The following paper was originally published in the

Proceedings of the 3rd USENIX Windows NT Symposium

Seattle, Washington, USA, July 12–13, 1999

EFFICIENT USER-LEVEL THREAD MIGRATION AND CHECKPOINTING ON WINDOWS NT CLUSTERS

Hazim Abdel-Shafi, Evan Speight, and John K. Bennett

Efficient User-Level Thread Migration and Checkpointing on Windows NT Clusters[†]

Hazim Abdel-Shafi, Evan Speight, and John K. Bennett
Department of Electrical and Computer Engineering
Rice University
Houston, Texas
{shafi, espeight, jkb}@rice.edu

Abstract

Clusters of industry-standard multiprocessors are emerging as a competitive alternative for large-scale parallel computing. However, these systems have several disadvantages over large-scale multi-processors, including complex thread scheduling and increased susceptibility to failure. This paper describes the design and implementation of two user-level mechanisms in the Brazos parallel programming environment that address these issues on clusters of multiprocessors running Windows NT: thread migration and checkpointing. These mechanisms offer several benefits: (1) The ability to tolerate the failure of multiple computing nodes with minimal runtime overhead and short recovery time. (2) The ability to add and remove computing nodes while applications continue to run, simplifying scheduled maintenance operations and facilitating load balancing. (3) The ability to tolerate power failures by performing a checkpoint before shutdown or by migrating computation threads to other stable nodes. Brazos is a distributed system that supports both shared memory and message passing parallel programming paradigms on networks of Intel x86-based multiprocessors running Windows NT. The performance of thread migration in Brazos is an order of magnitude faster than previously reported Windows NT implementations, and is competitive with implementations on other operating systems. The checkpoint facility exhibits low runtime overhead and fast recovery time.

1. Introduction

Recent advances in multiprocessor and network performance have made cluster-based computing an increasingly cost-effective option for large-scale parallel computing. Distributed systems constructed of industry-standard multiprocessors and networks offer an excellent price-to-performance ratio compared with monolithic multiprocessor systems, especially for large systems. Advances in user-level communication mechanisms [4], memory consistency models [12, 14], and network technologies (e.g. [4] and [9]) have improved the performance of these systems significantly. However, several issues limit the widespread adoption of clustered multiprocessors for distributed parallel computing. First, clusters of multiprocessors have an increased risk of failure simply because there are more components that might fail. This tendency to fail is exacerbated by the presence of other users running applications that might cause the system to crash. Second, multiple instances of the operating system running concurrently on each node require more administration and maintenance than a single operating system. Finally, reducing inequitable load distributions in parallel applications may require a system-level solution. Addressing these issues necessitates effective thread migration and checkpointing capabilities.

In this paper we describe the design and implementation of user-level mechanisms for thread migration and checkpointing. Together, these mechanisms support the following functionality:

- Tolerating the failure of multiple computing nodes with minimal runtime overhead and recovery time.
- Addition and removal of computing nodes while applications continue to execute.
- Handling power failures on systems with limited power backup by performing a checkpoint before shutdowns are triggered, or by migrating computational threads to other stable nodes.
- Distribution of the computational load among nodes in a cluster.

We have implemented both mechanisms within the Brazos system [21], a Windows NT-based parallel programming environment that runs on industry standard networks of Intel x86-based multiprocessors. In addition to providing programmers with the

[†] This research was supported in part by grants and donations from Compaq Computer Corporation, Intel Corporation, Microsoft Corporation, Packet Engines, Inc., and by the Texas Advanced Technology Program under Grant No. 003604-022.

abstraction of running on a single shared memory multiprocessor, Brazos supports message passing by implementing the MPI library [20].

Thread migration in the context of a distributed system involves the movement of a computation thread from one currently executing process to another running process. Thread migration has been previously proposed as a tool for load-balancing and communication reduction in distributed shared memory systems [13, 23]. Our work extends the use of thread migration to fault tolerance and cluster management. Migration can be used to tolerate shutdowns due to scheduled maintenance or power loss by dynamically moving all computation threads and necessary data of the application to another available node, without restarting the application. Migration can also be used to add or remove multiprocessor nodes on-the-fly by relocating existing computation threads to the new nodes as appropriate. Finally, the runtime system or programmer may elect to migrate a thread to another node in cases where moving the thread to the data is a better option than moving the data to the thread.

Applications that run for a long time or that require high-availability need a means of recovering from failures, while minimizing the runtime overhead required to ensure recoverability. Previous work in distributed fault tolerance schemes can be categorized as either transaction or checkpoint-based, although combinations of both have been used. Transaction-based recovery is similar to database recovery, in that the distributed system maintains a list of memory transactions or messages [5]. Single node failures can be tolerated by replaying the transactions related to the failed node. Checkpointing is used to save the state of a process. In case of a failure, the checkpoint files are applied and computation can proceed from the point of the last checkpoint [1, 22]. Systems that combine transactions and checkpoints attempt to minimize the amount of work lost due to failure as well as the space requirements for recovery data.

