The following paper was originally published in the
Proceedings of the USENIX Windows NT Workshop
Seattle, Washington, August 1997

Brazos: A Third Generation DSM System

Evan Speight and John K. Bennett
Department of Electrical and Computer Engineering
Rice University
Houston, TX

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

Brazos: A Third Generation DSM System[‡]

Evan Speight and John K. Bennett

Department of Electrical and Computer Engineering
Rice University
Houston, TX 77005
{espeight,jkb}@rice.edu

Abstract

Brazos is a third generation distributed shared memory (DSM) system designed for x86 machines running Microsoft Windows NT 4.0. Brazos is unique among existing systems in its use of selective multicast, a software-only implementation of scope consistency, and several adaptive runtime performance tuning mechanisms. The Brazos runtime system is multithreaded, allowing the overlap of computation with the long communication latencies typically associated with software DSM systems. Brazos also supports multithreaded user-code execution, allowing programs to take advantage of the local tightly-coupled shared memory available on multiprocessor PC servers, while transparently interacting with remote "virtual" shared memory. Brazos currently runs on a cluster of Compaq Proliant 1500 multiprocessor servers connected by a 100 Mbps FastEthernet. This paper describes the Brazos design and implementation, and compares its performance running five scientific applications to the performance of Solaris and Windows NT implementations of the TreadMarks DSM system running on the same hardware.

1. Introduction

Recent improvements in commodity general-purpose networks and processors have made networks of multiprocessor PC workstations an inexpensive alternative to large bus-based distributed multiprocessor systems. However, applications for such distributed systems are difficult to develop due to the need to explicitly send and receive data between machines. By providing an abstraction of globally shared memory on top of the physically distributed memories present on networked workstations, it is possible to combine the programming advantages of shared memory and the cost advantages of distributed memory. These distributed shared memory (DSM) runtime systems transparently intercept user accesses to remote memory and translate these accesses into messages appropriate to the underlying communication media. The programmer is thus given the illusion of a large global address space encompassing all available memory, eliminating the task of explicitly moving data between processes located on separate machines.

Both hardware DSM systems (e.g., Alewife [19], DASH [17], FLASH [15]) and software DSM systems (e.g., Ivy [18], Munin [6], TreadMarks [14]) have been implemented. Because of the traditionally higher performance achievable on engineering workstations relative to personal computers (PCs), the majority of existing DSM systems are Unix-based. Recent increases in PC performance, the exceptionally low cost of PCs relative to that of engineering workstations, and the introduction of advanced PC operating systems combine to make networks of PCs an attractive alternative for large scientific computations.

Software DSM systems use page-based memory protection hardware and the low-level message passing facilities of the host operating system to implement the necessary shared memory abstractions. The large size of the unit of sharing (a page) and the high latency associated with accessing remote memory combine to challenge the performance potential of software DSM systems[18]. A variety of techniques have been developed over the last decade to address this challenge [6, 10, 14]. DSM systems built using these techniques can be roughly grouped into three "generations": early systems like Ivy [18] that employ a sequentially consistent memory model [16] in a single-processor

[‡] This research was supported in part by substantial equipment donations from Compaq Computer Corporation and Schlumberger Company, and by the Texas Advanced Technology Program under Grant No. 003604-016. John Bennett was on sabbatical leave at the University of Washington while a portion of this work was conducted.

workstation environment, second generation systems like Munin [5] and TreadMarks [13] that employ a relaxed memory consistency model on similar hardware, and third generation systems that utilize relaxed consistency models and multithreading on a network of multiprocessor computers.

This paper describes the design and preliminary performance of Brazos, a third generation DSM system that executes on x86 multiprocessor workstations running Windows NT 4.0. Brazos is unique among existing DSM systems in its use of selective multicast, software scope consistency, and adaptive runtime performance tuning. Brazos uses selective multicast to reduce the number of consistency-related messages, and to efficiently implement its version of scope consistency [11]. Brazos uses a software-only implementation of scope consistency to reduce the number of consistency-related messages, and to reduce the effects of false sharing[2]. Finally, Brazos incorporates adaptive runtime mechanisms that ameliorate the adverse effects of multicast by dynamically reducing the size of pages' "copysets"[6], as well as an early update mechanism that improves overall network throughput. Brazos provides thread, synchronization, and data sharing facilities like those found in other shared memory parallel programming systems.

Brazos has been designed to take advantage of several Windows NT features, including true pre-emptive multithreading; support for the TCP/IP transport protocol, and in particular, multicast support; and OS support for symmetric multiprocessing (SMP) machines. Unlike most previous DSM systems, the Brazos runtime system is itself multithreaded. This allows computation to be overlapped with the long communication latencies typically associated with software DSM systems. Brazos also supports multithreaded user-code execution, allowing programs to take advantage of the local tightly-coupled shared memory available on SMP PC servers, while transparently interacting with remote "virtual" shared memory physically resident on other clusters. Brazos consists of four parts: a user-level library of parallel programming primitives, a service that allows the remote execution of DSM processes similar to the Unix **rexec** service, a memory management device driver that allows two virtual addresses to be mapped to the same physical address, and a Windows-based graphical user interface. Brazos currently runs on a cluster of Compaq Proliant 1500 multiprocessors connected by FastEthernet. In addition to describing the design and implementation of Brazos, this paper compares the performance of Brazos to the performance of Solaris and Windows NT implementations of the TreadMarks DSM system[14] running on the same hardware platform. For the applications studied, Brazos offers superior performance to both implementations of TreadMarks.

