INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®
RICE UNIVERSITY

Reliable Parallel Computing on Clusters of Multiprocessors

by

Hazim M. Abdel-Shafi

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

John K. Bennett, Associate Professor, Chair
Electrical and Computer Engineering

J. Robert Jump, Professor
Electrical and Computer Engineering

Peter Druschel, Assistant Professor
Computer Science

W. Evan Speight, Research Scientist
Electrical and Computer Engineering

Houston, Texas
April, 2000
ABSTRACT

Reliable Parallel Computing on Clusters of Multiprocessors

by

Hazim M. Abdel-Shafi

This dissertation describes the design, implementation, and performance of two mechanisms that address reliability and system management problems associated with parallel computing clusters: thread migration and checkpoint/recovery. A unique aspect of this work is the integration of these two mechanisms. Although there has been considerable prior work on each of these mechanisms in isolation, their integration offers synergistic benefit to both functionality and performance. Used in conjunction, these mechanisms facilitate failure recovery, and node addition and removal with minimal disruption of executing applications. Our implementation differs from previous work in the following ways. First, by using thread migration instead of process migration, the overhead of moving computation among nodes is reduced. Second, because our implementation of checkpoint/recovery separates computation and data, it is possible to distribute data and threads among other nodes during recovery. This is possible because the underlying support for thread migration in the system allows the recovery of a thread from any checkpoint on any node. Third, our implementation does not require repartitioning of a running parallel application when resources are added or removed.
Finally, the checkpoint/recovery and thread migration mechanisms are both implemented at user-level. The benefits of a user-level implementation include ease of development since operating system source code is not required, adaptability to other platforms, and simple upgrades to new versions of the underlying operating system and hardware. The prototype implementation described in this thesis was developed as an extension to the Brazos software distributed shared memory system. Brazos allows multithreaded parallel applications to execute on networks of multiprocessor servers running the Windows NT/2000 operating system.
Acknowledgements

First and foremost, I thank my parents, Majida and Mustafa, for offering me the gift of life, inspiring me with their knowledge and wisdom, and allowing me the freedom to choose my own path. As much as completing this work brings me joy and happiness, I am saddened that you are no longer with us to share the joy, mother. May you rest in peace.

I thank my wife Rana, whose constant encouragement, patience, humor, and love enabled me to endure through this long journey. Most of all, I thank her for putting up with my mental and/or physical absence for so many days and nights. This could not have happened without you Rana. I thank aunt Khalida and uncle Jamil for their encouragement and support.

I thank my advisor, Dr. John Bennett, for being a great source of advice, encouragement, and support, and for being a caring human being. You have made all this work very enjoyable. I’ll miss the latte’s and the barbeque. I thank my friend and mentor Dr. Evan Speight. I shall never forget the countless discussions, phone conversations, lunches, and laughs that we shared. Working with you has been a source of great pleasure.

I thank Professors J. Robert Jump and Peter Druschel for their input and advice on this work, and for serving on my thesis committee. Last but not least, I also thank all my previous teachers. I hope that I have made you all proud.
# Table of Contents

Chapter 1: Introduction .................................................................................................................. 1

1.1 Contributions .......................................................................................................................... 3

1.2 Thread Migration ....................................................................................................................... 4

1.3 Checkpoint/Recovery ............................................................................................................... 5

Chapter 2: Introduction to Brazos .................................................................................................. 8

2.1 Shared Memory ......................................................................................................................... 8

2.2 Message Passing ....................................................................................................................... 12

2.3 The Virtual Interface Architecture ........................................................................................ 13

2.3.1 Connection Procedures ..................................................................................................... 16

2.3.2 Memory Registration ......................................................................................................... 16

2.3.3 Data Transfer Modes ......................................................................................................... 17

Chapter 3: Integrating Thread Migration and Checkpoints ........................................................... 18

3.1 Objectives ............................................................................................................................... 18

3.2 Design Alternatives ................................................................................................................ 19

3.2.1 Environment ..................................................................................................................... 19

3.2.2 Checkpoint/Recovery Alternatives .................................................................................... 20

3.2.3 Adding and Removing Nodes ............................................................................................ 21

3.2.4 Automatic Failure Recovery ............................................................................................. 23
Chapter 4: Thread Migration ................................................................. 26

4.1 Windows NT Threads .................................................................. 26
4.2 Migrating Thread Stacks and Contexts ........................................ 27
4.3 Ensuring Correctness..................................................................... 32
  4.3.1 Thread-Private Data ............................................................. 32
  4.3.2 Interaction With Consistency Models .................................... 34
  4.3.3 Interaction with Synchronization Primitives ......................... 37
  4.3.4 TLS and File System Access ................................................ 39
4.4 Anatomy of Thread Migration ..................................................... 43
4.5 Performance Analysis ................................................................... 49

Chapter 5: The Brazos Checkpoint and Recovery Facility ..................... 54

5.1 Checkpoint and Recovery in Brazos ............................................. 54
5.2 Implementation Details ............................................................... 57
  5.2.1 Saving Data and Thread States ............................................. 58
  5.2.2 The Checkpoint Agent and Checkpoint Initiation ................... 62
  5.2.3 Checkpoint File Management ............................................. 66
5.3 Programmer Interface ................................................................... 67
5.4 Checkpoint Performance ............................................................. 71

Chapter 6: Node Removal .................................................................. 74

6.1 Node Removal Using Checkpoint/Recovery .................................. 75
6.1.1 Memory and Thread Recovery ................................................................. 77
6.2 Automatic Recovery .................................................................................. 82
  6.2.1 Checkpoint-based Node Removal Performance .................................... 87
6.3 Online Node Removal .............................................................................. 88
  6.3.1 Implementation Details ........................................................................ 89
  6.3.2 Online Node Removal Performance ....................................................... 95

Chapter 7: Node Addition .................................................................................. 96
  7.1 Network Communication .......................................................................... 97
  7.2 Memory Allocation .................................................................................. 101
  7.3 Memory Coherence .................................................................................. 103
  7.4 Synchronization ...................................................................................... 104
  7.5 Performance of Node Addition .................................................................. 106

Chapter 8: Related Work ................................................................................. 107
  8.1 Thread Migration ..................................................................................... 107
  8.2 Fault Tolerance ......................................................................................... 116
  8.3 Adaptability and Process Migration ........................................................... 121

Chapter 9: Conclusions and Future Work ....................................................... 123
  9.1 Future Work ............................................................................................. 123
List of Figures

Diagram of the Virtual Interface Architecture ................................................................. 14
Thread stacks in Intel x86-based systems running Windows NT ....................................... 30
Example of a correctness problem using thread migration ................................................ 35
Solving the problem using scope consistency locks ......................................................... 37
Thread Migration Pseudo Code ....................................................................................... 45
Implementation of the MigrateMe API ............................................................................ 47
Anatomy of a Thread Migration Operation .................................................................... 48
Thread migration performance benchmark .................................................................... 49
The Cost of Win32 Calls Used in Thread Migration ....................................................... 50
Thread Migration Latency vs. Stack Size for UDP ......................................................... 51
Thread Migration Latency vs. Stack Size for VIA ......................................................... 52
Checkpoint Creation Pseudo Code ................................................................................. 61
Checkpoint Agent Thread Procedure .............................................................................. 63
Checkpoint File Management ......................................................................................... 65
Outline of an Application Utilizing Checkpoint/Recovery ............................................... 68
Checkpoint and Recovery Configuration File Options .................................................... 70
Checkpoint Execution Time Overhead vs. Checkpoint Interval ...................................... 73
Checkpoint Recovery for Node Removal ....................................................................... 80
Heart Beat Thread Procedure ......................................................................................... 81
Automatic Recovery Process ......................................................................................... 85
Performance of Checkpoint-Based Node Removal ........................................ 87
Online Node Removal Event Handler .................................................................. 91
Online Node Removal Process ............................................................................ 93
Network Connections Phase ............................................................................... 100
Node Addition Process (after connection establishment) .................................... 103
List of Tables

Performance of Important SDSM Operations .......................................................... 12

Shared Memory Page States in Brazos................................................................. 58

Benchmark Applications and Problem Sizes....................................................... 71

Checkpoint and Recovery Statistics...................................................................... 74

Online Node Removal Performance Statistics................................................... 94

Performance of Node Addition............................................................................. 106
Chapter 1 Introduction

Parallel computing systems are typically categorized as either message-passing or shared memory systems, although some combinations of these two parallel computing paradigms have been proposed (e.g., [18, 43]). Although message-passing systems can achieve good performance, the need to explicitly move data as needed makes them generally more difficult to program. In an attempt to mitigate this programming complexity, many research efforts have concentrated on building parallelizing compilers that discover data-dependences and insert appropriate communication primitives to parallelize sequential applications. Shared memory systems, on the other hand, generally offer a simpler programming paradigm that makes it easier to develop parallel applications; however, the simplicity of programming shared memory systems comes at the cost of increased hardware complexity, primarily as a result of the cache coherence problem [3, 4]. The need to maintain cache coherence makes it expensive and difficult to build large-scale hardware cache-coherent multiprocessors. This cost has led to the investigation of software alternatives for constructing shared memory multiprocessors.

One such software approach is the use of clusters of workstations or servers networked together, with an intervening layer of system software. Although clusters are inherently message-passing systems, it is possible to use them as shared memory systems through software. Such systems are variously referred to as software distributed shared memory (SDSM), or software virtual memory (SVM) systems. SDSM systems provide the
programmer with the abstraction of running on a cache-coherent multiprocessor by implementing coherence via a (typically user-level) runtime system. This runtime system relies on the underlying operating system’s paging hardware for detecting accesses to potentially stale data. SDSM systems can be more cost-effective compared to their hardware counterparts because they use off-the-shelf computer systems, including small-scale multiprocessors, and networks. Early SDSM systems did not perform well. This was mostly due to the effects of false-sharing, which is exacerbated because of the page-level coherence granularity and the excessive amount of network communication required to maintain coherence. A large body of research has attempted to ameliorate the execution time penalty associated with software coherence. Recent advances in SDSM systems such as the use of multiple-reader protocols [9], relaxed memory consistency models [2, 16, 22, 26], and fast user-level network technologies (e.g., [19] and [11]), have significantly improved the performance of SDSM systems. Besides their relatively low cost, SDSM systems are easy to upgrade, allowing them to benefit from the latest advances in commodity computers, networks, and operating systems. SDSM systems also offer a simple and relatively inexpensive means of potentially improving performance by adding more resources as they become available.

In addition to performance limitations, clusters are more susceptible to failure compared to monolithic computer systems simply because they are comprised of more components that can fail. This is a significant drawback to the cluster approach because applications that benefit from large-scale systems are typically long running, and thus would be most
affected by system unreliability. Therefore, in order to be used for production work, clusters must incorporate some form of fault-tolerance support. Clusters also present system management issues. Since a cluster is comprised of independent systems, each running its own operating system, there is no single software or hardware component that is responsible for managing the cluster's resources. This makes it problematic to share resources, or perform software maintenance or other operating system management tasks.

1.1 Contributions

This dissertation describes the design, implementation, and evaluation of two mechanisms that address the reliability and system management problems associated with parallel computing clusters: thread migration, and checkpoint/recovery. A checkpoint and recovery mechanism allows users applications to recover from failure. It is possible for some machines to fail while others remain functional. This requires a mechanism that allows the recovery of a failed SDSM application on a cluster comprised of a smaller number of machines. An efficient thread migration mechanism, if present, allows the intentional removal and addition of computing resources from a cluster while applications continue to execute. The ability to migrate threads between processes on different machines facilitates both reliability and system management.

The checkpoint/recovery and thread migration mechanisms described in this dissertation are both implemented as user-level facilities. The benefits of a user-level implementation
include the ease of development since operating system source code is not required, and simple upgrades to new versions of the underlying operating system and hardware. The integration of the mechanisms offers synergistic benefits to both functionality and performance. The functional benefits of this integration include the following capabilities:

- Tolerating the failure of multiple computing nodes, and the automatic recovery from single node failure.
- Addition and removal of computing nodes while applications continue to execute.
- Handling both power loss on systems with limited power backup and intentional shutdowns.

The mechanisms described in this dissertation have been implemented within the Brazos parallel programming environment [47-50]. Brazos supports both shared memory and message passing parallel programming on clusters of multiprocessors running Windows NT 4.0 or Windows 2000. The work described here is concerned only with the shared memory aspects of Brazos.

1.2 Thread Migration

Thread migration in the context of a distributed system involves the movement of a computation thread from an executing process on one system (or node) to another process
running on a second node. Both processes are presumed to be part of a single, parallel, user application running on top of a SDSM system. Thread migration has been previously proposed as a tool for load-balancing and communication reduction in software distributed shared memory systems [24, 55]. This work extends the use of thread migration to fault tolerance and resource 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 [24, 55]. The performance of the thread migration mechanism described in this dissertation is significantly better than that of previously reported mechanisms [24, 55, 57].