Our implementation of checkpointing is distinguished in two ways. First, we minimize the amount of data saved during a checkpoint operation by leveraging some of the existing coherence-related information available in the Brazos runtime system. This reduces both the overhead required to create checkpoints and the time needed to recover from failures. Second, our checkpoint facility can be initiated either explicitly upon user request or implicitly using predetermined checkpointing intervals. Our results indicate that the facility, given an appropriate choice of checkpoint interval, exhibits low execution time overhead and fast recovery times.

The rest of the paper is organized as follows. In Section 2 we described the design and performance of the Brazos thread migration mechanism. Section 3 contains a similar analysis of the Brazos checkpointing mechanism. In Section 4, we describe how thread migration and checkpoints can be combined to perform several fault tolerance and cluster management functions. Related work is discussed in Section 5. We conclude and describe future research directions in Section 6.

2. Thread Migration

This section describes the design issues that must be addressed to implement a thread migration mechanism, as well as the specific implementation decisions appropriate for use in a Windows NT environment.

2.1. Thread Contexts and Stacks

A thread in Windows NT is comprised of the processor's register set (or context), a thread-specific stack, and a Windows NT-specific area called Thread Local Storage (TLS) [18] intended to contain data instanced per thread. In order to migrate a thread from one process to another, a thread's stack, context, and the user portions of the TLS (Brazos uses the TLS for certain runtime system information) have to be packaged and sent to the remote node. Upon receiving the thread migration message, the remote process copies the contents of the remote thread's stack into a local thread's stack and *injects* the context and TLS data of the remote thread into that of the local thread.

Since stacks may include pointers that reference stack data, a mechanism must be in place to guarantee that these pointers have the same meaning on the new host (issues related to heap and shared memory coherence are addressed in Section 2.3). Furthermore, the stack contains the saved state from any functions executed before migration, implying that both nodes must have the program code loaded at the same virtual address. Code location is not an issue for Brazos, since a distributed application executing on the Brazos system employs multiple instances of the same program.

Two solutions addressing stack data pointer management have been proposed [13, 23]. First, the destination host can scan the received stack data and adjust or remap any pointers encountered by adding the appropriate offset. This solution has several potential problems:

- Stack data may be misaligned, making it difficult to identify the location of pointers.

- Actual variables stored on the stack may contain values that are similar to stack addresses. Changing such values will likely result in incorrect computation.
- Stack pointer values may reside in processor registers, making it necessary to also examine register contents and adjust them accordingly.

These problems make the scanning or remapping of pointers unattractive in the general case. We have chosen to adopt the alternative solution, in which the destination thread's stack must be located at the same virtual address as the source thread's stack [13, 23]. We ensure that the stacks of both threads begin at the same virtual address by reserving the thread stack space for all user threads that may exist during the execution of the distributed process. During the Brazos runtime system initialization on each node, we create a number of threads equal to the total number of user threads executing on all nodes. A potential problem with this approach is the memory and operating system overhead for each thread created. Upon thread creation, Windows NT reserves a default 1MB region from the process' virtual address space for the thread's stack. However, only 2 pages (a total of 8KB on x86-based systems) of memory are initially committed. The amount of memory committed for a stack then increases as needed. The operating system also reserves the necessary internal data structures needed for manipulating and scheduling threads. Since user processes may address up to 2GB of virtual memory (3GB on Enterprise Server systems), we believe that the cost in terms of the memory space wasted by idle user threads is relatively low. In addition, since typical thread stack sizes are often much less than the default 1MB provided for by Windows NT, the amount of wasted address space may be reduced by lowering the maximum thread stack size to a more appropriate value.

2.2. Win32 Support for Thread Migration

Because Windows NT threads are managed by the operating system, we need a mechanism to find a thread's stack and context before we can perform migration. The Win32 API provides several functions that are used to manipulate both thread state and virtual memory [18]. A thread's context may be acquired and set using the GetThreadContext and SetThreadContext functions, respectively. In order to find the thread's stack, we first acquire the thread's current stack pointer, which is part of a thread's context. The Win32 VirtualQuery function is then used to determine the region of committed memory associated with the thread's stack. At this point, the thread's state is completely known, and the context and stack are copied into a message buffer. The migrating thread on the local machine is suspended using the Win32 SuspendThread call[1], and the migration message is sent to the destination node. Upon receipt of the message, the destination node sets the local thread's context, copies the stack data, and activates the thread using the ResumeThread routine. Since Brazos uses UDP, the destination node explicitly acknowledges that the migration message was successfully received.