The rest of this paper is organized as follows. Section 2 describes some of the important differences between Unix and Windows NT as they relate to the development of software DSM systems. Section 3 discusses the design issues related to the Brazos system, and motivates many of the implementation decisions. Section 4 presents some preliminary performance results of Brazos relative to the TreadMarks DSM system, which represents the existing state-of-the-art in software-only DSM systems. Section 5 provides a brief discussion of related work. Section 6 concludes the paper and describes plans for future development.

2. Unix/Windows NT Differences

Windows NT differs substantially from Unix. The major differences that directly affect DSM implementation and performance include:

- Windows NT has native multithreading support build into the OS.
- BSD-style signals are not available.
- Exception handling is implemented through structured exception handling.
- Windows NT implements TCP/IP through the WinSock user-level library.

2.1. Multithreading

One of the significant features of the Windows NT operating system is the native support for multithreaded operation. Windows NT provides support for multiple lightweight threads executing within the same process address space. The Win32 API provides a rich set of calls to address threading issues, including support for thread priority manipulation, synchronization, thread context manipulation, and thread suspension and resumption. Standard Unix does not provide for lightweight threads, although there are several lightweight thread packages, such as Pthreads, that are available to run on top of Unix.

2.2. Signals

Unix makes use of *signals* to inform the operating system of events that must be handled. Signals that are used extensively in software DSM systems include SIGIO, which indicates that an I/O operation

is possible on a file descriptor (a *socket*), and SIGALARM, used as a timing device. The SIGIO signal is generally used to notify a Unix DSM system that an asynchronous message from another process is waiting to be received. An asynchronous message on any socket causes the function associated with the SIGIO signal to be invoked immediately, unless the user has explicitly blocked the delivery of the signal. The **select()** function must then be used to determine which socket has available data.

Windows NT does not support this kind of upcall mechanism. Instead, when a socket is made asynchronous in Windows NT, the function that will handle the asynchronous message is also specified. Thus, the function to be called is tied to the *socket* instance instead of a signal, allowing asynchronous messages to invoke different handler routines depending on the socket on which they are received. Additionally, the receipt of an asynchronous message does not automatically halt other threads (i.e., user-code threads). This allows independent threads to process incoming messages concurrently, while still allowing computation to proceed within scheduling guidelines.

2.3. Structured Exception Handling

Coherence is maintained in a page-based DSM system by setting page protection attributes to indicate the access permissions for a specific shared page. Processes "invalidate" pages in memory by altering the page protection such that any attempt to read or write to the page causes an access violation fault. For example, the Munin [5] and TreadMarks [13] DSM systems install interrupt handlers that specify a function to be called when a page-access violation occurs. When an access to an invalid page occurs, the DSM system enters the interrupt handler, and messages are sent to other processes to acquire the data needed to bring the page up-to-date in order to allow the user program to continue.

Windows NT accomplishes exception handling through a mechanism known as *structured exception handling* (SEH). SEH allows a greater amount of control over how exceptions are handled. Instead of installing a separate handler for each exception, as is done in Unix, Windows NT implements SEH through the **try-except** block. The **try-except** block is similar in flavor to the **throw-catch** block in C++, but it is used to trap software or hardware generated exceptions. The routine identified by the **__except()** keyword is called whenever any exception is raised within the **try-except** block, and this routine decides whether to handle the exception, pass the exception up to the operating system for handling, or continue execution without addressing the exception. In this way, exceptions that occur in different parts of code can be easily handled in different ways.

The code fragment below shows the use of the **try-except** block in the Brazos DSM system. The function **UserMain()** is the entry-point to the user code, and is called by each user thread. By placing the **try-except** block around this function, Brazos catches any page-access violations caused by threads accessing invalid pages. The function **AccessViolationHandler()** is the Brazos equivalent of the interrupt handler function installed in Unix-based DSM systems.

```
__try {
     UserMain(GlobalId,LocalId);
__except(AccessViolationHandler());
```

Table 1 gives the measured performance for two virtual memory operations (running on the same hardware) for both Windows NT and Solaris, a Unix System V derivative available from Sun Microsystems. The two systems are comparable in speed for setting the protection attributes of a page. However, Windows NT is more than twice as fast as Solaris handling an access violation, due to the lower overhead of the **try-except** block relative to the interrupt handling capabilities of Solaris. In practice, only those applications that exhibit a large number of access violation faults will substantially benefit from this difference.

OS	**Page Protect**	**try-except** *or* **Segv Handler**
Win NT 4.0	7.0 µsec	20 µsec
Solaris 2.5.1	6.57 µsec	47 µsec

Table 1. System Call Timings

2.4. WinSock

Processes in Brazos use functions in the WinSock Programming API to communicate with other processes in the system. All WinSock API calls are implemented in the WinSock library. With BSD sockets, some calls are direct system calls into the operating system, while other calls are made to functions in a static library that is linked at compile-time. Consequently, there is more overhead associated with many of the WinSock calls than with

their BSD counterparts, which can result in higher per-message overheads and lower overall throughput.

Figure 1 shows the average network throughput achievable between two machines connected by 100 Mbps FastEthernet. The graph shows throughput for Windows NT/WinSock as well as Solaris. These tests were conducted on the same hardware to rule out any variation due to architectural differences. The test conducted was a simple request-reply sequence that typifies the type of communication pattern seen in DSM systems. The client sends a 20 byte request to the server, which responds with a message of a length specified in the request. This loop is repeated 20,000 times, and the average results for different response sizes are presented here. The outlined area shows the range of response sizes that would be expected in most DSM applications (from a few bytes up to a single 4 Kbyte page).