1.3 Checkpoint/Recovery

Applications that run for a long time or that require high-availability need a means of recovering from failure, while minimizing the runtime overhead required to ensure recoverability. Previous work in distributed fault tolerance 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 [12]. Single node failures can be
tolerated by replaying the transactions related to the failed node. Checkpoints are 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 [7, 8, 30, 31, 41]. Systems that combine transactions (or logs) and checkpoints attempt to minimize the amount of work lost due to failure as well as the space requirements for recovery data. Space reductions are achieved using checkpoints, instead of just transaction logs. Performance benefit is obtained because reapplying transactions from the logs after the checkpoints are recovered minimizes the amount of work lost. The implementation described here does not maintain transaction logs, but is otherwise distinguished in two ways. First, it minimizes the amount of data saved during a checkpoint operation by leveraging existing coherence-related information available within the Brazos runtime system. This reduces both the overhead required to create checkpoints and the time needed to recover from failure. Second, the checkpoint facility can be initiated either explicitly upon user request or implicitly using user-defined checkpoint intervals. The facility exhibits low execution time overhead and fast recovery times. In addition, the recovery mechanism allows the recovery of multiple checkpoints, enabling the recovery from failures on a cluster that is smaller (in the number of nodes) than the failed one. This feature is demonstrated using an automatic failure detection mechanism that initiates recovery from single-node failures without any user intervention.

There has been no previous work that integrates thread migration and checkpoint/recovery to address high-availability and resource management issues in
SDSM clusters. However, there is a large body of research that treats recoverability
(e.g., [7, 8, 12, 30, 31]) and thread migration (e.g., [24, 55, 57]) separately for SDSM
systems. Previous work on adaptive clusters either requires the use of a specialized
language [40], the ability to repartition computation [45], or a more expensive process
migration mechanism [6, 45, 54].

The rest of the dissertation is organized as follows. Chapter 2 describes the Brazos
system. Chapter 3 includes a discussion of the design space, design decisions, and the
reasons why integrating thread migration and checkpoint/recovery is an attractive choice
for solving the reliability and resource management issues in clusters of multiprocessors.
Chapter 4 presents the details of our thread migration implementation. In Chapter 5, we
present the details of the checkpoint and recovery mechanism. Chapter 6 presents the
details of implementing node removal and automatic single-node failure. Chapter 7
presents the implementation of online node addition. Related work is discussed in
Chapter 8. Conclusions and future work is presented in Chapter 9.
Chapter 2  Introduction to Brazos

Brazos is a parallel programming environment that supports shared memory and message passing parallel applications running on a network of Intel x86-based SMPs (symmetric multiprocessors) using the Microsoft Windows NT operating system. Brazos supports multiple network interfaces, including Ethernet, ServerNet, and the Virtual Interface Architecture (VIA). Section 2.1 describes the details of the shared memory implementation that are most relevant to this work. Section 2.2 describes the message passing implementation, and Section 2.3 describes some details about the Virtual Interface Architecture.

2.1 Shared Memory

The Brazos parallel programming environment provides support for software distributed shared memory (SDSM) and message passing applications on both Winsock-enabled and VI-enabled clusters of SMP workstations [48]. SDSM systems implement a runtime environment that provides users with the abstraction of running on a monolithic (single system) multiprocessor while actually running on a network of workstations. Brazos implements a multithreaded shared virtual memory system using virtual memory protection mechanisms to maintain coherence across multiple nodes, and relies on hardware cache-coherence to maintain coherence between user threads in the same
process. Shared memory data pages are invalidated by setting the page protection attributes such that any access to the page results in a segmentation fault. An access to an invalid page is intercepted by the runtime system, and messages are sent to the set of processes (referred to as the \textit{invset}) with newly written portions of the page. Brazos uses multicast communication both to reduce the number of request messages that must be sent, and to allow responding processes to multicast responses to all processes that currently have copies of the page (referred to as the \textit{copyset}). The use of multicast results in a large reduction in the overall message count for many applications [50]. After the faulting thread receives all portions of the page, computation is allowed to proceed. Multicast is emulated using point-to-point messages on VIA. Changes to a page are tracked by creating a copy of the page (a \textit{twin}) when any thread in a process writes to a page for the first time after the page is made valid. When a remote process requests the data, the twin is used to construct a run-length encoding (a \textit{diff}) of the changes made to the page by comparing it with the data currently on the page. This diff, which is typically much smaller in size than the entire page, is then sent to the requesting process. The performance of the operating system's virtual memory management functions, as well as the protocol used for maintaining coherence, are major determinants of a SDSM system's performance.

Brazos is designed for and implemented on the Microsoft Windows NT operating system. One of the important reasons for choosing Windows NT as the operating system platform was its support for multithreading. Multithreading support is an important feature of the
Brazos system for two main reasons. First, multithreading allows the efficient utilization of multiprocessor systems to improve the performance of user applications. Second, multithreading is used by the Brazos runtime system to minimize the overhead of communication by implementing separate runtime system threads to manage system-level request and reply messages.

To improve performance, the shared memory implementation of Brazos uses a multiple-writer protocol [9] and two relaxed memory consistency models: release consistency [16], and scope consistency [22]. Multiple-writer protocols allow the same shared memory page to be modified by multiple nodes concurrently. This is possible because of the semantics of the memory consistency models used, as explained next.

A memory consistency model defines the rules that govern when the value of a store operation on a memory address has to become available to potential consumers (e.g., load operations) of that value\(^1\). Release consistency, in the context of SDSM systems, states that stores should be propagated to other nodes at release synchronization operations. A release is defined as a barrier arrival, lock release, or flag set operation. Between synchronization points, the system has the option to reorder and overlap some operations to optimize performance. Programs written correctly for release consistency ensure that no data races exist. That guarantees that it is impossible for two threads or nodes to be

\(^1\) This is a simplistic definition of memory consistency that is sufficient for our purposes. More details on memory consistency models may be found in [2,15,16,32].
accessing the same memory location without synchronization. This is the reason why multiple writer protocols are legal under such relaxed consistency models.

The main benefit of release consistency is that it allows the aggregation of coherence traffic into fewer, larger messages instead of many small messages as in sequentially consistent systems [32] (e.g., IVY [33]). At every release point under release consistency, each node in the cluster either invalidates or updates the copies of its modified pages that reside on other nodes. The scope consistency model attempts to further reduce the amount of coherence data exchanged by exploiting the following deficiency in release consistency. In the case of critical sections, release consistency dictates that all modifications prior to the lock’s release must be sent to all other nodes in the system. The key observation is that it is possible to write a correct program that requires updating or invalidating only those pages that were modified within the critical section. Further, it is only necessary to perform these coherence actions at the next acquisition of the lock. Release consistency and scope consistency are identical with respect to memory coherence actions performed at barriers. One possible disadvantage of scope consistency is that it requires some application programs written for release consistency to be modified to ensure correctness [22]. Table 1 lists the performance of some critical SDSM operations for both UDP (on VIA hardware) and native VIA using a cluster of Compaq ProLiant 5500 Servers using GigaNet cLAN GNN1000 VI interface cards.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>UDP</th>
<th>VI</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSM Runtime</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Page fetch (request + reply)</td>
<td>937 μsec</td>
<td>279 μsec</td>
</tr>
<tr>
<td>Barrier (8 nodes)</td>
<td>1153 μsec</td>
<td>192 μsec</td>
</tr>
<tr>
<td>Lock (local)</td>
<td>3 μsec</td>
<td></td>
</tr>
<tr>
<td>Lock (remote)</td>
<td>581 μsec</td>
<td>109 μsec</td>
</tr>
<tr>
<td>Operating System</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Page fault</td>
<td>6.7 μsec</td>
<td></td>
</tr>
<tr>
<td>Page protect</td>
<td>2.7 μsec</td>
<td></td>
</tr>
<tr>
<td>Page registration</td>
<td>N/A</td>
<td>14 μsec</td>
</tr>
<tr>
<td>Base Network Latency (unidirectional)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 byte</td>
<td>241 μsec</td>
<td>10 μsec</td>
</tr>
<tr>
<td>4 Kbyte latency (one page)</td>
<td>422 μsec</td>
<td>51 μsec</td>
</tr>
<tr>
<td>64 Kbyte latency (max msg)</td>
<td>1969 μsec</td>
<td>626 μsec</td>
</tr>
</tbody>
</table>

Table 1: Performance of Important SDSM Operations.

2.2 Message Passing

Brazos includes an implementation of a subset of the MPI message passing library [39]. The implementation supports both Winsock on Ethernet networks and VIPL on VIA networks. Since the MPI specification assumes that communication occurs among separate processes, the Brazos implementation modified the semantics of communication to occur among threads. This modification is important for two primary reasons. First, the use of threads more efficiently utilizes multiprocessor workstations. Second, it is possible to reduce communication operations to simple memory copies when communicating threads are co-resident (i.e., they both reside on the same node) [48].
2.3 The Virtual Interface Architecture

The Virtual Interface Architecture [11] represents a significant deviation from traditional operating system to network interfaces, placing more direct access to the network in the user space while attempting to provide the same protection as that provided by operating system-controlled protocol stacks. The VI Architecture provides hardware support for direct user access to a set of virtual network interfaces, typically without any need for kernel intervention.

The VI Architecture is designed to address three important problems associated with the high cost of network access:

- Low bandwidth – Network-related software overhead limits the usable bandwidth of the network. In many instances only a fraction of the possible network bandwidth can be utilized.
- Small message latency – Because processes in a distributed system must synchronize using network messages, high latency for typically small synchronization messages can greatly reduce overall performance.
- Processing requirements – The overhead of message processing can significantly reduce processing time that the CPU can dedicate to application code.

The VI Architecture is designed to reduce the amount of processing overhead of traditional network protocols by removing the necessity of a process taking a kernel trap
on every network call. Instead, consumer processes are provided a direct, protected interface to the network that does not require kernel operations to send or receive messages. Each such virtual interface is analogous to the socket endpoint of a traditional TCP connection: each VI is bi-directional and supports point-to-point data transfer. Support for these virtual interfaces is intended to be implemented in hardware on the network interface card (NIC). The network adapter performs the endpoint virtualization directly and performs the tasks of multiplexing, de-multiplexing, and data transfer scheduling normally performed by an operating system kernel and device driver.

Figure 1: Diagram of the Virtual Interface Architecture.

Figure 1 depicts the organization of the Virtual Interface Architecture. The VI Architecture is comprised of four basic components: Virtual Interfaces, Completion Queues, VI Providers, and VI Consumers. The VI Provider consists of the VI network
adapter and a Kernel Agent device driver. The VI Consumer is generally composed of an application program and an operating system communication facility such as MPI or sockets, although some "VI-aware" applications communicate directly with the VI Provider API. After connection setup by the Kernel Agent, all network actions occur without kernel intervention, resulting in significantly lower latencies than network protocols such as TCP/IP. Traps into kernel mode are only required for the creation and destruction of VI’s, VI connection setup and tear-down, interrupt processing, registration of system memory used by the VI NIC, and error handling. VI Consumers access the Kernel Agent using standard operating system mechanisms.

A VI consists of a Send Queue and a Receive Queue. VI Consumers post requests (Descriptors) on these queues to send or receive data. Descriptors contain all of the information that the VI Provider needs to process the request, including pointers to data buffers. VI Providers asynchronously process the posted Descriptors and mark them when completed. VI Consumers remove completed Descriptors from the Send and Receive Queues and reuse them for subsequent requests. Both the Send and Receive Queues have an associated “Doorbell” that is used to notify the VI network adapter that a new Descriptor has been posted to either the Send or Receive Queue. In hardware VI Architecture implementations, the Doorbell is directly implemented on the VI Network Adapter and no kernel intervention is required to perform this signaling. The Completion Queue allows the VI Consumer to combine the notification of Descriptor completions of multiple VI’s without requiring an interrupt or kernel call.
2.3.1 Connection Procedures

The VI Architecture provides a connection-oriented networking protocol similar to TCP. Programmers make operating system calls to the Kernel Agent to create a VI on the local system and to connect it to a VI on a remote system. Once a connection is established, the application send and receive requests are posted directly to the local VI. A process may open multiple VI's between itself and other processes, with each connection transferring information between the hosts. The VI architecture provides both reliable and unreliable delivery connections. The expense of setting up and tearing down VI's means that connections are typically made at the beginning of program execution.

2.3.2 Memory Registration

The VI architecture requires the VI Consumer to register all send and receive memory buffers with the VI Provider to eliminate the copying between kernel and user buffers. This copying typically accounts for a large portion of the overhead associated with traditional network protocol stacks. The registration process locks the appropriate pages in memory, allowing for direct DMA operations into user memory by the VI hardware without the possibility of an intervening page fault. After locking the buffer memory pages in physical memory, the virtual to physical mapping and an opaque handle for each memory region registered are provided to the VI Adapter. Memory registration allows the VI Consumer to reuse registered memory buffers and avoid duplicate locking and translation operations. Memory registration also takes page-locking overhead out of the performance-critical data transfer path. Since memory registration is a relatively
expensive VI event, it is usually performed once at the beginning of execution for each buffer region.