Migrating threads with open files are handled as follows. The Win32 calls that access files are intercepted by a wrapper function that saves the parameters necessary to reopen the files after migration (i.e., the name of the file, its sharing mode, read/write mode etc.) [11]. In addition, the current file pointer values are determined using standard Win32 calls. This information is then transmitted to the new node and used to reopen the appropriate files and reset their file pointers. Since the file handles used by the migrated thread will be different than the handles created at the new node, we include a mechanism for mapping file handles from the handle used by the thread to the actual handle at the new node. The same wrapper functions used to save access parameters also take care of this mapping. This mechanism requires that all nodes have access to a common file system. File contents are not migrated; however, users have the option of flushing output file buffers prior to migration. A similar mechanism is used for checkpoint and recovery, as described in Section 3.3.

2.3. Ensuring Correctness

There are two correctness issues that arise when threads are allowed to migrate: the effects of thread migration on shared memory coherence mechanisms, and the management of static (linker allocated) or non-shared heap data during thread migration [13, 23]. As described earlier, a thread's context and its automatic or stack variables are maintained throughout the migration process; however, no explicit actions are performed with respect to shared or static memory that may be accessed by a thread. In order to move such data, we need a method to identify static variables or non-shared heap memory that are only accessed by the thread being migrated. Although it is possible to construct a memory map for a Windows NT process [22] and to migrate any such data detected, it is both difficult and time consuming to do so. In order to achieve fast

[1] We initially used Win32 synchronization primitives (e.g., WaitForSingleObject) to suspend and resume threads. During development, we discovered that in order to correctly set the thread's context, it must be executing in user code where it is suspended explicitly using SuspendThread.

thread migration, our approach requires that the programmer ensure that no read/write static or non-shared heap data is used as thread-private data. All read/write data private to a thread must be allocated in Brazos shared memory. Since the runtime system will automatically migrate private data allocated in shared memory when it is accessed by a thread, this rule ensures that private data can be accessed by the migrating thread on any node. Aligning such structures on page boundaries reduces the chances of other threads causing more communication through false sharing.

Shared memory coherence issues increase in complexity if relaxed consistency models are employed [23]. For example, Brazos implements a multiple writer protocol and two relaxed memory consistency models: Release Consistency (RC) [8] and Scope Consistency (ScC) [12]. Both of these models allow multiple nodes to modify different portions of a virtual page concurrently, and only perform coherence or consistency actions at specific synchronization points.

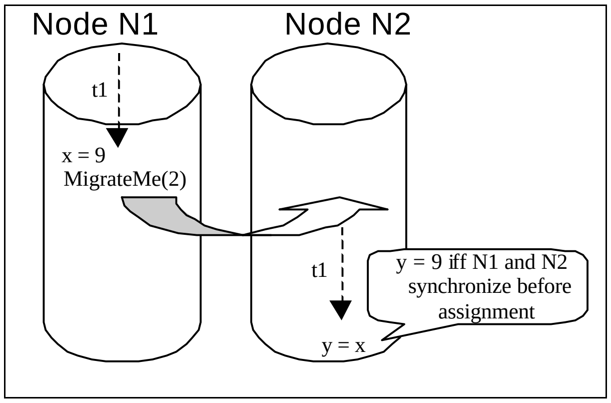

Figure 1: Correctness Issues in Thread Migration

Figure 1 identifies the problems that can occur during thread migration. Consider a thread *t1*, which initially resides on node *N1* and modifies a shared variable *x*. If thread *t1* is subsequently migrated to node *N2*, it may access the variable *x*, which could either contain a valid or stale value depending upon whether an appropriate synchronization event[2] occurred between the initial modification and the access following migration. The simplest approach to this problem requires that a global synchronization operation, such as a barrier, be performed before migration can take place. Another solution includes shipping coherence information with thread migration messages, possibly resulting in slower overall migration performance. Our current approach is to allow thread migration to occur whenever most appropriate, and to require the programmer or runtime system to synchronize as necessary prior to migrating.

[2] A barrier is sufficient for both RC and ScC. Locks are sufficient for RC, but for ScC the modification of *x* occurs within a lock's scope on *N1*. The same lock has to be acquired by a thread on node *N2* before the correct value of *x* may be read.

2.4. Thread Migration API

Our implementation of thread migration adds a single function to the Brazos API: MigrateMe(DestNode). A thread calls this function when it wants to migrate itself to another node. When a thread invokes this function, it is suspended, a special thread called the *Migration Agent* awakens, and the calling thread is migrated to the specified node. At the destination node, the corresponding thread continues execution from the same point in the program. The thread on the original node remains indefinitely suspended unless the same thread migrates back.

2.5. Performance and Analysis

The performance measurements described here were performed on a network of Compaq Professional Workstation 8000's. Each system contains four 200MHz Pentium Pro processors with 256KB of L2 cache. The systems are equipped with two 33MHz 32-bit PCI network interfaces: Fast Ethernet and Gigabit Ethernet. Each node contains 256MB of main memory and an Ultra-Wide SCSI-3 disk controller and drive.