Figure 1 shows that for all response sizes in the typical DSM operating range, Solaris achieves a higher throughput than Windows NT, although neither system attains even half of the possible throughput until the response size exceeds 4 Kbytes. For responses of 16 Kbytes and 32 Kbytes, WinSock is able to stream data out more quickly, after paying the cost of initiating the message. Additionally, UDP messages can be up to 64 Kbytes in length under WinSock, but are limited to 32 Kbytes under Solaris.

3. Design of Brazos

Brazos has been designed to take advantage of the features available to the Windows NT and WinSock programmer. In particular, Brazos makes use of multithreading to overlap communication with computation; multicast to reduce the number of messages sent across the network; scope consistency [11] to reduce false sharing; and adaptive runtime support to implement an early update protocol and a page migration facility.

3.1. Design Overview

The Brazos user-level library is statically linked with user applications at compile-time and provides the interface between user code and DSM code. The most important part of the Brazos library is the interface for capturing accesses to invalid data and initiating messages with other processes in the system. This is accomplished by placing a **try-except** pair around the user code (see Section 2.3). The Brazos API also includes synchronization primitives in the form of locks and barriers, which can be used to provide synchronization both between threads in different processes as well as between threads in the same process. Routines for error reporting, statistics gathering, and data output are also provided in the Brazos API.

Windows NT does not ship with a method of starting a process on a remote machine similar to the Unix **rexecd** daemon. Therefore, Brazos includes a *service* that must be installed on each machine that will run Brazos code. This service listens for incoming DSM

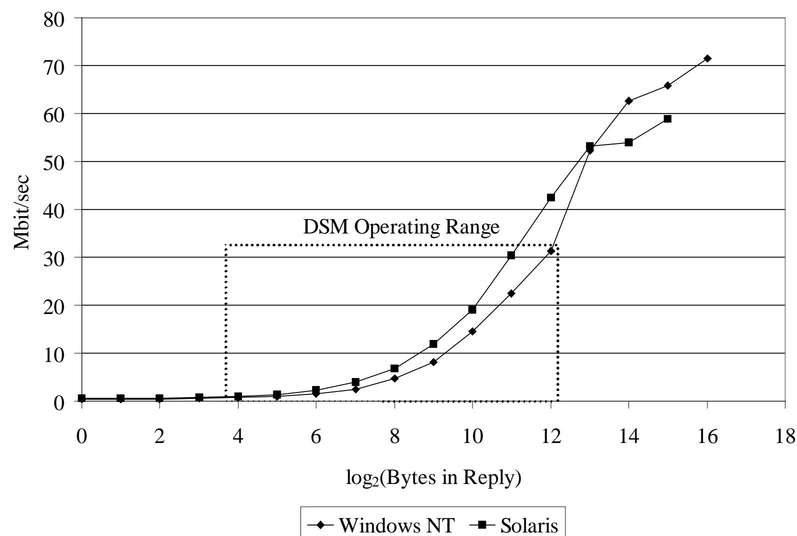

Figure 1. Average Network Throughput

session requests, authenticates encrypted passwords that users must have to run a Brazos session, starts and manages current DSM sessions running on the local machine, and provides a mechanism for the owner to remotely kill a runaway DSM application.

In a multithreaded DSM system such as Brazos it is necessary to have a mechanism to allow the DSM system threads to update a page of shared memory without changing a page's protection. If a DSM system thread changes a page's protection in order to bring it up to date, there is a chance that a user thread will read part of the page before the updating is complete. There are two solutions to this problem. The first is to suspend all user threads when the system needs to atomically change the contents of a shared page. This method carries a high cost, especially when the number of user threads per process is large. The second method is to provide a mechanism to map two virtual addresses to the same physical page in memory, as Unix provides with the **mmap()** call. This allows there to be two different sets of protections associated with a single physical page. Windows NT does not provide a call similar to **mmap()**, but we altered the **mapmem** device driver in the Windows NT Device Developers Kit to provide this functionality.

Brazos includes a graphical user interface that provides mechanisms for specifying the number of user threads to start in each process, the mapping of threads to processors, the priority at which the DSM application should run, extensive statistics gathering and presentation mechanisms, and a service that can probe the network for hosts willing to accept a DSM session. Brazos can also be run in console-application mode when a Windows desktop is not available (i.e., during a telnet login session).

3.2. Multithreading

The Brazos DSM system utilizes multithreading at both the user level and DSM system level. Multiple user-level threads allow applications to take advantage of SMP servers by using all available processors for computation. Coherence is maintained between threads on the same machine through the available hardware coherence mechanisms. In addition to user-level multithreading, the Brazos runtime system itself is multithreaded. There are two main system threads in Brazos. One thread is responsible for quickly responding to asynchronous requests for data from other processes and runs at the highest possible priority. The second thread handles replies to requests previously sent by the process. As a practical matter, it is necessary for any DSM system written for Windows NT to be multithreaded to some degree, because it is difficult to interrupt a thread that is executing computationally intensive user code to cause it to respond to an asynchronous request for data. This is due to the lack of the signal-style upcall mechanism described in Section 2.2. This multithreaded aspect of Brazos allows a greater amount of computation to communication overlap, especially if there are more processors located on a given server than the number of user threads assigned to it. Finally, the use of a separate thread to handle incoming replies allows Brazos to maintain multiple simultaneous outstanding network requests, which can significantly improve performance [22].

3.3. Software Scope Consistency

DSM systems must maintain data consistency to ensure that threads do not access stale or out-of-date data that was written by a thread on another machine. Although a detailed discussion of the many consistency models used in shared memory systems is beyond the scope of this paper, we will briefly outline the major consistency protocols in use in modern DSM systems.