2.3.3 Data Transfer Modes

The VI Architecture provides two different modes of data transfer: traditional send/receive semantics, and direct reads from and writes to the memory of remote machines. Remote data reads and writes provide a mechanism for a process to send data to another node or retrieve data from another node, without any action on the part of the remote process (other than VI connection). The send/receive model of the VI Architecture follows the common approach to transferring data between two endpoints, except that all send and receive operations complete asynchronously. Data transfer completion can be discovered through one of two mechanisms: polling, in which a process continually checks the head of the Descriptor Queue for a completed message; or blocking, in which a user process is signaled that a completed message is available using an operating system synchronization object.
Chapter 3 Integrating Thread Migration and Checkpoints

In this chapter we discuss the design alternatives and decisions that have guided the prototype implementation. In addition, we discuss why the integration of thread migration and rollback recovery is an attractive and efficient design choice for addressing the reliability and resource management issues present in a cluster-based parallel computing environment.

3.1 Objectives

Before discussing the design space, it is necessary to state our objectives:

1. We required a checkpoint/recovery mechanism for multithreaded SDSM applications with low runtime overhead and fast recovery.

2. In order to enable adaptive use of available cluster resources and tolerate downtime due to maintenance, efficient mechanisms for the addition and/or removal of systems from a cluster while applications continue to execute are needed.

3. Single node failures are expected to be common in a cluster environment. To tolerate such failures, an efficient mechanism to recover from a single-node failure when replacement systems are not available is needed. In addition, such a mechanism may be used to recover a failed process on a cluster with less computing nodes.
4. To ease the burden required to adopt our techniques, it is important to minimize the requirements imposed on the programmer or programming environment to enable the above functionality.

These goals constrained the choice of mechanisms among the available design alternatives. In the following discussion, we present several design alternatives for meeting the design objectives, as well as the reasoning behind the implementation decisions actually adopted.

### 3.2 Design Alternatives

#### 3.2.1 Environment

A parallel application executing on a SDSM cluster consists of a single process executing on each system being used. The runtime system is responsible for managing the pool of shared memory and ensuring data consistency across the systems. The runtime system is also responsible for managing communication and synchronization. The work to be accomplished by the parallel application is partitioned according to the number of processors or threads. The application programmer or compiler usually performs this division of work. We next discuss the design alternatives for each of our objectives.
3.2.2 Checkpoint/Recovery Alternatives

A checkpoint and recovery mechanism is needed to support reliability in clusters. One option was to use an existing implementation of a multithreaded process checkpoint facility for Windows NT (e.g., [51]) and adapt it for the Brazos SDSM system. However, we observed that a Brazos SDSM process is usually substantially larger than the total shared memory footprint of the application. This is because the many data structures, threads, and synchronization objects used by the runtime system consume substantial resources. If a process checkpoint facility was used, all this extraneous information would be saved, resulting in unnecessary checkpoint overhead. We elected to only save the state of shared memory and threads of a user application, resulting in smaller checkpoints and thus lower overheads.

The second design decision relates to the nature of the checkpoint facility itself. Our initial plan was to implement a two-level checkpoint scheme using consistent and log-based checkpoints, similar to [12]. Two-level checkpoints work by saving the full state of the running application infrequently, and using a log-based mechanism to keep track of changes in the state of the process on an on-going basis between checkpoints. If logs can be maintained with little overhead, this scheme reduces the overhead of performing checkpoints, and allows recovery up to the point of failure. In order for a log-based recovery to work in a SDSM system, it is necessary to reapply memory modifications during recovery in an order that preserves the correct state of shared memory. Unlike the Lazy Release Consistency [26] memory model employed in the TreadMarks system used
by Costa et al., Brazos does not require the association of timestamps with modifications. When present, these timestamps provide a tool for determining the order in which memory modifications occurred and is used for replaying memory modifications during log-based recovery. However, the amount of time required for applying memory modifications from logs can be substantial, approaching the total execution time lost in certain cases [12]. The high recovery overhead associated with log-based checkpointing resulted in our decision to implement a consistent checkpoint facility while trying to minimize its overhead. Since reapplying logs is such a time consuming process, we can achieve similar results by performing rollback recovery and re-executing the application up to the point of failure. The resulting checkpoint implementation exhibits low execution time overhead and fast recovery times (see Chapter 5). In addition, our checkpoint and recovery implementation enables unique functionality when integrated with thread migration, as discussed next.

3.2.3 Adding and Removing Nodes

Objective number 2 implies the need for the means to add and remove nodes online while applications continue to execute. Online addition requires a mechanism for transferring the state of data and computation from the removed node to the remaining ones. This could be achieved using process migration. The process being removed is saved in a checkpoint, or encapsulated and sent directly to another node. The destination node receives the process and the application continues. Unfortunately, this technique is problematic, for two reasons. First, it can result in multiple SDSM processes running on
a single node, which can seriously degrade performance. Second, it requires a heavy-
weight process migration mechanism. A more promising approach that does not result in
multiple processes sharing a system requires the ability to repartition a parallel
application and redistribute the data residing at the node being removed. This
repartitioning is accomplished by stopping the running process, and then reallocating
computation for the new number of nodes (node addition works in the same way) and
then either starting new processes or removing existing ones. The data on nodes being
removed is recovered by sending it to the remaining nodes [45]. Unfortunately, this
method suffers from a considerable drawback: the need to repartition a running
application. In typical parallel applications written in C or FORTRAN, there is no
language support that allows the runtime system to repartition a running application.
Such support is needed because the runtime system is not aware of the details of the
application that are necessary to ensure correct partitioning. If a parallel loop needs to be
partitioned, it is necessary to know its iteration count in order to divide the work
correctly. Such information is not available to the runtime system without additional
compiler and/or language support that is exposed to the runtime system. Some parallel
programming environments (e.g., OpenMP [40]) allow users to write sequential programs
and specify the portions of an application that can be run in parallel. A compiler is then
responsible for performing the partitioning of the parallel regions. Programming
environments of this sort allow the runtime system to change the number of processors
executing a program block at distinct "adaptation" points in user applications, typically at
the entry to parallel loops [45]. We rejected this approach for two reasons. First, if a
long time elapses between a request to remove a node and the next "adaptation" point, it may be necessary to revert to a more costly approach like process migration, resulting in overloading systems with processes. Second, requiring programmers to use special environments limits the usefulness of the system. We therefore looked for an efficient means to remove a node without the need for repartitioning. In order to arrive at this goal, it was necessary to recognize that a process is essentially a container for certain classes of objects that represent the state of a running application. The components of a process that are important for our purposes are the state of memory and the state of computation. Unlike single-threaded SDSM systems, Brazos is multithreaded, so the state of computation can be further broken down into the state of individual application threads. This separation of computation from data enabled us to develop the following solution. Instead of migrating a process or repartitioning an application, we distribute the process' data and computation state (threads) across the remaining nodes. This required the development of an efficient thread migration facility. Our thread migration implementation is similar to the ones used by other SDSM systems (e.g., [24, 55]) for load-balancing and communication reduction. The most important distinction between our work and previous studies using thread migration is that we use it for supporting reliability and resource management.

3.2.4 Automatic Failure Recovery

An important type of node removal occurs when a single system in a cluster fails. When no replacement systems are available, it is necessary to recover the state of the dead
process on the remaining nodes using the checkpoint recovery mechanism. We realized that an important benefit of integrating thread migration with checkpoint recovery allows our system to recover the threads and data of a checkpoint in a distributed manner. This addresses system failure in a manner that does not result in overloading a system with multiple processes, as is the case with previous checkpoint/recovery mechanisms. In addition, the application does not need to be repartitioned for recovery to take place.

For node addition, the integration of checkpoint recovery and thread migration facilitates a simple solution. In order to add a node, a number of existing threads have to be migrated to the new node. This is done to benefit from the new computing resource, or to recover from a prior node removal (e.g., for maintenance purposes). Again, this technique does not require a special programming language or programmer intervention to repartition a running parallel application. If repartitioning were used, existing processes would have to stop and wait for the new process to be created before the application could continue. Using thread migration the new process’ startup and initialization latencies are mostly overlapped with existing process execution.

To summarize, integrating thread migration and checkpoint recovery in cluster-based parallel computing environments allows efficient recovery from failure, and removal or addition of computing resources without interrupting running parallel applications. This approach is more efficient than techniques that use process migration because it uses
lower-weight mechanisms for transferring process state. Further, integrating thread migration and checkpoint recovery does not result in the possible overloading of processes on nodes during removal, nor does it require a special programming language that allows repartitioning of applications. Our consistent checkpoint recovery facility supports the recovery of a checkpoint on multiple nodes, allowing the distribution of data and computation to other processes. The key to achieving this is integrated support for thread migration, which allows any thread in an application to move freely among the nodes in the cluster.
Chapter 4  Thread Migration

This chapter describes the thread migration implementation adopted for our runtime system. We also discuss issues that must be addressed when implementing an efficient thread migration mechanism within the context of an SDSM system. Although the specific solutions presented here were implemented on a Windows NT/2000 platform, many of the design decisions are applicable to other systems.

4.1  Windows NT Threads

A thread represents the state of a particular execution path in a program. It is comprised of the processor state (or context), and a stack. A thread in Windows NT is comprised of the processor's register set, a thread-specific stack, and a special memory region called Thread Local Storage (TLS) [44] intended to contain data instanced per thread, or thread-private. Besides other possible uses, the Thread Local Storage was the mechanism chosen by Microsoft to support a multithreaded standard C library on Windows NT. The requirement for thread-local storage arises because some functions in the standard C library maintain state across calls (e.g., strtok, asctime, gmtime) [44]. To migrate a thread from one process to another, a thread's stack, context, and TLS are packaged and sent to a process executing on a remote node. Upon receiving the thread migration message, the remote process copies the contents of the thread's stack into a local thread's
stack and injects the context and TLS of the remote thread into that of the local thread.

The local thread is then resumed and computation continues normally.

4.2 Migrating Thread Stacks and Contexts

This section describes the mechanisms that we use to discover and migrate a thread’s stack and context. Before discussing the specific implementation details, we first discuss correctness issues related to stacks.

Since stacks may include pointers that reference stack data, a mechanism must be in place to guarantee that such pointers have the same meaning on the new host (issues related to heap and shared memory coherence are addressed in Section 4.3). 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 that are required to load at the same virtual addresses for the system to function correctly.

Two solutions addressing the stack data pointer problem have been proposed [24, 55]. In the first, the destination host scans the received stack data and adjusts or re-maps any pointers encountered by adding the appropriate offset. The offset is the difference
between the location of the source and destination stacks. This solution has several potential problems:

- Stack data may be misaligned, making it difficult to identify the location of pointers. Since Intel x86-based systems do not require data alignment, this problem applies to our implementation.
- 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. In addition, it is impossible to distinguish registers containing pointers from others containing data values similar to stack pointer values.

These problems make the scanning or re-mapping of pointers unattractive in the general case. We chose 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 [24, 55]. 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, the system creates a number of threads equal to the total number of user threads executing on all nodes. During the application’s lifetime, the maximum number of threads executing user code is fixed, although the number of active vs. dormant threads on a single node
varies according to thread-to-node distribution and migration behavior. A potential problem with our approach is the memory and operating system overhead for each thread created. This problem is ameliorated by the methodology used by Windows NT to manage stacks, which is described next.

Windows NT uses the concepts of memory reservation and commitment in its virtual memory management system. When an application reserves memory using the appropriate API, that range of memory addresses is allocated from the process' virtual address space; however, the operating system only allocates physical memory (from main memory and/or the swap file) when the memory is committed. Upon thread creation, the operating system reserves a default 1MB region from the process' virtual address space for the associated stack, but only 2 pages (a total of 8KB by default on x86-based systems) of physical memory are initially committed (see Figure 2). 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), the memory overhead of pre-allocating threads is relatively low. In addition, since typical thread stack sizes are often much less than the default 1MB (typically less than 4KB based on our experience with a number of applications), the amount of wasted address space may be reduced by lowering the default maximum thread stack size to a more appropriate value. The default maximum stack size and the amount of committed stack space can be set using linker options.
Figure 2: Thread stacks in Intel x86-based systems running Windows NT.

Because Windows NT threads are managed by the operating system, a mechanism is required for discovering a thread's stack and context before migration can take place. The Win32 API provides several functions that may be used to manipulate thread state and virtual memory [44]. A thread's context may be acquired and set using the GetThreadContext and SetThreadContext functions, respectively. Unfortunately, there is no standard API for discovering information about a thread's stack. This was one of the difficult problems encountered while implementing thread migration on NT/2000. We discovered two methods that can be used to find a thread's stack location and size. The first requires finding the thread's current stack pointer, which is part of a thread's context, and using it as an argument to the VirtualQuery function, which provides
information on memory regions. We use it to determine the region of memory associated with the thread's stack. To discover the length of the stack, it is also necessary to find the end of the stack region. One of the major drawbacks of this method is that it is necessary to have a-priori knowledge of the total stack size in order to find the last address (or stack base in Figure 2). Although it is possible to assume the default 1MB stack allocation limit used by the operating system, this solution will not work for the general case.