Parameter	**FastEthernet**	**Gigabit Ethernet**
Win32 calls	0.089 ms	0.089ms
Network/copy/synch.	1.121 ms	0.921 ms
Total Migration time	1.21 ms	1.01 ms

Table 1: Performance of Thread Migration in Brazos with 4KB Stacks

We measured the performance of thread migration by executing 1000 back-to-back thread migrations between two nodes. Each iteration contains two calls to the MigrateMe() function. Each call suspends the local thread and places the stack and context in a message destined for another node. At the receiver, an acknowledgement message is sent back to the originating node for reliability purposes[3]. The stack contents are copied, the context is injected into the local thread, and the local thread is then resumed. We also measured the overhead of the various Win32 calls used in the migration process by averaging over 1000 instances of each call. As shown in Table 1, thread migration takes between 1.01 and 1.21ms, depending on the network interface used. Communication accounts for the majority of the migration time.

[3] There were no retries or dropped messages observed during the performance measurements presented here.

Win32 Function (No. of calls)	Cost per call
GetThreadContext (1)	27 μsec
SetThreadContext (1)	27 μsec
SuspendThread (1)	14 μsec
ResumeThread (1)	5 μsec
VirtualQuery (2)	8 μsec

Table 2: The Cost of Win32 Calls Used in Thread Migration

Table 2 identifies the fixed costs associated with migration. Certain costs associated with migration, such as the time needed to wake up the migration agent thread and the time needed to schedule threads after being migrated, are difficult to quantify. The overhead of the Win32 calls that we measured was low compared to the network, stack copying, and synchronization overheads.

The performance of the Brazos thread migration mechanism is an order of magnitude faster than that reported for the Millipede Windows NT-based system [13], and is competitive with other user-level thread migration implementations (see Section 5.1).

3. Checkpoint and Recovery

The ability to tolerate faults becomes increasingly important as the number of multiprocessors in a cluster grows. This is because the probability of failure increasing along with the complexity of the system: a system with sixteen power supplies is more likely to experience power supply failure than a system with one. This is an especially important concern for long-running applications. Additionally, clusters of multiprocessors may be geographically distributed with different local loads that vary over time. Clustered systems need to be able to adapt to these variations. In the extreme, it may be necessary to move all threads off a particular node, and then resume them elsewhere. Checkpoint and restart is an effective mechanism for this situation. Finally, since maintenance and upgrade functions typically require the interruption of service, fault tolerance support allows processors or systems to be shutdown and restarted, thus interrupting rather than terminating the running application. In the remainder of this section we describe the Brazos checkpointing mechanism.

3.1. Checkpointing in Brazos

In order to checkpoint a running Windows NT process, it is necessary to save its state. A process' state includes the stacks and contexts of all threads that exist in that process, the contents of memory (the heap and static data areas, code may not be important), and any operating system objects owned by the process, such as open file handles, synchronization objects, etc. In a networked computing environment, it is also necessary to recreate any network connections upon recovery. This may require additional information concerning the overall composition of the cluster to be saved. The amount of time needed to create a checkpoint is a performance concern because of the potential for a large amount of process-specific information. For a checkpointing mechanism to be practical, it must incur low overhead during normal operation and allow recovery in substantially less time than the potential time lost due to failure.

The structure of a Windows NT process within the Brazos environment can be broken down into the following components: user and runtime system threads, shared memory, static or other heap memory, operating system handles for various synchronization objects and files, and network connections to other nodes participating in the computation. We will discuss each of these components in the context of checkpointing and recovery. Before discussing these details, it is necessary to review the mechanics of starting a Brazos distributed process.

Before a user can execute a Brazos parallel application, a configuration file must be created. The configuration file contains information about the executable program, the names of nodes participating in the computation, and the number of user threads on each node. The program executable, configuration files, and input data files must exist on a shared disk volume accessible from all nodes. Every node that hosts a Brazos process runs a Windows NT service responsible for starting Brazos processes on the local node. The first node listed in the configuration file starts execution by sequentially sending process start requests to other nodes. To avoid deadlock, each Brazos process listens on two network ports: one for requests and the other for replies. The runtime system creates Brazos system threads to handle requests and replies from the network. Once the Brazos runtime system is initialized, the user's application program is started. Programs typically allocate shared memory and initialize data structures using input files at the beginning of execution. The application then proceeds until completion. Several characteristics of the Brazos runtime system facilitate checkpointing:

- The initialization of Brazos is identical for any given configuration file; therefore, it is not necessary to checkpoint most of the runtime system-specific entities. For example, it is not necessary to checkpoint the runtime system's threads, sockets, and most other data structures because they can be easily recreated at recovery by rerunning the initialization routines.