Sequential consistency (**SC**) [16] is the most intuitive, but also most restrictive, consistency model. Sequential consistency requires that all data accesses be consistent with a global ordering that does not violate program order. The simplest method of implementing sequential consistency requires threads to globally invalidate a page after every write to a shared variable on that page. This guarantees that no two threads will access out-of-date data, but can result in unacceptably high communication overhead in software DSM systems [5].

To reduce the amount of communication, consistency constraints can be relaxed by guaranteeing that shared data is only up-to-date after specific synchronization operations have been performed. For example, *release consistency* (**RC**) [8] guarantees that data is current only after a thread has performed a release operation. Simply put, a release operation can be thought of as the releasing of a lock variable or the departure from a barrier. Between release operations, it is the user's responsibility to ensure that no two threads perform competing accesses to the same storage space in memory (competing accesses are multiple accesses, one of which is a write). By relaxing the consistency model in this way, software DSM systems can buffer writes until they are required to be globally performed by the semantics of the consistency protocol. This results in substantial performance benefits because of the large reduction in communication overhead [5]. *Lazy release consistency* (**LRC**) [13] further delays the

propagation of invalidations until a synchronization variable is next acquired.

Scope consistency (**ScC**) [11], introduced as an enhancement to the SHRIMP AURC system [3], is a relaxed consistency model that seeks to reduce the *false sharing* present in page-based DSM systems. False sharing occurs when two or more threads modify different parts of the same page of data, but do not actually share the same data element. This leads to unnecessary network traffic, and can be a significant performance problem for DSM systems due to the large granularity of sharing. Scope consistency divides the execution of a program into *global* and *local scopes,* and only data modified within a single scope is guaranteed to be coherent at the end of that scope. Global scope delimiters include global synchronization events such as barriers. After a global scope is closed (completed), all shared data in the program is guaranteed to be coherent. A lock acquire-release operation is an example of a *local scope*. When a thread acquires a lock, it enters a new local scope. All changes made until the closing of the local scope (i.e., the lock release) are guaranteed to be visible to the next acquirer of the lock, but not changes made before the lock acquisition. This is in direct contrast to **RC**, which guarantees that all shared data is coherent after a release, regardless of when the shared write occurred or what type of release was performed.

The Brazos implementation of scope consistency (**SScS**) differs from that described in [11] in two ways: **SScS** is a software-only implementation of scope consistency that requires no additional hardware support, and Brazos uses **SScS** in conjunction with a distributed page management protocol similar to TreadMarks[14] as opposed to a home-based system such as Munin [5] and AURC [3]. In distributed page-based protocols, each process maintains dirty portions of each shared page of data, requiring processes to communicate with all other processes that have a modified portion in order to bring an invalid page up to date. In a home-based system, however, processes flush changes to a designated "home process", which always has the most up to date copy of a page. Other processes simply send a single message to the home process to re-acquire an invalid page. Brazos uses a distributed page management algorithm because for systems without DSM hardware support, distributed page management protocols outperform home-based page management protocols [13].

In order to be more precise about the specific conditions required to implement scope consistency for local scopes, the following conditions delineate the differences between release consistency and scope consistency using release-consistency nomenclature[‡]. Conditions associated with scope consistency only are shown in **[boldface]**. Conditions associated with both release consistency and scope consistency are shown in normal type. The use of the term "performed with respect to" in these conditions is consistent with that of [11].

1. Before an ordinary load or store is allowed to perform with respect to any other processor, all previous acquires must be performed.
2. Before a release is allowed to perform with respect to any other processor, all previous ordinary loads and stores **[after the last acquire to the same location as the release]** must be performed **[with respect to that processor]**.
3. Synchronization accesses (acquires and releases) must be sequentially consistent with one another.

Figure 2 demonstrates these concepts. In Figure 2,

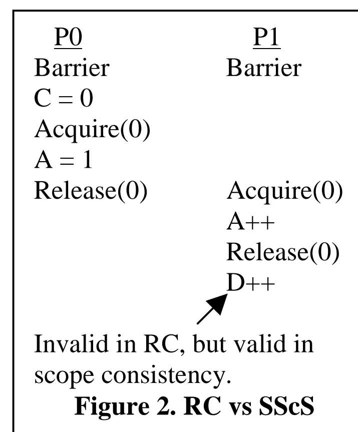

Figure 2. RC vs SScS

assume that variable **A** is on one page of shared memory, and variables **C** and **D** are on another. In a coherence protocol such as release consistency, variable **D** will be invalid in process **P1** after the **Release(0)** performed by process **P0**. This is because variable **C** was written to by process **P0** before the release, and variables **C** and **D** are on the same shared page. Under scope consistency, variable **D** will not be invalidated because the write to **C** by process **P0** occurred outside of the local scope delimited by the **Acquire(0)-Release(0)** pair. The effect of the write to the page containing variables **C** and **D** will not be propagated to process **P1** until the end of the current

---

[‡] In order to use terminology consistent with previous work in the area of relaxed consistency, "opening" and "closing" a *local* scope, as defined in [11], will be considered to be equivalent to an "acquire" and "release" to the same location. "Opening" and "closing" a *global* scope, as defined in [11], will be considered to be equivalent to consecutive barrier events.

global scope. Had the programmer wanted to ensure that the correct value of **C** would be available to process **P1** after the critical section performed by process **P0**, either the write of **C** should be moved into the critical section, or a global scope must be used after the write to variable **C**.