The second method for discovering information about the stack relies on the poorly documented Thread Information Block (TIB) data structure, which is maintained by the operating system for each thread. The TIB is stored at the beginning of the segment pointed to by the (FS) segment register on Intel x86-based systems and is user accessible. The Thread Information Block contains information used for structured exception handling as well as pointers to the base and commit limit of the stack. Since stacks grow towards lower memory addresses in Intel x86 processors, the base of the stack is defined as the highest memory address or bottom of the stack. The stack commit limit points to the current top of the stack, which is not necessarily equal to the lowest possible address contained in the stack, because Windows NT might not have committed the complete stack region. This technique solves the problem of having to know the stack size since it directly exposes the location of the bottom of the stack. The stack size is simply computed as the difference between the stack pointer and the stack base address. In addition, this method eliminates the need to call the VirtualQuery function and replaces
it with three in-line assembly instructions that read the appropriate data from the TIB.

The VirtualQuery function consumed several microseconds of CPU time in our previous implementation [1], so removing it was a performance gain.

The next section presents the correctness issues that arise when threads are allowed to migrate and their respective solutions in our implementation.

4.3 Ensuring Correctness

There are several correctness issues that arise when threads are allowed to migrate: the effects of thread migration on shared memory coherence mechanisms, the management of static (linker allocated) or non-shared heap data during thread migration, interaction with global synchronization primitives, and TLS and file access problems. We describe these issues and their solutions in detail next.

4.3.1 Thread-Private Data

As described earlier, a thread's context and its automatic or stack variables are maintained throughout the migration process by copying the thread's stack and context to the destination node; however, no explicit actions are performed with respect to shared or static memory that may be accessed by a thread. To move this data, a method for identifying static variables or non-shared heap memory that are only accessed by the thread being migrated is needed. Although it is possible to construct a memory map for a Windows NT process [51] and to migrate any such data detected, it is both difficult and
time consuming to do so. To achieve fast thread migration, our approach requires that
the programmer ensure that no read/write static (linker allocated) or non-shared heap data
is used as thread-private data. Thread-private data is defined as those memory locations
that are only read from or written to by a single thread throughout the lifetime of a
parallel application. All read/write data private to a thread must be allocated from the
runtime system's shared memory pool. This solution offers some advantages and
disadvantages. Because the runtime system will automatically migrate private data
allocated in shared memory only when it is accessed by a thread, this rule ensures both
that private data can be accessed by the migrating thread on any node, and that only data
actually needed will be migrated. Aligning such structures on page boundaries reduces
the chances of other threads causing more communication through false sharing. A
disadvantage of this solution is that programmers have to be aware of this to write
programs that will work correctly when using thread migration. Further, this method
does not migrate data during the thread migration operation. Thus, threads will incur the
additional latency of bringing pages in from other nodes when their accesses fault on
such data. Overall, we chose to implement this approach because its disadvantages are
much less than the alternative of discovering and transferring all thread-private data on
each thread migration operation. In addition, because thread migration can be used to
reduce communication in SDSM systems by migrating a thread to the location of the data
rather than migrating the data to it, our design decisions that emphasized performance
avoided the extra overheads that would eliminate the benefits expected using such
techniques. In Section 4.5, we compare the latency of a thread migration operation to that of migrating a page from another node in the Brazos SDSM system.

4.3.2 Interaction With Consistency Models

Shared memory coherence issues increase in complexity if relaxed consistency models are employed. For example, Brazos implements a Multiple Writer [9] protocol and two relaxed memory consistency models: Release Consistency (RC) [16] and Scope Consistency (ScC) [22]. The Multiple Writer protocol allows multiple nodes to modify different portions of a virtual page concurrently, and only performs coherence or consistency actions at specific synchronization points. To improve the performance of thread migration in Brazos, the semantics of a thread migration operation do not include synchronization and related coherence actions. This exposes the system to possible correctness problems unless care is taken when threads are migrated [55], as will be explained next.
Figure 3 shows an example of a correctness violation that can occur as a result of thread migration. Consider a thread, which initially resides on node 1 and modifies a shared variable $x$. A correct program under RC or ScC does not require synchronization before the value of $x$ is later read if it is not shared with other threads. If the thread is subsequently migrated to node 2 before it reads the variable $x$, the resulting load operation could return either a valid or stale value depending upon whether an appropriate synchronization event occurred between the initial modification and the access following migration. The simplest solution to this problem requires that a global synchronization operation, such as a barrier, be performed before the read of $x$ on node 2 can take place. This solution is applicable to both release consistency and scope consistency. For scope consistency, it is also possible to solve this problem using lock
synchronization as shown in Figure 4 with the restriction that the same lock instance be used. For RC, any lock synchronization would suffice. Thread migration inside critical sections is not supported by the runtime system's implementations of coherence and distributed lock management; therefore, threads are not allowed to migrate while holding locks. This is accomplished by ignoring a migration request if a lock is being held. Although it is possible to include synchronization semantics when implementing thread migration and perform the necessary coherence actions to avoid correctness problems, we chose not to do that because of the resulting performance penalty. We could also avoid this problem by only allowing thread migration at global synchronization events, such as barriers. Our current approach allows thread migration to occur whenever most appropriate, and requires the programmer or runtime system to synchronize as necessary. Whenever the runtime system chooses to migrate threads, it does so during barrier operations after all coherence actions are performed. This ensures correctness when automatic migration is performed for cluster reconfiguration, load-balancing, or fault tolerance operations.
4.3.3 Interaction with Synchronization Primitives

When threads are migrated from one node to another, it is possible to introduce problems that affect the operation of barriers. For example, the multithreaded barrier implementation in Brazos works as follows. Unless a thread is the last thread on a node to arrive at the barrier, it will sleep waiting on a Windows synchronization event associated with the barrier. The last thread to arrive at the barrier is responsible for exchanging coherence information with other nodes and for receiving the barrier release message from the barrier manager. A key to the success of this method is the knowledge
of the total number of threads executing on a node. When threads are allowed to migrate, the number of threads on a node can vary dynamically. The problem introduced by this can be illustrated with the following example. Consider a node that initially has two threads running on it, thread 0 and thread 1. Suppose that thread 1 arrives at the barrier, notices that it is not the last thread expected to arrive at the barrier, and sleeps waiting on the barrier event. Suppose also that thread 0 is migrated before it arrives at the same barrier instance and after thread 1 has already put itself to sleep. Even though the migration process will decrement the number of local threads within a critical section, thread 1 will never wake up and the barrier will not execute correctly, resulting in a deadlock condition. It is important to note that this condition only occurs when users are allowed to migrate threads and is avoided when the system migrates threads because it migrates threads only after all threads have arrived at a barrier.

This deadlock can be avoided as follows. It is possible to detect whether another thread on the same node has arrived at the next barrier. If the total number of threads that have arrived at the barrier is (n-1), where n is the total number of threads on the node, requests to migrate a thread can be ignored. Otherwise, the total number of threads can be atomically modified while preventing other threads from entering the barrier. This technique solves the problem at the thread’s source node; however, a similar problem may occur at the receiving node. The receiving node’s problem is complicated by the fact that it is impossible for the sender to detect whether any threads have arrived at a barrier on the destination node without a very complex and expensive distributed
protocol. The solution to this problem relies on the destination detecting how many threads have entered a barrier when it receives a thread migration message. If that number is less than the total number of threads already on the node, it allows the thread migration to take place and atomically updates the total number of threads. This ensures that the barrier will execute properly. If all threads have already arrived at the barrier before the thread migration message is received, the destination thread sends a negative acknowledgement to the source node, resulting in the abortion of the migration operation. While a sending node is waiting for the thread migration acknowledgement, no other thread is allowed to enter a barrier in case a migration operation needs to be aborted. This does not result in a high performance penalty because the threads intending to enter the barrier will not be doing useful work. The only performance penalty imposed by this solution is the necessity of introducing an acknowledgement message, which is not otherwise required when thread migration is implemented using reliable networks.

4.3.4 TLS and File System Access

Thread migration can violate correctness when application threads use the Windows NT thread local storage (TLS) facilities, or access the file system across migration events. Thread local storage usage can relies on either static or dynamic allocation. In static allocation, TLS storage is allocated by the linker and managed by the operating system and the compiler. In dynamic TLS allocation, storage is allocated and de-allocated dynamically by the application program. When threads are allowed to migrate among nodes, our system has to ensure that the contents of a thread’s TLS will be kept consistent. Designing a solution to this problem, requires an understanding of the way in
which the system manages TLS as well as the Win32 APIs that manipulate that storage. Because static TLS management is not exposed to the runtime system, we limit our support and this discussion to dynamic TLS. The Win32 API provides three functions for manipulating TLS. The TlsAlloc function is used to allocate a new TLS storage entry. The system guarantees a minimum number of available TLS entries for each thread and stores this number in the TLS_MINIMUM_AVAILABLE constant defined in the WINNT.H header file. TLS may be thought of as an array of integers. When an entry is allocated, TlsAlloc returns the index of an available entry in the array. This index can then be used by the TlsSetValue and TlsGetValue functions to access the data. I describe the way by which we manage TLS storage during thread migration next.

It is necessary to discover the number of TLS entries that have been allocated and their indices prior to initiating the transfer of a thread's TLS storage during migration. This is accomplished by trapping calls to TlsAlloc into a Brazos runtime system wrapper function that calls the actual Win32 function. The wrapper function keeps track of the number and actual indices of the allocated TLS storage using a bitmap of size TLS_MINIMUM_AVAILABLE. When a thread is being migrated, the TLS values are read and included in the migration message. At the destination node, they are written back to the same TLS locations. We impose two restrictions on TLS usage in Brazos to enable this method to work. First, all nodes have to allocate the same number of storage entries. Second, all such allocation has to happen during initialization so that the indices may be stored in global read-only variables that are identical on all nodes. These two
requirements avoid the problem of having different indices for the same TLS entry on different nodes, as well as the problem of a thread having a number of variables on one node and another on the migrated-to node. Since migrating TLS consumes time, we recommend that programmers avoid using TLS and instead rely on stack or shared memory locations to store thread private data.

We have recently discovered that the Thread Information Block (TIB) contains a pointer to the TLS contents of each thread. This allows an alternative implementation in which TLS contents are automatically included in the migration message without incurring the overhead of intercepting TLS-related Win32 functions. This will result in a small latency because the TLS array will increase the size of the thread migration message by about 256 bytes. This overhead is not necessary if TLS is not being used, so we provide an option for users to disable TLS transfer using migration.

Another important problem with thread migration is ensuring that a thread’s view of a file does not change when it is moved from one node to another. An understanding of all the issues that need to be addressed is necessary for implementing a solution. First, a technique to capture the important information necessary to reopen the files at the destination node is needed. Second, the location of the file pointer has to be discovered and set at the destination node. Third, when files are opened at the destination node, the file handle returned would most likely not be equal to the handle that the thread will use to access the file once it is migrated. A mechanism to translate file handles to avoid this
problem is required. Finally, there is the issue of access modes and file sharing settings. Application programmers might not enable file sharing when they open or create files, resulting in the failure of attempts to open the same files on other nodes. The mechanism to address this problem is described next.

The Win32 calls that access files are intercepted by wrapper functions that save the parameters necessary to reopen the files after migration (i.e., the name of the file, its sharing mode, read/write mode etc.). In addition, the current file pointer values are determined using the `SetFilePointer` Win32 call. `SetFilePointer` alters the value of the file pointer and returns its previous value. We use a zero increment as an argument and read the current file pointer value. This information is then transmitted to the new node and used to reopen the appropriate files and to reset their file pointers. Because the sharing parameters used to open the file might conflict with thread migration operations, the wrapper function alters these setting as appropriate to avoid problems when the file is reopened at the destination node. File handles used by the migrated thread may be different than the handles created at the new node. We created a mechanism for mapping file handles from the handle used by the thread to the actual handle at the new node. In case of a conflict between a handle that was identical on both nodes, the current implementation reverts to the handle associated with the thread in question. The handle will first be translated to the new handle if such a translation exists, otherwise the handle passed to the function is used. 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, files open for writing are flushed prior to migration. Files that are open remain open on the source node, in case other threads require access to the same file. A similar mechanism is used for checkpoint and recovery, as described in Chapter 5. One of the drawbacks of this mechanism is the extra overhead of the file I/O wrapper functions. We are investigating other methods and tools that are less intrusive (e.g., Detours [21]). In addition, it is still possible to violate correctness using this scheme. For example, if a parallel application is written in a way that allows multiple threads on a node to append to a file, this sharing of the file will fail once one of the threads is moved to another node. This is primarily due to the fact that the file pointer is no longer implicitly incremented as appropriate by the file system (i.e., the NT file system is not distributed in nature).