- Brazos synchronization objects such as locks and barriers do not need to be saved because checkpoints are created only at barriers. Thus synchronization objects can be reinitialized during recovery.
- Since checkpoints are independent of the Brazos runtime system initialization process, it is possible to recover processes using different configuration files. This proves to be a valuable feature, allowing recovery on a different number of nodes than were present when the checkpoint is made. This allows a Brazos distributed application to be moved to accommodate varying system loads without seriously affecting performance.

3.2. Memory Issues

Although the Brazos parallel programming environment supports both shared memory and message passing programs, we only discuss issues related to shared memory checkpointing in this paper. The checkpoint facility must save a consistent view of memory to permit full recovery. Memory coherence in Brazos is maintained at the granularity of a virtual page. Operating system support for virtual memory is used to protect pages, and special virtual memory exception handlers are used by the runtime system to ensure that coherence is maintained. Since Brazos supports multiple writer protocols [3], if a node performs a store to an address on a page, a *twin* or duplicate copy of the page is made before the store is allowed to complete. The twin is later used to create a *diff* when another node requires a copy of the page. A diff is a list of addresses and values for locations that have been modified on a particular page. Since multiple nodes may modify a single page simultaneously, multiple nodes may contain diffs for each page. When a node faults on such a page, it will receive diffs from all nodes with modifications to the page in order to reconstruct a valid version of that page. The Brazos runtime system maintains the state of every shared memory page, and this state is used by the checkpoint facility to examine the status of every page and save the necessary coherence-related information, including diffs or twins.

3.3. Implementing Checkpoint

In order to guarantee that the state of pages is consistent and that no coherence actions are pending, we only create checkpoints at global synchronization points such as barriers. At the checkpoint, all data necessary to perform a recovery operation is stored. These data include all shared memory pages, including any twins and diffs and their related runtime system status data structures. In addition, all user threads and their contexts and stacks are saved using the methodology described in Section 2. Since Brazos includes multiple instances of barrier data structures, it is necessary to save the number of the barrier instance used at the time of creating the checkpoint. This ensures that the barrier structures are updated before threads are resumed after recovery. The time consumed in creating checkpoints is a function of the amount of shared memory in use at each of the nodes in the system, but is considerably smaller than the size of the running process (see Section 3.5).

Before a checkpoint is initiated at a barrier, all computation threads have to arrive at that barrier. The thread currently responsible for managing synchronization performs the necessary communication with other nodes to supply coherence information. When all nodes have arrived at the barrier, a message is sent to release all nodes. If a checkpoint is to be performed, threads are not immediately resumed upon receiving the barrier completion message. Instead, a *Checkpoint Agent* thread initiates the checkpoint. Application threads resume once the checkpoint is completed.

Open files during checkpoint and recovery are handled in much the same manner as open files during thread migration (see Section 2.2). File access functions are wrapped and the necessary parameters are saved. This information is stored in the checkpoint file and is used to reopen the files and to set their file pointers during recovery. File contents are not included in the checkpoint; however, users have the option of flushing output file buffers during checkpoint creation.

Two optimizations were added to our checkpointing facility. First, checkpoint files are initially written to each node's local disk to improve performance. In order to allow recovery from other nodes, we also copy the checkpoint files to a network file system. In order to hide the latency of the copy process, the checkpoint agent wakes up the computation threads as soon as the local checkpoint files are written. The copy is performed in the background while the application continues execution. Second, to minimize the size of checkpoint files, we modified the runtime system to keep track of shared memory pages that are allocated and used before the checkpoint takes place and only save information related to modified pages. This substantially reduces the size of checkpoint files by eliminating the need to copy the full pool of shared memory pages.

3.4. Programmer Interface

There are a small number of checkpoint-related issues that must to be addressed by the programmer. Since the checkpoint facility only saves shared memory pages, the programmer allocates all application data (except for stack variables) in shared memory, even if they will not be shared in practice. This restriction may be ignored only for static or heap data that is read-only (note that this restriction is already enforced in order to provide thread migration as described in Section 2.3). Variables that are declared and initialized in this manner will be reinitialized upon recovery and execution will proceed correctly.

Either the user or the runtime system may initiate a checkpoint. Programmers are able to specify a flag to a barrier call that instructs the system to perform a checkpoint or recovery operation. Since users will likely be more concerned with the amount of time potentially lost due to failure, Brazos also supports an automatic checkpoint interval that can be specified in the configuration file, on the command line, or at any point in time during execution using the Brazos user interface. Using the interval method, when the time since the previous checkpoint exceeds the specified interval, the runtime system instructs all nodes to perform a checkpoint. A user interface initiated checkpoint is performed at the next barrier instance. This feature is useful for planned shutdown situations and avoids checkpoint overhead unless necessary. All Brazos programs must explicitly include at least one recovery barrier. For programs that do not use barriers, our current implementation requires inserting additional synchronization. We are investigating techniques to perform checkpoint and recovery without barriers.