In Brazos, **SScS** provides two main benefits. First, in the code fragment just discussed, the new value of **A** written by process **P0** is sent along with the lock grant to process **P1**, thereby eliminating a message that would result from the fault of **P1** when trying to increment **A** under **RC**. Secondly, the page containing **C** and **D** is falsely shared in Figure 2. Therefore, process **P1** will find the page invalid under **RC**, even though **P0** did not actually modify **D**. **SScS** removes the effects of this false sharing by not invalidating the page containing **C** and **D** when the lock ownership is transferred.

In some situations, programmers may be faced with situations where it is not easy to switch from **RC** semantics to **SScS** semantics. For such instances, Brazos provides both a release consistent lock release primitive, which will flush all invalidation messages before allowing the release to complete; and a lazy release consistent acquire primitive that flushes updates to modified pages to all processes at a lock acquire. The flexibility provided by the inclusion of these two extra lock primitives makes porting existing parallel programs to Brazos easier.

3.4. Multicast Communication

In order to reduce the number of consistency-related messages, and to efficiently implement scope consistency in software, Brazos makes use of the multicast primitives provided by the WinSock 2.0 library [23]. In a time-multiplexed network environment such as Ethernet, sending a multicast message is no more expensive than sending a point-to-point message, and large reductions in both the number of messages sent and the number of bytes transferred to maintain coherence can be achieved by specifying multiple recipients for each message. Brazos uses multicast to reduce consistency-related communication traffic during *global* synchronization as follows.

When a process arrives at a global synchronization point (i.e., a barrier), the process sends a message to a statically assigned barrier manager indicating that the process has arrived at the barrier. Included in this message is a list of pages that the process has written to since the last synchronization point (*dirty* pages).

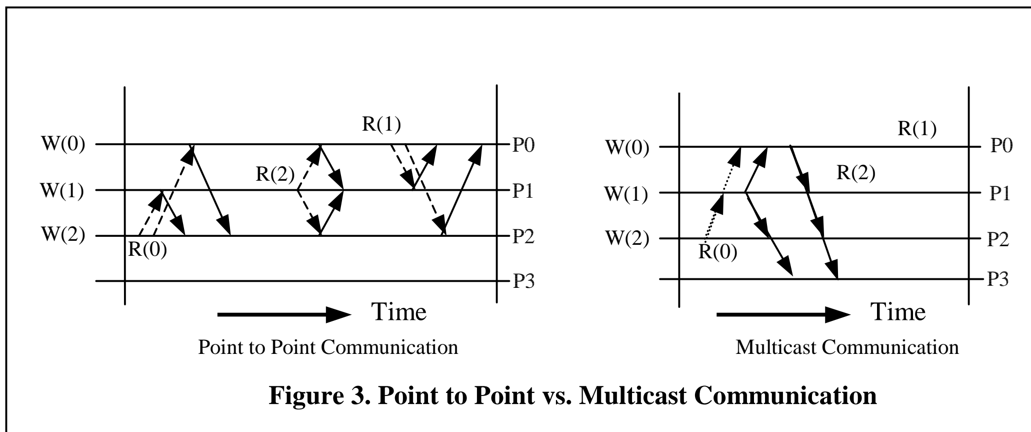

Figure 3. Point to Point vs. Multicast Communication

When the barrier manager receives notification from all processes, the manager collates information regarding dirty pages, and sends a message to each process indicating the pages that should be invalidated, who has dirty pieces of the page, and that the process is free to proceed from the barrier.

After being released from the barrier, threads will begin faulting on pages that were invalidated by the barrier manager. Since the manager also communicated which processes have copies of all dirty pages, a single multicast message may be sent to the subset of all processes that have dirty pieces of the page. These processes, in turn, multicast their response to not only the requesting process, but to all processes in the current copyset for that page. The responses come in the form of *diffs*, which are runlength encodings of the changes made to the page since the last invalidation. Thus, processes that need the *diffs,* but have not yet faulted on the pages, will receive *indirect diffs* for these pages, with the intent of bringing them up-to-date before a page fault resulting from the accessing of the invalid page occurs.

This mechanism is illustrated in Figure 3, which shows the differences in communication patterns between a distributed page-based DSM system that

relies on point-to-point communication and Brazos. In Figure 3, processes **P0**, **P1**, and **P2** each write to a different variable on the same page of shared data sometime before the first barrier, **B1**, as indicated by **W(0)**, **W(1)**, and **W(2)**. After **B1**, the processes each read a value written by another process, as indicated by **R(0)**, **R(1)**, and **R(2)**. The messages required to satisfy each of these read requests are shown with arrows, with request messages using dashed lines and response messages using solid lines. As can be seen, it takes 12 point-to-point messages to completely bring processes **P0**, **P1**, and **P2** up-to-date for the data written before **B1**, but only 3 messages are required for the method employing multicast. However, in the multicast implementation, process **P3** also receives indirect *diffs* for the page, even though these are not used. This potential performance problem is addressed in the next section.

3.5. Adaptive Runtime Support

Adaptive performance tuning mechanisms can have a beneficial effect on performance when used to tailor runtime data management to observed behavior[1, 2]. Brazos employs four adaptive techniques: dynamic copyset reduction, early updates, an adaptive page management protocol, and a performance history mechanism.