4.4 Anatomy of Thread Migration

This section provides details on the methodology that we used to implement an efficient user-level thread migration primitive under Windows NT. Before discussing our solution, it is important to emphasize our goals. First, the implementation must be efficient, as reflected in many of the design decisions described in both this and the previous sections. Second, the implementation must be a general mechanism that can be used by both application programmers as well as the runtime system, providing the flexibility needed to enable a wide variety of potential applications of a thread migration facility. We present the details of the implementation in the remainder of this section.
The first design issue is concerned with implementing an interface that can be used by either application programmers or the runtime system to perform thread migration. We decided to implement an API in the Brazos library called \texttt{MigrateMe(NodeId)}\texttt{.} This function can be called from an application program or the Brazos runtime system to migrate the calling thread to the node specified in the \texttt{NodeId} argument. From the runtime system’s perspective, this decision implied that a call to the thread migration API has to occur in code that will be executed by user threads, as opposed to runtime system threads. Although a thread may invoke a function to migrate itself to another node, it is not possible for the same thread to perform the actual migration. To understand the reasons for this, it is necessary to discuss some of the details involved in a thread migration operation.
void ThreadCkpt(USHORT Tid, USHORT ToNode, BCThread *thread)
{
    HANDLE ThreadHandle;
    DWORD StackDataSize, MsgSize, ThisMsg, MyId, FileInfoSize;
    char *StackEnd;
    BOOL Done;
    CONTEXT ThreadContext;

    GETMYLOCALID(&MyId);

    ThreadHandle = gDsmThreadHandle[Tid];
    SuspendThread(ThreadHandle);
    // Tell GetThreadContext that you want everything
    ThreadContext->ContextFlags = CONTEXT_FULL;
    // Now get the context of the source thread
    GetThreadContext(ThreadHandle, ThreadContext);
    // Figure out where the stack is and copy it to a message
    StackEnd = (char*)gDsmThreadStackBase[Tid];
    StackDataSize = StackEnd - (char*)ThreadContext->Esp;
    FileInfoSize = FileInfoList[Tid].number * sizeof(FileInfoStruct);
    FileInfoSize += sizeof(DWORD); // added for the number of files
    // number of TLS entries and their indices and values
    TLSSize = gDsmTLSInfo.number * (2*sizeof(DWORD)) + sizeof(DWORD);

    MsgSize = StackDataSize + sizeof(CONTEXT) + FileInfoSize + TLSSize;
    // copy context to msg
    msg <-- ThreadContext
    // copy file structures and TLS values and send msg
    msg <-- Number of values of FileInfoStruct's and TLS
    // copy the stack
    msg <-- stack from SP to end
    // send the thread to the destination node
    MsgSend(ToNode, REQUEST, WAIT_FOR_ACK);

    return;
}

Figure 5: Thread Migration Pseudo Code.

Before migrating a thread, it is first necessary to get its context and stack contents and then copy those to another thread on the destination node (see Figure 5). The migrating thread has to be suspended at the source node just before it migrates in a manner that is similar to the way it is suspended at the destination node prior to receiving the migration
message. A crucial issue in this process is defining the legal places in a program at which a thread may be suspended. If a thread is suspended inside the operating system while using a synchronization or signaling mechanism, it will also be resumed inside a call to such a mechanism. Changing a thread's context and stack while it is suspended and then waking it up inside of a system call is not a viable solution, because it relies on the state of kernel manipulated objects. Therefore, it is necessary to suspend the migrating thread in user code and resume it in user code. This makes it very difficult, if not impossible, for a thread to migrate itself. The solution to this requires another thread to perform the migration and to suspend and resume the thread in user code. I will refer to this special thread as the Migration Agent. The MigrateMe function stores the thread id of the calling thread in a global variable, wakes up the Migration Agent, and causes the calling thread to spin on a local variable. The implementation of the API appears in Figure 6. A few observations should be made about the implementation. First, since there is only one Migration Agent, only one thread may be migrated at a time. This is why the calling thread first acquires a critical section. The same critical section protects the global variables used by the Migration Agent to know the identity of the thread to be migrated and the destination node. The Migration Agent is created during the runtime system initialization phase before user threads are started and immediately waits on a thread migration event (hThreadMigrationEvent in Figure 6). The migrating thread wakes up the agent by setting this event and then spins on a local variable in user space (notice the while loop in the figure). Threads are suspended and woken up inside of this while loop using the Win32 SuspendThread and ResumeThread functions, avoiding
the problems discussed earlier. Although it seems as though there is a small window of opportunity for the thread to be suspended and woken up inside the call to SetEvent Win32 function due to preemption by the operating system, this has not been a problem in practice.

```c
void MigrateMe(int NodeId) {
    DWORD globalid;

    if (NodeId == THIS_NODE)
        return;
    if (NodeId < 0 || NodeId >= TOTAL NODES)
        Error("MigrateMe: invalid node \%d\n", NodeId);
    GETMYGLOBALID(&globalid);
    if (HoldingLock[globalid] == TRUE) {
        printf("Thread \%d not migrated (holding lock)\n", globalid);
        return;
    }
    EnterCriticalSection(&csThreadMigration);
    dwThreadToMigrate = (short) globalid;
    shMigrateToNode = (short) NodeId;
    SetEvent(hThreadMigrationEvent);
    while (nThreadSpinLocks[globalid] == 0) (;;) 
    nThreadSpinLocks[globalid] = 0;
}
```

Figure 6: Implementation of the MigrateMe API.

Once the thread is suspended, its context is acquired and its stack region discovered using the techniques discussed in Section 4.2. The thread’s context and stack contents, in addition to any TLS and/or file information blocks, are then copied into a migration message that 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. In the Winsock UDP and non-reliable VIA
implementations, the destination node explicitly acknowledges that the migration message was successfully received. When user thread migration is enabled, an acknowledgement message is also required in the reliable VIA implementation to ensure correctness as discussed in Section 4.3.3. At the destination node, the corresponding thread continues execution from the same point in the program at which it was suspended. The thread on the original node remains indefinitely suspended unless the same thread migrates back. This process is depicted in Figure 7.

Figure 7: Anatomy of a Thread Migration Operation.
4.5 Performance Analysis

Unless otherwise noted, the performance measurements presented throughout this dissertation were gathered on a network of Compaq ProLiant 6400R Servers running Windows NT 4.0 (Service Pack 5). Each system contains four 500MHz Pentium III Xeon processors with 512KB L2 caches. The systems are equipped with two PCI network interface cards: Fast Ethernet, and GigaNet GNN1000 in 64-bit slots. Each node also contains 512MB of main memory and a Wide Ultra2 SCSI disk controller and drive.

```c
if ((NPROCS > 1) && (MYNODE==0)){
    home = MYNODE;
    START_CLOCK(Start);
    for (count = 0; count < 1000; count ++) {
        MigrateMe(MYNODE+1);
        MigrateMe(home);
    }
    END_CLOCK(Start, Total);
}
```

Figure 8: Thread migration performance benchmark.

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 as shown in Figure 8, which shows an example of how the API can be used in application programs. 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 for
the UDP version\textsuperscript{2}. The acknowledgement was not required for the reliable VIA measurements. 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 Figure 9.

<table>
<thead>
<tr>
<th>Win32 Function (No. of calls)</th>
<th>Cost per call</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetThreadContext (1)</td>
<td>15.6 µsec</td>
</tr>
<tr>
<td>SetThreadContext (1)</td>
<td>15.4 µsec</td>
</tr>
<tr>
<td>SuspendThread (1)</td>
<td>5.1 µsec</td>
</tr>
<tr>
<td>ResumeThread (1)</td>
<td>5.6 µsec</td>
</tr>
</tbody>
</table>

Figure 9: The Cost of Win32 Calls Used in Thread Migration.

\textsuperscript{2} There were no retries or dropped messages observed during the performance measurements presented here.
The average thread migration latency using 4KB stacks was 1.040 ms for UDP on Fast Ethernet (Figure 10) and 240.4 μsec using GigaNet VIA (Figure 11). This latency is comprised of the Windows NT overheads to acquire the thread’s context and stack, as well as all synchronization and network operations. Communication accounts for the largest portion of the migration time, although overheads such as synchronization time were difficult to measure. Since the size of a thread’s context and the Windows NT overheads involved are fixed, the figures also show the effect of varying the stack size on thread migration latency for both UDP on Fast Ethernet and VIA. As shown in the figures, the thread migration latency increases with larger stack sizes as expected.
Figure 11: Thread Migration Latency vs. Stack Size for VIA.

The cost of a thread migration in Brazos is less than the cost of fetching a single page for both Winsock and VIA (refer back to Table 1), assuming a thread stack size of 4KB. The reason why this is important is that thread migration can also be used to reduce communication in SDSM systems [24, 55]. If the cost of thread migration is low, there will be more opportunities to improve performance by reducing data communication.

The latency of the Brazos thread migration mechanism is the lowest reported to date for SDSM systems. It is an order of magnitude better than that reported for the Millipede Windows NT-based system (70ms), although their performance data was measured on Pentium-based systems using Fast Ethernet. Our thread migration latency is also lower
than that reported by other implementations, using different hardware and operating system platforms (see Section 8.1).
Chapter 5  The Brazos Checkpoint and Recovery Facility

Clusters of multiprocessors are more susceptible to failure than single systems. In a cluster system, the probability of failure is directly proportional to the number of nodes. This makes the ability to tolerate failures important in cluster-based parallel computing environments. Further, the need for fault tolerance is amplified by the fact that applications that benefit from large-scale parallel processing systems are typically long-running ones, exposing them to a larger window of failure opportunities. 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 recovery is an effective mechanism for this situation. Since maintenance and upgrade functions typically require the interruption of service and are more frequent in a cluster environment, fault tolerance support allows processors or systems to be shutdown and restarted, thus interrupting rather than terminating running applications. In the remainder of this chapter, we describe the implementation of the Brazos checkpoint and recovery mechanism.

5.1  Checkpoint and Recovery in Brazos

A checkpoint saves a subset of the state of a running process. This subset includes all data necessary to recreate or restart the process from the point of its last checkpoint. A typical process checkpoint includes the stacks and contexts of all running threads, the
contents of memory (the heap and static data areas), and any operating system objects owned by the process, such as open file handles and synchronization objects. For processes participating in network communication, it is also necessary to save the state of open sockets and network connections. Checkpoint schemes generally fall into two categories: consistent and log-based checkpoints. Consistent checkpoints save the state of a process at fixed points and typically include all of the state needed to recover or restart a process. Log-based checkpoint and recovery mechanisms rely on maintaining a log of transactions or events that can be replayed at recovery time to restore the state of a process. The biggest advantage of consistent checkpoints is that they are usually compact in size, because only the state necessary for recovery is saved. However, consistent checkpoints, by definition, occur at discrete intervals during a program's execution. This makes it impossible to completely recover the state of a failed process. For this reason, recovery based on consistent checkpoints is called "rollback-recovery". To address this problem, log-based checkpoints allow recovery up to the point of failure by replaying events saved in a log. Log-based schemes can be expensive in terms of space requirements and execution time overhead, so many systems implement a combined rollback and log-based recovery mechanism to combine the benefits of the two methods.

The time needed to create a checkpoint is a performance issue because of the potential for a large amount of process-specific information. Large checkpoints consume large amounts of disk storage and take significant time to write to disk. For a checkpoint 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. I will discuss each of the components of a checkpoint and how they are created and recovered in the remainder of this chapter. However, before discussing these details, it is necessary to review the mechanics of starting a Brazos parallel application and how they affect the choice of checkpoint mechanism.

Each instance of a Brazos parallel application requires the creation of a configuration file. The configuration file contains information about the executable program, the names of the 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 (or “daemon” in Unix terminology) responsible for starting Brazos processes on the local node. The first node listed in the configuration file starts execution by sending process start requests to other nodes. All nodes synchronize at a startup barrier after all runtime system structures are initialized and before the user application may proceed. 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. User programs typically allocate shared memory and initialize data structures at the beginning of execution. The application then proceeds until completion. Several characteristics of the Brazos runtime system facilitate checkpoint creation:
• The initialization of Brazos is identical for any given configuration file. Using the same configuration file, all runtime system-specific objects such as system threads, socket or virtual interfaces, internal coherence data structures, and operating system synchronization objects can be recreated during recovery. This characteristic also implies that it is only necessary to checkpoint application data and threads to recover a failed Brazos process. Details about the configuration of the cluster are not saved.

• Since checkpoints can be independent from the Brazos runtime system initialization process, a different configuration can be used to recover the distributed application. This proves to be a valuable feature, allowing a Brazos application to be recovered on a cluster that is different in composition from the original one.

Section 5.2 presents more details on the implementation of checkpoint and recovery in Brazos. Section 5.3 describes the programming interface modifications necessary for utilizing the checkpoint facility. Finally, Section 5.4 presents performance measurements that show that our implementation exhibits both low execution time overhead and fast recovery times.