Brazos provides two types of recovery mechanisms: automatic recovery on the remaining nodes, or recovery with the addition of a new node to replace the failed node. For recovery on a replacement node, a new configuration file identifying this node is created using the Brazos user interface. Then, the application program calls the recovery process at a barrier placed after the runtime system and application initialization phases in each Brazos process. The only application initialization required before recovery is shared memory allocation and global read-only variable initialization. After recovery, all threads exit the recovery barrier and continue execution from the last checkpoint. During automatic recovery, the Brazos runtime system detects that a node has failed through the use of *heartbeat* messages that are sent out by a designated node every 10 seconds (the heartbeat period is user-selectable). When a node fails, the remaining nodes will restart their Brazos processes from the last successful checkpoint, and the threads from the failed node will be distributed to the surviving nodes in a round-robin fashion. We currently do not attempt to optimize the assignment of threads to nodes.

3.5. Checkpoint Performance

To demonstrate the checkpointing facility, we used two applications: Water, a molecular dynamics application from the SPLASH benchmark suite [19] using a 4096 molecule dataset, and a locally-written SOR (successive over relaxation) using 4000×4000 matrices. Both applications were run on four nodes, each with a single user thread. Water is intended to be representative of applications with a small shared memory footprint, whereas SOR is intended to be representative of applications with a large shared memory footprint. To measure the overhead of the checkpoint facility, we used the system initiated checkpoint mode while varying the checkpoint interval from two to thirty two minutes.

Ckpt interval in min (number of ckpts)	Execution time in minutes	Ckpt overhead as % of exec time
(none)	42.23	0%
32 (1)	42.40	0.39%
16 (3)	42.38	0.34%
8 (6)	42.58	0.82%
4 (11)	43.20	2.28%
2 (20)	43.88	3.90%

Table 3: Checkpoint Overhead vs. Interval Length for Water

Table 3 shows that for Water the overhead of performing checkpoints to network disks every two minutes was about 3.9% of the total execution time. The size of the Water process was about 45MB and the average checkpoint size was 5.2MB per node. Checkpoint time averaged about 1.25 seconds, and recovery time was 21 seconds (17 seconds for runtime system and user initialization, and 4 seconds for actual recovery) on four nodes. When only local files were created, the runtime overhead was essentially the same.

Ckpt interval in min (number of ckpts)	Execution time in minutes	Ckpt overhead as % of exec time
(none)	36.06	0%
32 (1)	36.45	1.08%
16 (2)	37.24	3.27%
8 (4)	37.97	5.30%
4 (9)	40.62	12.65%
2 (19)	46.18	28.06%

Table 4: Checkpoint Overhead vs. Interval Length for SOR

As Table 4 shows, the checkpoint execution time overhead of SOR (with a 122MB footprint) is relatively high for low checkpoint interval settings (28% for two-minute intervals). These results suggest that checkpoint intervals of eight minutes or higher are more appropriate in this case, resulting in runtime overheads of about 5% or less. The process size for SOR was about 163MB; the average size of the checkpoint file was 62.1MB. The average checkpoint time was 25.8 seconds, and recovery time was 46 seconds (19 seconds for runtime system and user initialization and 27 seconds for data recovery).

In order to improve checkpoint performance for large applications, we are adding support for incremental checkpoints. Incremental checkpoints reduce runtime overhead in two ways. First, they eliminate the need to stall the computation threads at barriers by allowing checkpoint operations to be performed in parallel with computation. Second, the amount of data copied to checkpoint files is reduced because only modifications made since the previous full checkpoint need to be saved. This is accomplished by tracking pages that have been modified since the last checkpoint and only saving the information necessary to apply these changes from the last checkpoint instance. We are presently implementing this mechanism.

4. Integrating Thread Migration and Checkpoint/Recovery

Thread migration can be used to allow the addition or removal of nodes participating in a distributed application. To add a node, the application is started on the new node and network connections are established with all nodes currently participating in the computation (as described by the configuration file) during the runtime system's initialization phase. The root process then takes note of the added node and sets a special flag. The current barrier manager uses this information to synchronize all existing nodes with the new node. Any threads that need to be migrated to the new node are moved before execution resumes.

Shared memory accesses on the new node are handled in the usual way by the Brazos runtime system; however, all static or non-shared heap data have to be initialized before the node can participate in the computation. For this purpose, an initial user thread is used to perform any required user data initialization. In addition, it is necessary to modify some runtime system data structures to reflect the addition of the new node. This work is accomplished at the barrier at which the new node is detected.

Removing nodes is basically the reverse process. An important difference is that once the computation threads have migrated to other nodes, the contents of shared memory at the node to be removed need to be transferred to other nodes before the process is terminated. This is accomplished by creating a checkpoint of the shared memory state of the node to be removed. All remaining nodes then apply the necessary changes to their state using the contents of the checkpoint file, and update the appropriate runtime system information to reflect the departure of a node before proceeding with computation.

