleher, Steven Rodrigues, Weimin Yu, and Willy Zwaenepoel. Network multicomputing using recoverable distributed shared memory. In *Proceddings of COMPCON '93*, pages 519–527, February 1993.

[7] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical Report Computer Science Department CS-94-230, University of Tennessee, Knoxville, TN, 1994.

[8] Philip Papadopoulus G.A.Geist, James Kohl. Visualization, debugging and performance in pvm. In *Proceedings of Visualization and Debugging Workshop*, October 1994.

[9] James E.Mankovich Gary J.Nutt, Adam J.Griff and Jeffrey D. McWhirter. Extensible parallel program performance visualization. Technical Report CB 430, Department of Computer Science, University of Colorado, Boulder, CO.

[10] Aaron Golberg and John Hennessy. Mtool: An integrated system for performance debugging shared memory multiprocessor applications. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):28–40, January 1993.

[11] James Griffioen, Rajendra Yavatkar, and Raphael 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.

[12] James A.Kohl Jack Dongarra, Orlie Brewer and Samuel Fineburg. A tool to aid in the design, implementation and understanding of matrix algorithms for parallel programs. *Journal of Parallel and Distributed Computing*, 9:185–202, 1990.

[13] 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*, January 1996.

[14] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[15] Anoop Gupta Margaret Martonosi and Thomas Anderson. Memspy: Analyzing memory system bottlenecks in programs. pages 1–12, June 1992.

[16] David Reed and Rajendra Kanodia. Synchronization with eventcounts and sequencers. *Communications of the ACM*, 22(2):115–123, February 1979.

[17] Evan Speight and John K. Bennett. Paraview: Performance debugging of shared-memory programs. Technical report, Rice University Department of Computer and Electrical Engineering, Houston, TX 77251-1892, March 1994.

[18] V.S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4), December 1990.

[19] Micheal T.Heath and Jennifer A.Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5), 1991.

[20] John T.Stasko and Eileen Kraemer. A methodology for building application-specific visualization of parallel programs. *Journal of Parallel and Distributed Computing 18(2)*, pages 258–264, June 1993.

[21] Jerry Yan. Performance Tuning with AIMS–An Automated Instrumentation and Monitoring System for Multicomputers. In *proceedings of the 27th HICS, Wailea, Hawaii*, January 1994.

[22] 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, November 1995.