5.2 Implementation Details

There are several issues encountered when implementing a checkpoint and recovery facility in a SDSM system. The next sections describe how we addressed the following
issues: saving data and thread state, checkpoint initiation, and checkpoint file management.

5.2.1 Saving Data and Thread States

One of the main challenges of creating data checkpoints in a SDSM system is deciding the point at which shared memory pages and their runtime system management structures as well as application threads are in a stable state. This is primarily a function of the coherence activity that may be ongoing in the system. To guarantee that all pages are consistent and that no coherence actions are pending, checkpoints are created at global synchronization events (e.g., barriers). In addition, checkpoints are created while all user threads are waiting at the barrier and after all barrier-related coherence actions have been completed at all nodes. More information about the mechanism used to initiate checkpoint and recovery actions will be provided in Section 5.2.2.

<table>
<thead>
<tr>
<th>State</th>
<th>Empty</th>
<th>Valid</th>
<th>Twin</th>
<th>Diff</th>
<th>Protection</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>True</td>
<td>False</td>
<td>None</td>
<td>None</td>
<td>NO_ACCESS</td>
<td>Page never accessed.</td>
</tr>
<tr>
<td>2</td>
<td>False</td>
<td>False</td>
<td>None</td>
<td>None</td>
<td>NO_ACCESS</td>
<td>Page is invalid.</td>
</tr>
<tr>
<td>3</td>
<td>False</td>
<td>False</td>
<td>None</td>
<td>Yes</td>
<td>NO_ACCESS</td>
<td>Page invalidated, but was dirty at this node.</td>
</tr>
<tr>
<td>4</td>
<td>False</td>
<td>True</td>
<td>None</td>
<td>None</td>
<td>READONLY</td>
<td>Page being read.</td>
</tr>
<tr>
<td>5</td>
<td>False</td>
<td>True</td>
<td>None</td>
<td>Yes</td>
<td>READONLY</td>
<td>Page is being read and was dirty before last barrier.</td>
</tr>
<tr>
<td>6</td>
<td>False</td>
<td>True</td>
<td>Yes</td>
<td>None</td>
<td>READWRITE</td>
<td>Page being written to.</td>
</tr>
<tr>
<td>7</td>
<td>False</td>
<td>True</td>
<td>Yes</td>
<td>Yes</td>
<td>READWRITE</td>
<td>Page being written and was dirty before last barrier.</td>
</tr>
<tr>
<td>8</td>
<td>False</td>
<td>True</td>
<td>None</td>
<td>None</td>
<td>READWRITE</td>
<td>State on node 0 before other node(s) request a page</td>
</tr>
</tbody>
</table>

Table 2: Shared Memory Page States in Brazos.
A checkpoint operation stores all data needed to perform a recovery operation, including all shared memory pages, the result of any modifications to these pages, and the related runtime system data structures. Table 2 summarizes the possible states in which a shared memory page in Brazos may be. The checkpoint file saves this page state structure, the page data, and any twins or diffs for any page in states 3 through 7. On node 0, pages are initialized with Empty=0 and are in READWRITE mode (state 8 in Table 2). Saving all such pages can be wasteful because they might not contain any useful data. To solve this problem, all pages at node 0 are also guarded during initialization. Page guarding in Windows NT allows users to trap the first access to a page. When an access is detected to such a page, an accessed flag is set by the runtime system and the operating system automatically removes the guard. This mechanism allows the checkpoint facility to only save pages that have been accessed, reducing the size of the checkpoint file. In addition to the page state structure, the runtime system maintains two bitmaps per page: the copyset and the invset. The copyset keeps track of the number of nodes that have a copy of a page and is used by the runtime system’s coherence mechanisms to send updates to nodes that have a copy of a page. The invset keeps track of the nodes that have performed modifications to a page and is used to determine the nodes that have to be contacted when a node faults on a page, requiring its data to be brought up-to-date. The copysets and the invsets are also saved in the checkpoint file to maintain the coherence information necessary for the system to proceed correctly upon recovery. The runtime system allocates a pool of shared memory pages during initialization. To avoid saving shared memory pages that have not been allocated
by user programs, the checkpoint facility also requires the addition of an *allocated* flag for each page. This flag is checked before a page is saved to the checkpoint file. Finally, we modified the structure of page status blocks to decrease the amount of redundant information saved in a checkpoint by only copying the portion of the page status blocks that cannot be recreated during recovery. The checkpoint creation process is summarized in Figure 12.

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. It is also necessary to ensure that barrier structures are allocated at the same virtual addresses both before and after application recovery. This ensures that the barrier structures are updated before threads are resumed during recovery. Ignoring this step could result in either a deadlock (because the state of the barrier does not reflect the correct number of suspended threads), or an access violation (if the barriers do not reside at the same virtual addresses before and after recovery).
void CkptProcess() {
    // Open the checkpoint file
    ckptfile = CreateFile(ckptfilename, GENERIC_WRITE, 0, NULL,
                                CREATE_ALWAYS, FILE_FLAG_SEQUENTIAL_SCAN, NULL);

    // Count the number of pages/twins/diffs for directory
    for (all shared memory pages) {
        if (page is allocated, used and (valid or has a diff)) {
            NumPages++;
            if (page has a twin) NumTwins++;
            if (page has a diff) NumDiffs++;
        }
    }

    // Create the ckpt directory and copy it to ckptfile
    <include NumPages, NumTwins, NumDiffs, NumThreads, Node Id,
      ckpt instance number, file info blocks >

    // Write invsets and copysets arrays
    ckptfile <-- Invsets and Copysets
    
    for (all shared memory pages) {
        if (page is allocated, used, and (valid or with diff)) {
            ckptfile <-- page status block
            ckptfile <-- page data
            if (page has a twin) ckptfile <-- page twin
            if (page has a diff) ckptfile <-- diff
        }
    }

    // Copy file structures
    ckptfile <-- file information blocks
    DSM_Print("Number of threads = %d\n", ckptdirectory.NumThreads);

    // Copy the user threads (DSM threads not needed for recovery)
    for (all threads on this node) {
        ckptfile <-- thread information block (Id, stacksize, etc.)
        <Perform Thread Migration actions>
        ckptfile <-- thread stack/context
    }
    CloseFile(ckptfile);
}

Figure 12: Checkpoint Creation Pseudo Code.

In addition to shared memory data and states, the checkpoint includes application thread contexts and stacks, and file status structure for files that are open during the checkpoint creation operation. The mechanisms used to save thread and file information are identical to those used for thread migration. The only significant difference is that
information is saved to disk, rather than sent to another node. The details of this process are presented in Chapter 4. Special runtime system data structures keep track of all the information needed to reopen files upon recovery (i.e., the arguments to the corresponding CreateFile Win32 call), including the current file pointer values. The contents of these structures are saved in the checkpoint after the shared memory contents. During recovery, the files are reopened and the file pointers are set to the appropriate values using standard Win32 calls.

During recovery, which also occurs at barriers, each node opens its checkpoint file and applies its contents to the shared memory pages, setting page protections and copyset/invset values as necessary. Thread contexts are set and stacks are copied before application threads are woken up and allowed to exit from the recovery barrier. At that point, the state of the applications is equivalent to its state at checkpoint creation.

5.2.2 The Checkpoint Agent and Checkpoint Initiation

Checkpoint creation presents a dilemma similar to the one encountered in implementing thread migration. An application thread cannot capture its own state while performing the checkpoint creation or recovery operation. The solution to this problem is to create a Checkpoint Agent thread that performs checkpoint and recovery operations.
Figure 13: Checkpoint Agent Thread Procedure.

The Checkpoint Agent is created during runtime system initialization and immediately executes an infinite while loop. Figure 13 shows the basic structure of the Checkpoint Agent’s behavior. The thread’s priority is set to `TIME_CRITICAL` to ensure that it has exclusive access to the CPU during checkpoint creation. This is important because one of the application threads is usually spinning on a variable when checkpoint generation is initiated. This spinning allows the thread in question to be suspended in user code rather than in operating system synchronization calls (see the discussion in Section 4.4).
Our checkpoint facility supports two methods of initiating checkpoint creation. Programmers can call a Brazos checkpoint barrier, instructing the system to perform a checkpoint. In addition, each program must include a call to a recover barrier after all initialization operations are complete to enable recovery. Since most users will be more concerned with the amount of time potentially lost due to failure, Brazos also supports an automatic checkpoint interval that can be specified through the configuration file. If the time since the previous checkpoint exceeds the specified interval, the runtime system initiates a checkpoint operation at the next barrier and resets the timer. The Brazos user interface can be enhanced to send a message to the application forcing the system to perform a checkpoint at the next barrier. This feature may be useful for planned shutdowns or when it is desirable to avoid regular checkpoint overhead. Regardless of the mechanism used, the checkpoint is initiated by the last thread to arrive at the barrier manager node by setting the synchronization event on which the Checkpoint Agent is waiting (SystemLocks->m_hCheckpointProcessEvent in Figure 13). Other nodes are signaled to perform a checkpoint by a special checkpoint barrier release message. Upon signaling the Checkpoint Agent, the same thread spins on a system variable awaiting suspension during checkpoint creation and is later resumed once the checkpoint creation is completed. When the Checkpoint agent wakes up, it determines whether it is to perform a checkpoint or recovery operation using the contents of a global variable. Then the appropriate checkpoint or recovery procedures are initiated. If multiple recoveries are necessary, as is the case with node removal due to failure (see Section 6.1), the Checkpoint Agent also calls the DSM_RecoverOtherProcess procedure.
After copying shared memory state, all threads are suspended by the Checkpoint Agent, and their states (contexts and stacks) are saved to the checkpoint file. To aid the process of recovery, the checkpoint includes a directory of contents. This directory contains the number of pages, files, and threads that need to be recovered and can also be used for checking the integrity of the checkpoint file, whose size can be determined given the knowledge of its contents. After checkpoint or recovery is completed, all user threads are resumed and computation continues normally.

The time consumed in creating checkpoints is a function of the amount of shared memory in use at each node in the system, but is considerably smaller than the size of the running process (refer to Section 5.4 for performance measurements).

![Checkpoint File Management](image)

Figure 14: Checkpoint File Management.
5.2.3 Checkpoint File Management

There are two important performance decisions related to checkpoint file management. First, it is important to perform the necessary disk I/O in the shortest time possible while maintaining the integrity of the files for possible recovery. Second, it is desirable to overlap as much of the I/O operations as possible with useful computation to reduce the runtime overhead of checkpoint creation. We addressed the first issue as follows. Each node initially creates its checkpoint file locally to maximize filesystem throughput. The best filesystem write throughput was achieved using the SEQUENTIAL SCAN file access mode while allowing the NT filesystem buffering mechanisms to remain active, although previous work has indicated that good write throughput can also be achieved using hardware write caches on Windows NT\(^3\) [42]. Since node failures can render access to local files impossible, it is also necessary to copy the checkpoint files to remote file servers that are accessible by all nodes. To reduce checkpoint creation overheads, all of these steps have to be accomplished while minimizing the amount of time during which user threads remain suspended. This is accomplished by having the Checkpoint Agent close the checkpoint file after it is created locally. To minimize overhead, no flushing of operating system file buffers is performed before application threads are resumed (refer to Figure 14). The responsibility for I/O completion therefore lies with the Windows NT filesystem. We chose this scheme because it allows the overlap of computation with I/O operations, resulting in up to a 70% reduction in checkpoint

\(^3\) The hard disk drives on our experimental platform did not support hardware write caching.
overhead compared to our previous implementation, which is described in [1]. Once the local checkpoint is closed and application threads are resumed, the Checkpoint Agent initiates a **CopyFile** Win32 call to copy the checkpoint file to a network filesystem before accepting any new requests for checkpoints. To overcome the possibility of losing a checkpoint due to failure during I/O operations, the runtime system retains two checkpoint files and alternates between them as new checkpoints are created. In case a checkpoint file is found to be corrupt or incomplete during recovery, the system can revert to the older checkpoint. In the unlikely event that this recovery would be problematic, the configuration file allows the user to require file I/O completion prior to exiting the checkpoint barrier.