In case of a power failure, our system triggers checkpoints of all running distributed processes before the system is shutdown. This requires that the system have an UPS with sufficient backup time to allow checkpointing to take place. If other nodes are not affected by the power failure, threads on the affected machine(s) migrate to other nodes using a methodology similar to that of removing and adding a node to the system.

5. Related Work

This section discusses previous relevant work in fault tolerance and thread migration on software distributed shared memory systems. Since we are not aware of any previous work that combines both techniques, related work is separated into two parts. First, we discuss other systems that utilize thread migration. Then we discuss various fault tolerance techniques proposed for distributed shared memory systems. Finally, we discuss work related to checkpointing Windows NT processes.

5.1. Thread Migration

Millipede [13] and D-CVM [23] both employ thread migration. Millipede is a Windows NT-based DSM system that includes a user-level thread migration mechanism similar to ours. In particular, they use the same method of ensuring that all thread stacks are at identical virtual addresses across all nodes, and impose the same restrictions on memory use. Millipede uses thread migration to reduce communication costs by keeping track of all accesses made by a thread that result in inter-node communication. Millipede requires that memory be sequentially consistent [17], which results in lower overall performance. The reported migration time on Millipede is 70ms on Pentium-based systems with 100Mbps Ethernet.

D-CVM uses thread migration to dynamically redistribute computation threads to nodes to reduce communication and improve load-balancing [23]. Instead of relying solely on the tracking of page faults as in Millipede, D-CVM contains an active thread tracking mechanism. This mechanism tracks sharing among local threads by both serializing thread execution and adding per-page access counters for each thread in a DSM process. D-CVM's approach to coherence is similar to ours. It uses a multiple-writer LRC (Lazy Release Consistency [14]) coherence protocol and avoids correctness problems by restricting thread migration to only occur at barriers. We allow migration at other points in the program, as long as certain rules are followed (see Section 2.3). D-CVM also requires all static and heap-allocated thread-private data to be in shared memory. The best reported thread migration performance for D-CVM (using a reserved stack approach similar to ours) was 1.597ms on an IBM SP2 using 66.7MHz Power2 processors over a 40MB/s SP2 switch (with a stack size of 1704 bytes).

Active Threads [24] is a user level thread library that includes support for migration. One of the main goals of the Active Threads package is performance. This goal is achieved by utilizing an efficient user-level communication package based on active messages [6]. Their solution to the stack pointer problem is similar to ours. For SPARCstation-10 multiprocessor work-stations connected by a Myrinet network interface, thread migration latency was reported to be about 1.1ms for 2KB stacks.

5.2. Fault Tolerance

Costa et al. [5] implement a logging and checkpoint facility to recover from single and multi-node failures on the TreadMarks [15] DSM system. They implement a two-level checkpoint mechanism. They use a lightweight logging mechanism to support single node failures and perform occasional consistent checkpoints to implement multiple node recovery. The performance results reported in [5] show that the overhead of maintaining the logs is very small. On the other hand, the re-execution needed for recovery consumed from 72% to 95% of the total execution time of three benchmark applications used (SOR, Water, and TSP). In contrast, Brazos recovery takes 0.8% of the execution time for Water even with a considerably larger dataset. Checkpoint overhead was reported as less than 2% of execution time for both Water and TSP, and around 22% for SOR. They attribute the difference to the size of checkpoints in the respective applications, which are a function of the coherence traffic exhibited by these applications.

Cabillic et al. [1] implement a consistent checkpoint facility that is similar to ours in several respects. They perform checkpoints at barrier synchronization points that are annotated by the programmer. Their checkpoint facility requires the copying of pages and page information blocks only, and does not require saving diffs. This is because their DSM system, MYOAN [2], implements sequential consistency and an invalidation-based protocol. They require that all private data be allocated in shared memory in order to expose it to the checkpoint facility, similar to our system. They implement a special checkpoint server process that requires inter-node communication to perform the checkpointing, whereas we use a checkpoint agent thread per process and communication is through hardware shared memory.

Kermarrec et al. implement a recoverable distributed shared memory system called ICARE [16]. They modified an invalidation-based coherence protocol to maintain a recovery database in volatile memory that enables recovery from single node failures. Their system replicates pages on multiple nodes to allow recovery, which in some cases resulted in improved performance since page faults were avoided.

Previous researchers have developed checkpoint facilities for Windows NT processes [10, 22], although none of these work in a distributed shared memory environment. Huang et al. implement a recovery facility for NT processes, called NT-SwiFT [10]. It includes the Winckp library that can be used for rollback-recovery of NT applications. Their system intercepts system calls and discovers areas of memory to save using standard Win32 calls. Recovery can also be performed on applications that access the network by logging network traffic. Srouji et al. [22] implemented a general-purpose checkpoint facility for multi-threaded Windows NT processes. Similar to NT-SwiFT, they redirect Win32 API calls to a set of wrapper functions that are used to save state information before calling the actual Win32 routines. This enables them to build a database of open files and other handles that need to be recreated at recovery. They describe how data segments are reserved, including static and heap-allocated memory. Checkpoint file sizes were about the same size as the process itself and checkpoint time was twenty one seconds for a 50MB process.