3.5.1 Dynamic Copyset Reduction

One disadvantage with multicast is the potential harmful effect of unused indirect *diffs*, as in the case of **P3** in Figure 3. Receiving multicast *diffs* for inactive pages does not increase network traffic. However, it does cause processors to be interrupted frequently to process incoming multicast messages that will not be accessed before the next time that the page is invalidated, detracting from user-code computation time. The dynamic copyset reduction mechanism ameliorates this effect by allowing processes to drop out of the copyset for a particular page, causing them to be excluded from multicast messages providing *diffs* for the page. The decision to drop out of the copyset is made by counting the number of unused multicast *diffs* received for a specific page. When this number reaches a certain threshold, the process will place this page on the "to be dropped" list, and this list of pages is piggybacked on the next barrier arrival message. The process then removes itself from the current copyset. When a thread in the process next faults on the page, the entire page is retrieved from the manager and any outstanding *diffs* from other processes are retrieved and applied immediately. This adaptation is particularly beneficial to performance in situations where two processes actively share a page of data, and neither of these processes is the page manager. Although Brazos follows a distributed page management algorithm (i.e., processes must retrieve modified pieces of a dirty page from several processes, not just one), each shared page is assigned a page manager at the beginning of execution from which the page is retrieved by other processes only on the first access to the page. Because the page manager is always in the current copyset for a shared page, the manager will always receive indirect multicast *diffs* for the page. The adaptive copyset reduction mechanism allows the manager to drop out of the copyset and migrate the manager status to a process actually involved in the sharing. This reduces the number of useless indirect *diffs* that the original page manager receives.

3.5.2 Early Updates

Another form of runtime support provided by Brazos is an *early update* mechanism. Because the majority of the DSM-related network activity in release-consistent DSM systems occurs immediately after synchronization events, network traffic in these systems tends to be bursty. Referring again to Figure 3, assume that **R(0)**, **R(1)**, and **R(2)** all happened immediately after the barrier. This would result in processes being sent indirect multicast *diffs* for pages for which the process currently has outstanding requests, resulting in extraneous messages. In order to reduce this burstiness, processes in Brazos note pages for which they receive indirect *diffs* while they are waiting for a response to a request for *diffs* to the same page. At the next global scope, processes send the page numbers of these early update pages to the barrier manager along with the barrier arrival message. The manager distributes the list of the new early update pages with the barrier release message to all processes, and thereafter processes multicast their changes for all early update pages in a single bulk transfer message before each arrival at a barrier. This eliminates the flurry of network traffic resulting from threads simultaneously faulting on the same page in memory immediately after a synchronization point, since the early update pages will not be invalidated after the barrier. Pages may switch back from the early update protocol to the default multicast invalidation protocol by the dynamic copyset reduction mechanism described above. Specifically, processes count how many updates go unused for each page. When this number reaches a certain threshold, the process drops from the copyset. When the number of processes in the copyset reaches one, the page's protocol is changed back from early update to the multicast invalidate protocol.

3.5.3 Other Techniques

Brazos incorporates two other adaptive techniques. The first of these allows pages to be managed with either a home-based protocol similar to Munin [5], or a distributed page protocol similar to TreadMarks [13]. The Brazos runtime system adaptively alters pages' management protocol based upon observed behavior in order to provide the best management technique for each shared page.

Brazos also incorporates a history mechanism that allows the runtime system to more quickly adapt to programs' behavior. Brazos saves information regarding the performance of the adaptive protocols in a file for each application. These files store information about how well the various adaptive techniques worked for each program variable. This low level of granularity is desirable because the mapping between pages and data may change across program execution, but program variables generally do not. The history mechanism and dynamic page management protocol were not used in obtaining the results presented in Section 4. Details on these techniques can be found in [22].

3.6. Brazos Program Development

Users write Brazos programs using familiar shared-memory programming semantics. Any shared data in the system may be transparently accessed by any thread without regard to where in the system the most current value for that data resides. The Brazos runtime system is responsible for intercepting accesses to stale data and bringing shared pages up-to-date before program execution is allowed to continue.

Programs written for Brazos specify the function **UserMain()** instead of the normal **main()** function as the entry point into user code. User code is linked with the static library **brazos.lib** at compile-time. This library contains the DSM system code for maintaining shared-memory across the network, providing synchronization between threads (both within the same process and between processes) and collecting statistics on the performance of the local DSM process. The resulting console-based executable is started on each machine participating in the DSM run through the use of the Windows NT service described in Section 3.1.

The Brazos DSM programming library provides parallel programming macros based on a superset of the PARMACS macro suite [4]. Locks and barriers are the two forms of synchronization available to the parallel programmer, and the synchronization macros for these may be used without regard to where the synchronizing threads are located (e.g., in the same process, or between processes). Windows NT synchronization primitives are embedded inside the PARMACS synchronization macros to allow for this transparency, which also allows the same executable to be run as a DSM program across servers, or as a strictly hardware-based shared memory program on a single SMP server. All shared data must be dynamically allocated through the PARMACS macro **G_MALLOC** in order to make the DSM subsystem aware of which portions of the address space must be maintained as shared data. All other data is considered to be private to the threads running in each individual process.

4. Performance of Brazos

This section presents preliminary performance results for various configurations of eight processors. Figure 4 presents speedup numbers comparing Brazos to two versions of the TreadMarks DSM system: TMK-SOL is the standard release of TreadMarks 1.0 on Solaris, and TMK-NT uses a version of TreadMarks 1.0 that we have ported to Windows NT. All data were obtained on a network of four Compaq Proliant 1500 servers connected by a 100 Mbps FastEthernet. Each Proliant 1500 has two 200 MHz Pentium Pro processors with 192 Mbytes of main memory.