### 5.3 Programmer Interface

There are a small number of checkpoint-related issues that must 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 to provide thread migration as described in Section 4.3). Variables that are declared and initialized in this manner will be reinitialized upon recovery and execution will proceed correctly. Figure 15 shows the outline of an application that is written to utilize the Brazos checkpoint/recovery facility.
Either the user or the runtime system may initiate a checkpoint. Programmers are able to call special barriers that instruct the system to perform either a checkpoint or recovery operation. 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, the Checkpoint Agent thread initiates the checkpoint. Application threads resume once the checkpoint is completed.

```c
void UserMain() { // all user threads start execution here

    DWORD MyId, LocalId;

    GETMYGLOBALID(&MyId); // global thread id
    GETMYLOCALID(&LocalId); // thread id for this node only

    if (MyId == 0) {
        <G_MALLOC shared memory and initialize it>
    }
    BARRIER(0);
    if (LocalId == 0) {
        <allocate static/heap read-only data>
    }
    RECOVER(0); // init. done, recover here if needed!

    Slave(MyId,LocalId); //Initiate parallel computation

    BARRIER(0);

    if (MyId == 0) {
        <output results and/or perform cleanup>
    }
}
```

Figure 15: Outline of an Application Utilizing Checkpoint/Recovery.
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 using the Brazos Graphical User Interface (see Figure 16), or on the command line. Using the interval method, when the time since the previous checkpoint exceeds the specified interval, the runtime system initiates a checkpoint. An explicit checkpoint is initiated when all application threads arrive at a CHECKPOINT barrier. All Brazos programs must explicitly include at least one RECOVER barrier that is usually placed after all initialization activity is completed by the application (see Figure 15). For programs that do not use barriers, our current implementation requires inserting additional synchronization.
Brazos provides two types of recovery mechanisms: automatic recovery from single-node failures on the remaining nodes (see Section 6.2), or manual recovery on the same cluster or a different one with the same number of nodes. 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.

The performance of the Brazos checkpoint and recovery facility is presented next.

### 5.4 Checkpoint Performance

<table>
<thead>
<tr>
<th></th>
<th>SOR</th>
<th>Water</th>
<th>3DFFT</th>
<th>Barnes-Hut</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Size:</td>
<td>4000x4000</td>
<td>4096 Mol</td>
<td>$2^6 \times 2^7 \times 2^6$</td>
<td>32K Bodies</td>
</tr>
<tr>
<td>Iterations:</td>
<td>10000</td>
<td>80</td>
<td>2500</td>
<td>750</td>
</tr>
<tr>
<td>Processors:</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Nodes:</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Threads/Processor</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Execution Time (sec.)</td>
<td>1400.19</td>
<td>1233.12</td>
<td>1110.98</td>
<td>1146.86</td>
</tr>
</tbody>
</table>

Table 3: Benchmark Applications and Problem Sizes.

Four parallel applications were used to measure the overhead of the checkpoint facility: SOR (successive over-relaxation) using 4000x4000 matrices for 10000 iterations, Water, a molecular dynamics simulator from the SPLASH benchmark suite [46] using 4096 molecules for 80 iterations, 3DFFT from the NAS benchmark suite [5] with a $2^6 \times 2^7 \times 2^6$ three-dimensional matrix for 2500 iterations, and Barnes-Hut, a simulator for the interaction of planetary objects due to gravitational forces also from the SPLASH suite using 32768 bodies for 750 iterations. All of the measurements were performed on a 4-node, 16-processor cluster interconnected using VIA. Checkpoint files were both written to local disks and copied to a network file server over Fast Ethernet. Table 3 summarizes the applications and problem sizes used. To measure the overhead of the checkpoint
facility, we used the system-initiated checkpoint mode while varying the checkpoint interval from two to sixteen minutes. For each application, Figure 17 shows the execution time overhead of generating checkpoints at the respective intervals compared with the Base execution time with checkpoints disabled.

These applications were chosen for the following reasons. First, we think that they are representative of typical scientific parallel applications that can be run on shared memory systems. Second, they include both regular and irregular data access patterns, which affects the amount of coherence data generated by the runtime system and stored in checkpoints. Third, they include shared memory footprints of varying sizes to study the effect of that on checkpoint overheads. Fourth, all four applications are iteration-based, so it was possible to increase their execution times arbitrarily to capture at least a single checkpoint at 16-minute intervals.
As you can see in Figure 17, the overhead of generating checkpoints was 8.6% or less of the total execution time for all applications using two-minute checkpoint intervals. This overhead is lower at lower checkpoint frequencies as one would expect. Table 4 provides additional parameters that are useful for the performance evaluation. First, it shows that the checkpoint facility is successful at minimizing the amount of data saved, compared to the size of the running process. The checkpoint size relative to the process size ranged from 3.35% to 24.66% across all four applications. Table 4 shows the total recovery time for each of the four applications, both with and without runtime system and application
initialization. The recovery times, including initialization, using the checkpoint files maintained on network disks ranged from 7.64 to 43.33 seconds across all four applications. Since we use a consistent checkpoint scheme that does not maintain logs of memory transactions, the applications will spend some time (half of the checkpoint interval on average) to reach the state that they were in before failure occurred. Even if log-based recovery were supported, additional time would be required to apply the logs. Previous work has shown that recovering logs can also be time consuming (refer to Chapter 8). Table 4 also shows other statistics on the average number of pages, twins, and diffs included in a checkpoint.

<table>
<thead>
<tr>
<th></th>
<th>SOR</th>
<th>Water</th>
<th>3DFFT</th>
<th>Barnes-Hut</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recovery from Network Disks</td>
<td>33.83 sec.</td>
<td>14.39 sec.</td>
<td>18.69 sec.</td>
<td>3.21 sec.</td>
</tr>
<tr>
<td>Total Recovery Time</td>
<td>43.33 sec.</td>
<td>17.21 sec.</td>
<td>22.24 sec.</td>
<td>7.64 sec.</td>
</tr>
<tr>
<td>Bandwidth seen during recovery</td>
<td>3.65 MB/s</td>
<td>3.42 MB/s</td>
<td>4.84 MB/s</td>
<td>3.87 MB/s</td>
</tr>
<tr>
<td>Average Checkpoint Size</td>
<td>76.95 MB</td>
<td>32.68 MB</td>
<td>34.77 MB</td>
<td>6.64 MB</td>
</tr>
<tr>
<td>Average Pages per Checkpoint</td>
<td>13703</td>
<td>5262</td>
<td>7672</td>
<td>1241</td>
</tr>
<tr>
<td>Average Twins per Checkpoint</td>
<td>5861</td>
<td>2019</td>
<td>1295</td>
<td>11</td>
</tr>
<tr>
<td>Average Diffs per Checkpoint</td>
<td>0</td>
<td>505</td>
<td>3</td>
<td>835</td>
</tr>
<tr>
<td>Shared Memory Total Size</td>
<td>122 MB</td>
<td>45 MB</td>
<td>88 MB</td>
<td>12 MB</td>
</tr>
<tr>
<td>Checkpoint vs. Shared Memory</td>
<td>252 %</td>
<td>292 %</td>
<td>158 %</td>
<td>221 %</td>
</tr>
<tr>
<td>Process Size</td>
<td>312 MB</td>
<td>248 MB</td>
<td>226 MB</td>
<td>198 MB</td>
</tr>
<tr>
<td>Checkpoint vs. Process Size</td>
<td>24.66 %</td>
<td>13.18 %</td>
<td>15.38 %</td>
<td>3.35 %</td>
</tr>
</tbody>
</table>

Table 4: Checkpoint and Recovery Statistics.

Chapter 6 Node Removal

In this chapter, we describe three mechanisms that are facilitated by the integration of thread migration and checkpoint/recovery: node removal using checkpoint and recovery,
automatic single node failure detection and recovery, and online node removal. Node removal using checkpoint/recovery is a useful mechanism for restarting a failed Brazos parallel application on a cluster with a smaller number of nodes. This mechanism is important because the most common failure in cluster-based systems is likely to involve a single node. In addition, the availability of an additional system to recover the complete parallel application while the failed node is being repaired is not guaranteed. To complement node removal using checkpoint/recovery, we implemented a simple automatic failure detection mechanism that allows the recovery from single node failures without any user intervention on the remaining nodes. Finally, the ability to reconfigure parallel applications to use fewer resources is a desirable feature in a cluster-based parallel computing environment. This is achieved by an online node removal mechanism that does not require checkpoint creation or recovery. We describe the implementations and performance of these mechanisms next.

6.1 Node Removal Using Checkpoint/Recovery

Removing a node (and its associated process) requires the ability to save both its memory image and the state of any computing threads present on the node. Saving the contents of memory is conceptually simple, because it only requires data to be copied from one node to another. This data copying is facilitated by the Brazos runtime system since it includes information about the state of all shared memory pages. The checkpoint support in Brazos further simplifies the transfer of memory contents from one node to another. The
more difficult aspect of node removal is the saving and restoration of the state of 
computation, especially in a multi-threaded environment. While it is relatively simple to 
copy a process' state and stack (which includes the threads associated with the process) to 
another node, the ability of other nodes to accept those threads is best facilitated by the 
thread migration infrastructure. This is especially apparent when the correctness issues 
associated with thread migration are considered (see Section 4.3). The alternative to 
thread migration would require migrating the process and re-creating it on one of the 
existing nodes [45]. This solution has two significant disadvantages relative to our thread 
migration approach. First, process migration is more expensive, because the entire 
process footprint has to be copied from one system to another. Second, process migration 
may require placing two active copies of the same distributed application on a single 
node, which will likely result in unacceptable performance for large applications due to 
swapping. If process is terminated after transferring its data, as an alternative to process 
migration, then the application has to be repartitioned. This is necessary because the 
computation threads of the dead process cannot be adopted by the remaining nodes, 
making it impossible to proceed correctly with a repartitioning of work.

The remainder of this section describes how distributed applications may be stopped and 
restarted on a smaller number of nodes. This capability provides the core of the 
automatic recovery mechanism that allows the unattended recovery from single node 
failures.
6.1.1 Memory and Thread Recovery

To recover a checkpoint, both the user threads and shared memory contents in use at the time of the checkpoint must be restored. This recovery takes place on either a replacement node or on some subset of the remaining nodes. The support for thread migration in the Brazos system makes the recovery of threads a relatively simple process. The thread information stored in the checkpoint files allows recovered threads to be treated exactly like migrating threads. Recovering the memory contents of the removed node is somewhat more complicated. First, the runtime system modifies the copyset and invset bitmaps to remove the bits corresponding to the removed node, because the bitmaps that were saved in all the nodes' checkpoints reflected the existence of a node that is no longer available. Clearing the appropriate bits in these bitmaps avoids the problem of sending requests for pages to non-existent nodes. Additionally, the runtime system must identify pages that were only valid at the removed node, and recover those pages on at least one of the remaining nodes. This is accomplished by examining the local state of the page and comparing it against the state represented in the checkpoint. Pages that were valid only at the removed node are recovered on one of the remaining nodes. For pages that had diffs on the removed node, the runtime system applies those diffs to other copies of the page that reside on other nodes. Finally, pages that have twins on the removed node are copied along with their twins on one of the remaining nodes. The copysets and invsets are modified to reflect the new state of a page. Figure 18 shows the algorithm used for shared memory page recovery during node removal.
Thread recovery during node removal is simple. The threads of the removed node are distributed in a round-robin fashion across the remaining nodes. This is accomplished by essentially performing a thread migration using the removed node checkpoint file as the source of the threads. A more sophisticated thread distribution heuristic that uses load and system configuration information (possibly through querying an extended version of the Brazos remote access service) can be employed to optimize this operation. In such an implementation, each node will query its local Brazos service as well as those of the other nodes and determine the distribution of threads. The only disadvantage to this approach is the additional communication that needs to occur among nodes to arrive at an acceptable distribution. The current implementation does not require any such communication, but can result in overloading certain nodes with threads while leaving processors on other nodes idle.