Finally, the Microsoft Cluster Service (MSCS) supports high availability applications such as database servers on Windows NT [7]. The MSCS can detect failures of hardware or software resources and can restart or migrate failed components. MSCS does not support rollback recovery of DSM applications, but handles situations that we do not address, including hardware fail-over.

6. Conclusions and Future Work

We have described the implementation and performance of thread migration and checkpointing mechanisms for clusters running Windows NT. The performance of thread migration was found to be competitive with other systems and an order of magnitude faster than a previously published Windows NT implementation. The checkpoint facility exhibited low runtime overhead and fast recovery times. We are currently implementing an incremental checkpointing mechanism to further reduce the overheads for large applications. We are also implementing additional techniques that combine thread migration and checkpointing for fault tolerance.

We would like to thank Karin Petersen and the anonymous reviewers for their helpful comments on earlier versions of this paper.

Brazos is available free for non-commercial use at http://www-brazos.rice.edu/brazos.

Bibliography

[1] G. Cabillic, G. Muller, and I. Puaut. The Performance of Consistent Checkpointing in Distributed Shared Memory Systems. In *Proceedings of the 14th Symposium on Reliable Distributed Systems*, September 1995.

[2] G. Cabillic, T. Priol, and I. Puaut. MYOAN: An Implementation of the KOAN Shared Virtual Memory on the Intel Paragon. IRISA, Research Report 812, March 1994.

[3] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pp. 152-164, October 1991.

[4] Compaq Corporation, Intel Corporation, and Microsoft Corporation. *Virtual Interface Architecture Specification, Version 1.0.* , 1997.

[5] M. Costa, P. Guedes, M. Sequeira, N. Neves, and M. Castro. Lightweight Logging for Lazy Release Consistent Distributed Shared Memory. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pp. 59-73, 1996.

[6] T. v. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrating Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pp. 256-266, 1992.

[7] R. Gamache, R. Short, and M. Massa. Windows NT Clustering Service. *IEEE Computer*, vol. 31(10), pp. 55-62, 1998.

[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*, pp. 15-26, May 1990.

[9] R. W. Horst. *TNet: A Reliable System Area Network*. Tandem Computers, 1995.

[10] Y. Huang, P. E. Chung, C. Kintala, C.-Y. Wang, and D.-R. Liang. NT-SwiFT: Software Implemented Fault Tolerance on Windows NT. In *Proc. of the 2nd USENIX Windows NT Symposium*, pp. 47-55, August 1998.

[11] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, July 1999.

[12] L. Iftode, J. P. Singh, and K. Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 277-287, June 1996.

[13] A. Itzkovitz, A. Shuster, and L. Shalev. Thread Migration and its Applications in Distributed Shared Memory Systems. *Journal of Systems and Software*, vol. 42, pp. 71-87, 1998.

[14] P. Keleher. *Lazy Release Consistency for Distributed Shared Memory*. Ph.D. Dissertation, Department of Computer Science, Rice University, Houston, TX, January 1995.

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

[16] A.-M. Kermarrec, C. Morin, and M. Banatre. Design, Implementation and Evaluation of ICARE: An Efficient Recoverable DSM. *Software--Practice and Experience*, vol. 28(9), pp. 981-1010, 1998.

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

[18] J. Richter. *Advanced Windows*. Third Edition., Microsoft Press, 1997.

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

[20] E. Speight, H. Abdel-Shafi, and J. K. Bennett. An Integrated Shared-Memory/Message Passing API for Cluster-Based Multicomputing. In *Proceedings of the Second IASTED International Conference on Parallel and Distributed Computing and Networks*, pp. 146-153, December 1998.

[21] E. Speight and J. Bennett. Brazos: A Third Generation DSM System. In *Proceedings of the 1st USENIX Windows NT Symposium*, pp. 95-106, August 1997.

[22] J. Srouji, P. Schuster, M. Bach, and Y. Kuzmin. A Transparent Checkpoint Facility on NT. In *Proceedings of the 2nd USENIX Windows NT Symposium*, pp. 77-85, August 1998.

[23] K. Thitikamol and P. Keleher. Thread Migration and Communication Minimization in DSM Systems. *Proceedings of the IEEE*, vol. 87(3), pp. 487-497, 1999.

[24] B. Weissman, B. Gomes, J. Quittek, and M. Holtkamp. Efficient Fine-Grain Thread Migration with Active Threads. In *Proceedings of the 12th International Parallel Processing Symposium*, March 1998.