Application	Input Set
SOR	2048 X 2048 matrix
ILINK	*Amish* input set
Barnes Hut	32,768 bodies
Water	729 molecules, 10 steps
Raytrace	*balls4* input set

Table 2. Application Input Sets

Five applications were studied. SOR is a nearest-neighbor algorithm used to solve differential equations and was taken from the TreadMarks sample application suite. ILINK [9] is a parallel implementation of a genetic linkage program that traces genes through family genealogies. Barnes Hut solves hierarchical n-body problems and was taken from the SPLASH benchmark suite[21]. Water calculates forces and potentials in a system of water molecules, and was also taken from the SPLASH suite. Finally, Raytrace is a graphics rendering application from the SPLASH-2 benchmark suite [24]. Input data for each of these programs is shown in Table 2.

The speedups shown in Figure 4 are relative to the uniprocessor execution time for the same operating system, e.g., the speedups for TMK-SOL are shown relative to the uniprocessor execution time under Solaris, and the Windows NT configurations are shown relative to the uniprocessor execution times under Window NT. The uniprocessor times are not the same for the TMK-NT and TMK-SOL implementations due to compiler differences. For further details, see [22].

SOR achieves a higher speedup under both Windows NT configurations than under Solaris. Because SOR has little communication, the higher achievable network throughput for Solaris shown in Figure 1 does not give the Solaris implementation a significant advantage. The overlap of communication with available computation allows TMK-NT to slightly outperform TMK-SOL. The Brazos implementation further improves performance by making use of the available native hardware shared memory support for threads located on the same machine, reducing the overall network communication by half. SOR's sharing is all pair-wise between at most two threads, therefore the use of multicast does not help the performance of the Brazos implementation.

Examining the performance of ILINK, we see that the TreadMarks implementation under Solaris performs 9% better than the Windows NT implementation. ILINK has a moderate amount of communication, and the increased network throughput of Solaris give TMK-SOL an advantage over TMK-NT for this application. However, the use of the available native hardware shared memory support again gives the Brazos configuration a significant advantage in ILINK. Additionally, the sharing patterns in ILINK favor the use of multicast, and these two factors decrease the overall communication needed by 238% over the course of ILINK's execution. Finally, the early update adaptive mechanism further reduces communication by another 37%, leading to the 54% performance improvement of ILINK under Brazos.

Barnes Hut displays behavior similar to that of ILINK, with Brazos significantly outperforming either TreadMarks implementations. Barnes Hut benefits slightly from multithreading, but the use of multicast reduces communication rates by more than seven-fold through a reduction in the effects of false sharing. Barnes Hut displays a large amount of false sharing, with threads faulting on pages that are write shared, even though individual data elements on the pages are not shared. In Brazos, processes receive indirect *diffs* for falsely shared data before the access violation occurs, eliminating the detrimental effects of the false sharing, reducing communication, and leading to a near doubling in performance for Barnes Hut.

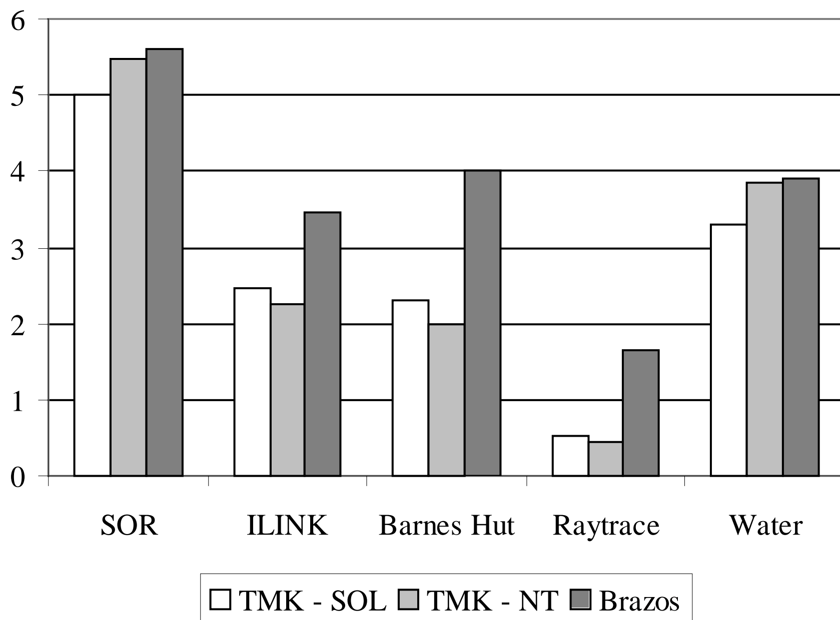

Figure 4. Performance of Brazos vs TreadMarks on 8 Processors

Raytrace has been shown in previous studies to benefit from the use of scope consistency [11]. With the hardware currently used by Brazos (very fast processors and a relatively slow communication media), programs like Raytrace that have large amounts of communication do not achieve good performance. Raytrace was included in this study to show the effects of scope consistency in Brazos, and to demonstrate that an all software implementation of scope consistency based on a distributed page management algorithm can be beneficial to applications that exhibit the correct sharing patterns. Because all shared data in Raytrace that must be propagated between threads is contained within small critical sections, Brazos is able to reduce the number of messages sent by 202% through a reduction in false sharing. This reduction leads to a 614% decrease in the number of bytes sent for Raytrace by not invalidating pages that are written outside of these critical sections. As a result, Brazos obtains a speedup of 1.64 on 8 processors, whereas both TreadMarks implementations are so communication bound that a slowdown in moving from 1 to 8 processors was observed.

Although Water performs better under both Windows NT implementations, it benefits minimally from the use of multicast and user multithreading. Consequently, Water does not perform appreciably better under Brazos than TMK-NT. This is mainly because over 70% of the messages in Water are synchronization messages, which currently do not benefit from the use of multicast. More detailed results on these and other application programs can be found in [22].

5. Related Work

We are aware of only one other software DSM system built using Windows NT [12]. The Millipede project provides a simple programming interface and portability, while implementing adaptive measures such as thread migration and load balancing. Brazos builds upon ideas from several earlier DSM systems, including Ivy [18], Munin [6], and TreadMarks [13]. To the best of our knowledge, Amoeba [20] is the only other DSM system to make use of group communication (multicast), although recent industry interest (most notably the IP Multicast Initiative from Stardust Technologies) in the use of multicast may make the use of multicast more widespread.

The implementation of DSM on SMP machines has been addressed in several systems. In [10], a multiple writer protocol with automatic updates is described. This design relies on specific hardware extensions to implement automatic update, whereas Brazos uses commodity PC's and networks. Erlichson et al. [7] describe a single-writer, epoch-based release consistency DSM design for SMP machines. They conclude that network bandwidth limited the performance potential of this approach. We have shown that network traffic can be reduced significantly through the use of multicast and adaptive protocols.

Scope consistency was first introduced in [11] and relied on specific hardware support provided in the SHRIMP multicomputer [3] to achieve performance gains over a software-only implementation of LRC. Software-only scope consistency models have been proposed [25], but these systems are home-based systems rather than a distributed page-based system like Brazos.

6. Conclusions and Future Work

This paper has described Brazos, a software DSM system that runs under Windows NT 4.0 on a network of PC servers. We have demonstrated that such a system can be competitive with networks of Unix computers for scientific applications, despite the lower per-message overhead of Unix. This was accomplished by taking advantage of available local hardware coherence mechanisms, support for multithreading, a relaxed consistency model, and selective multicast. We have briefly presented these concepts, and have shown their aggregate effect on the performance of five scientific applications. We are working to improve the performance of the adaptive techniques presented here, as well as to develop new techniques. We are also investigating mechanisms for thread migration across distributed processes, thread checkpoint and restart, and support for multiprogramming. Finally, we are working on a faster transport protocol to reduce the high startup overhead we currently observe with WinSock.

Source code for Brazos, including future enhancements, will be made available for non-commercial use via the World Wide Web in the near future.

References

[1] C. Amza, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. Software DSM Protocols that Adapt between Single Write and Multiple Writer. In *The Third International Symposium on High-Performance Computer Architecture.* p. 261-271, 1997.

[2] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type - Specific Memory Coherence. In *Proceedings of the 1990 Conference on the Principles and Practice of Parallel Programming.* p. 168-176, 1990.

[3] M.A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E.W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture.* p. 142-153, 1994.

[4] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors*. 1987: Holt, Rinehart and Winston, Inc.

[5] J.B. Carter. *Efficient Distributed Shared Memory Based on Multi-Protocol Release Consistency*. Ph.D.Thesis, Rice University, Houston, 1993.

[6] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems. *Transactions on Computer Systems*, **13**(3):205-243,1995.

[7] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. Soft FLASH : Analyzing the Performance of Clustered Distributed Virtual Shared Memory. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Systems.* p. 210-220, 1996.

[8] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J.L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture.* p. 15-26, 1990.

[9] S.K. Gupta, A.A. Schaffer, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. Integrating Parallelization Strategies for Linkage Analysis. *Computers and Biomedical Research*, **28**:116-139,1995.

[10] L. Iftode, C. Dubnicki, E.W. Felton, and K. Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *The 2nd IEEE Symposium on High-Performance Computer Architecture.* p. 14-25, 1996.

[11] L. Iftode, J.P. Singh, and K. Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. In *The 8th Annual ACM Symposium on Parallel Algorithms and Architectures.* 1996.

[12] A. Itzkovitz, A. Schuster, and L. Wolfovich, *Millipede: Towards Standard Interface for Virtual Parallel Machines on Top of Distributed Environments*, Technical Report 9607, Technion IIT, 1996.

[13] P. Keleher. *Lazy Release Consistency for Distributed Shared Memory*. Ph.D.Thesis, Rice University, 1995.

[14] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the 1994 Winter Usenix Conference.* p. 115-131, 1994.

[15] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture.* p. 302-313, 1994.

[16] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Distributed Multiprocess Programs. *IEEE Transactions on Computers*, **C-28**(9):690-691,1979.

[17] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, **25**(3):63-79,1992.

[18] K. Li. Ivy: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the 1988 International Conference on Parallel Processing.* p. 94-101, 1988.

[19] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. Ph.D.Thesis, Yale University, 1986.

[20] S.J. Mullender, G.V. Rossum, A.S. Tanenbaum, R.V. Renesse, and H.V. Staveren. Amoeba - A Distributed Operating System for the 1990s. *IEEE Computer*, **23**(4):44-53,1990.

[21] J.P. Singh, W.-D. Weber, and A. Gupta, *SPLASH: Stanford Parallel Applications for Shared-Memory*, Technical Report CSL-TR-91-469, Stanford University, 1991.

[22] E. Speight. *Efficient Runtime Support for Cluster-Based Distributed Shared Memory Multiprocessors*. Ph.D. Thesis, Rice University, Houston, 1997.

[23] Stardust Technologies. *Windows Sockets 2 Application Programming Interface*. 1996.

[24] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. Methodological Considerations and Characterization of the SPLASH-2 Parallel Application Suite. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture.* p. 24-36, 1995.

[25] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation.* p. 75-88, 1996.