Before the system can be recovered with a smaller number of nodes, a new Brazos configuration file that indicates all nodes participating in the computation must be created. In addition, a flag is set to signal that a removed node recovery needs to be performed. To simplify recovery, the system reassigns node identifiers so that the node with the highest numeric identifier is removed (i.e., the node corresponding to the highest order bit of the copysets and invsets). This optimization eliminates the need for complex manipulation of bitmaps upon recovery and thus does not introduce any overhead during the recovery process. Finally, synchronization issues related to barriers and distributed
lock management are eliminated by this scheme because all synchronization primitives are reinitialized during the recovery process with the correct number of nodes.
void DSM_RecoverOtherProcess(DWORD NodeId) {
    ckptfile = open checkpoint file (sequential scan mode)
    // ckptdirectory data recovered from file
    ReadFile(ckptfile, &ckptdirectory, sizeof(CkptDir), &BytesRead, NULL);
    skip invsets, copysets that are in checkpoint;
    // remove NodeId from invsets and copysets
    for (icount = 0; icount < SharedMem.m_chNumPages; icount++) {
        SharedMem.m_bpInvset[icount] &= ~(1<<NodeId);
        SharedMem.m_bCopyset[icount] &= ~(1<<NodeId);
    }
    // Recover the pages and their associated diffs/twins if any
    for (iCount = 0; iCount < (int)ckptdirectory.NumPages; iCount++) {
        ReadFile(ckptfile, &page, sizeof(BCRecoverPage), &BytesRead, NULL);
        if (page.m_pTwin != NULL) { // removed node's page is dirty!
            if (lpage->m_tcValid == 0) {
                if (NODE_ID == 0) { // get it yourself, first get page
                    lpage->m_tcValid = 1;
                    copy page,
                    // allocate a twin and get it
                    lpage->AllocateTwin();
                    copy twin;
                    lpage->SetProtection(PAGE_READWRITE);
                } else { // we didn't get it, but mark its new node
                    SharedMem.m_bpInvset[PageNumber] = 0x1;
                    SharedMem.m_bCopyset[PageNumber] = 0x1;
                    skip the page and twin;
                }
            } else { // get the page
                copy the page
                lpage->AllocateTwin();
                copy the twin;
            }
        } else if (page.m_tcValid == 0 && page.m_pDiffStruct != NULL) {
            if (lpage->m_tcValid == 0 && lpage->m_pDiffStruct != NULL) {
                skip page but apply diff to our page;
            } else { // skip page and diff;
                if (page.m_tcValid == 1 && page.m_pTwin == NULL) {
                    if ((SharedMem.m_bCopyset[PageNumber] != 0)||(lpage->m_tcValid==1)) { // skip this page:
                        else { // no other copy, make sure to adopt this page somewhere
                            if (NODE_ID == 0) {
                                if (lpage->m_tcValid == 1) {
                                    copy the page and apply its diffs (if any)
                                    lpage->m_tcValid = 1;
                                    lpage->SetProtection(PAGE_READONLY);
                                } else { //just make sure to get the diff if any
                                    skip page, but apply its diff (if any)
                                }
                            } else { // we didn't get it, but mark its new node
                                SharedMem.m_bCopyset[PageNumber] = 0x1;
                                skip the page;
                            }
                        } else { // page is invalid and doesn't have a diff, just skip it
                            skip the page;
                        }
                    }
                }
            }
        }
    }
}

Figure 18: Checkpoint Recovery for Node Removal.
The checkpoint recovery of the removed node is initiated as part of the normal recovery process. It is invoked by the Checkpoint Agent after the recovery of each node's checkpoint is completed at the RECOVER() barrier (refer back to Figure 13). The performance of node removal using checkpoints will be discussed within the context of automatic recovery.

```c
void DSM_HeartBeat() {
    while (1) {
        Sleep(HeartbeatInterval);
        HeartBeatMask = ALL_NODES;
        prepare heartbeat message;
        SendMsg( HeartBeatMask, REQUEST, FALSE, DSM);
        // now wait for the replies (timeout in 1 sec)
    Wait:
        if (WAIT_FOR_REPLY(1000ms timeout) == -1) {
            DSM_Print("Missing heartbeat replies mask=0x%x!!!\n", SilentNodes);
            if (SilentNodes != 0) {
                if (NumRetries++ > 2) { // find out which nodes failed
                    identify failed nodes to user
                    set recovery flag in configuration
                    rearrange configuration file for node removal
                    // ask service to restart app w/node removal recovery
                    CreateRemoteProcess(0);
                    exit;
                } else {
                    SendMsg->Send(SilentNodes, REQUEST, TRUE, DSM);
                }
            }
            goto Wait;
        }
    NumRetries = 0;
    } // while loop
}
```

Figure 19: Heart Beat Thread Procedure.
6.2 Automatic Recovery

The ability to remove a node facilitates the implementation of an automatic recovery mechanism for single node failures. To implement this mechanism, several additional capabilities are required. First, the runtime system must be able to detect node failures. Second, once the failed node is identified, the remaining nodes must be automatically shutdown and restarted from the last checkpoint.

The automatic recovery mechanism described here assumes that the network itself is reliable (i.e., failures occur in a system rather than in the network). This simplifies some of the issues needed to identify failures. In addition, the current implementation relies on a single node to perform the failure detection and to initiate the automatic recovery process. Alternative failure detection protocols that are more robust are discussed later in this section.

To minimize the number of messages required to detect failures, we employ multicast heartbeat messages when available, and revert to point-to-point messages otherwise (e.g., for VIA, which does not support multicast). A designated node in the system creates a thread (the Heartbeat Thread, see Figure 19) that is responsible for sending multicast heartbeat message at regular intervals (10 seconds by default, but users have the option of changing this value using the Brazos Graphical User Interface, see Figure 16). If all replies are not received within a specified timeout period, the heartbeat message is resent. In order to minimize the unnecessary interruption of live nodes, the heartbeat retries are
only sent to the non-responding node(s). A node that does not respond after two retries is identified as failed. The current implementation of automatic recovery can recover from a single failure at a time, but multiple failures during the course of the program’s execution are identified. If multiple nodes fail simultaneously, the current implementation notifies the user, identifies the failed nodes, and exits. It may be possible to extend the current implementation to allow the recovery from multiple failures concurrently. In the case of a single node failure, the system creates a new configuration file that is used to automatically invoke the removal of the failing node. An exit message is sent to the other nodes, resulting in the termination of the program on those systems. A "Start Process" message is then sent to the Brazos remote access service on the first node in the configuration file (this is identical to the actions performed by the Brazos Graphical User Interface when a new application is started). The application restarts by performing recovery on the remaining nodes, recovering the data of the failing node as described in Section 6.1. The auto-recovery process is depicted in Figure 20.

In the current implementation, the node in charge of failure detection is a single point of failure that cannot be recovered from automatically; however, manual recovery is available. In addition, because a single node is in charge of detecting failures, network partitioning and rejoining problems cannot be addressed. To address these issues, our implementation can be extended as follows. While initially keeping the same node in charge of generating heartbeat messages, other nodes will also keep track of the time duration since they received the last heartbeat. After a preset timeout period expires, a
node that has not received a heartbeat will assume that the heartbeat node has either
died or is somehow isolated from the group. In reaction to this condition, the non-
heartbeat nodes will issue a broadcast "panic" message (using multicast if available) to
all the nodes that should be running the application. Each node that receives the panic
message will respond to it. By protocol definition, the node with the smallest node
number announces itself as the new "heartbeat" node and initiates recovery. This
implementation would allow the recovery from the heartbeat node's failure as well as
from network partitioning problems; however, it will not handle a network rejoin. This is
because, as a result of the initial failure, each group of nodes in a partition would have
formed their own cluster, thereby becoming oblivious to the return of the previous nodes.
One way of solving this is to maintain information about the composition of the initial
cluster across failures and to continuously attempt to contact the dead nodes in
anticipation of their return. Upon return, a node addition can be performed to return the
cluster to its initial state (see Chapter 7).
We measured the execution time overhead of the heartbeat messages and found it to be negligible, although it can become substantial if multicast was not available or if the size of the cluster was large. Assuming no retries or lost messages using multicast, heartbeat messages occur at a rate of $6n$ messages per minute with 10-second intervals, where $n$ is the number of nodes in the cluster. This is a very low number of messages, so it is not surprising that failure detection does not add considerable overhead. In the VIA implementation, this rate increases to $12(n-1)$ messages per minute since VIA does not support hardware multicast. When $n$ becomes large, this can add a substantial number of
messages for failure detection and may adversely affect performance. One solution to
this is to reduce the frequency of heartbeat messages.

Another solution that we implemented using VIA relies on a runtime system provided
callback function that is used to handle VIA-related errors. When a node is lost, its
associated connections to the other nodes are also dropped. This triggers the callback
function, which then performs the necessary cleanup. We use this technique to detect
failures instead of the heartbeat thread and observe three important benefits. First, there
is no runtime overhead to this approach, since the callback function is only called when
failure occurs. Second, the appropriate recovery actions are performed immediately upon
detecting a failure, so there is no time lag between the occurrence of a failure and
recovery initiation as is the case using the heartbeat method. Finally, drawbacks of
relying on a single node to perform failure detection are eliminated because all nodes in
the system can detect a failure. The node performing the recovery initiation is picked
depending on its node id and the node id of the system that has crashed.
Figure 21: Performance of Checkpoint-Based Node Removal.

6.2.1 Checkpoint-based Node Removal Performance

To measure the performance of node removal, we ran our four benchmark applications on four systems with four threads each, while enabling automatic failure detection. We then intentionally terminated a process running on one of the systems and measured the time taken to recover the application on the remaining nodes. We also measured the time it takes to detect the failure and start the recovery process. The average across several runs was 7.69 seconds, using a 10-second heartbeat interval. Since detection and restart time is independent of the user application, we used this average for all applications in Figure
21. which shows the total single failure recovery time subdivided into the
detection/restart, checkpoint recovery, and runtime system/application initialization
components for each application. The network filesystem bandwidth observed during
checkpoint recovery ranged from 3.61MB/s to 4.67 MB/s over 100Mbps Ethernet with a
single disk at the remote server\textsuperscript{4}. We expect the performance during checkpoint creation
and recovery to improve if a higher bandwidth disk array and/or network is used.

6.3 Online Node Removal

Although the ability to recover from a single node failure is useful, it is sometimes
desirable to reduce the number of nodes participating in a parallel execution without
having to checkpoint and restart. This can be the case when a system is needed for other
applications, or when a node needs to be brought down for maintenance purposes. This
section describes the implementation of a node removal feature that can be invoked while
an application is in progress without requiring a checkpoint or a repartitioning of the
parallel computation. In addition, this online node removal may be initiated manually by
the user, automatically as a reaction to a system logout or shutdown event, or by an UPS
(uninterruptible power supply) when a power failure is detected. The key to the success
of this implementation is the ability to migrate application threads and data from a

\textsuperscript{4} This bandwidth was observed at the node that recovered the largest total checkpoint file
size.
removed node to the remainder of the cluster. We provide a detailed description of the implementation and performance of this feature next.

6.3.1 Implementation Details

The implementation of online node removal involves three components. First, a mechanism for initiating node removal is required. Second, a means of moving data to other nodes while preserving coherence is needed. Finally, a way for moving threads to the remaining nodes needs to be devised. We discuss each of these components in turn next.

Our design objective for online node removal was to implement an efficient mechanism that can handle all the situations in which a node might be removed from the system. There are three situations that may necessitate online node removal:

1. A system is needed for a more important workload. This will typically involve a user or administrator’s input.

2. A system is undergoing maintenance and needs to be shutdown for some time before being brought back online. The person performing the administrative task may or may not be aware of running applications, so node removal should happen as transparently as possible.
3. A system has experienced power failure and has a limited amount of battery backup. This case should be handled automatically if the backup battery system is able to signal the system when power is lost.

The first two cases require the detection of a set of events, namely user input or a system shutdown/logout request. Our implementation handles these cases as follows. All parallel application programs that run on Brazos are "console" applications, implying that they all execute within the context of a Windows NT console window. The Win32 API provides a function that allows user applications to install special handlers for a set of events. This API, `SetConsoleCtrlHandler`, is used to install a Brazos console event handler (shown in Figure 22). The handler manages explicit node removal requests from the user, who is required to issue a Control-C to interrupt the application and request data and thread migration. For shutdown and logout events, Brazos assumes that the application should be preserved and initiates online node removal automatically. The final case can be handled in two ways. The first option leverages existing Windows UPS support that provides an option to trigger a user-specified program on certain power conditions, such as a UPS reaching a critical power level. This feature can be used to run a program that communicates with running Brazos applications to initiate node removal. The second option relies on a shutdown triggered by a UPS on power failure that is treated just like any other console shutdown event.
BOOL WINAPI CatchConsoleEvents(DWORD EventTypes) {

    char c;

    switch (EventTypes) {
    case CTRL_C_EVENT:
        printf("**********Received a Ctrl-C Event! **********\n");
        if (NODE_ID == 0) // IGNORE IT FOR NODE 0
            return (FALSE);
        else {
            while (1) {
                printf("Would you like to move the data/threads (y/n)?");
                c = getch(stdin);
                switch(c) {
                case 'y':
                case 'Y':
                    gDsmBeingRemoved = NODE_ID;
                    return (TRUE);
                    break;
                case 'n':
                case 'N':
                    exit(0);
                    break;
                default:
                    printf("Unknown input, please try again!\n");
                }
            }
            break;
    case CTRL_CLOSE_EVENT:
        printf("Received a Ctrl-Close Event!\n");
        gDsmBeingRemoved = NODE_ID;
        break;
    case CTRL_LOGOFF_EVENT:
        printf("Received a Log-off Event\n");
        gDsmBeingRemoved = NODE_ID;
        break;
    case CTRL_SHUTDOWN_EVENT:
        printf("Received a shutdown Event\n");
        gDsmBeingRemoved = NODE_ID;
        break;
    default:
        printf("Received an UNKNOWN Event\n");
    }
    return TRUE; // this disables the default ExitProcess() handler
}

Figure 22: Online Node Removal Event Handler.

The event handler performs one of three operations. If the event occurred on the root node, the event is ignored. This is because the root node, which currently acts as the home node for shared memory as well as the barrier manager, cannot be removed online. If the user requests a node removal or an automatic removal event occurred, such as a shutdown, logoff, or console close event, the handler sets a global variable to indicate
that a removal request should be sent out. Finally, if the user does not request removal, the process is terminated. Once the removal request is registered, node removal proceeds as follows. At the next application barrier, the removal flag is checked, and if it is set, the barrier arrival message sent to the barrier manager includes a flag informing it of the node’s request to leave. The barrier manager waits for all nodes to arrive and sends a special node removal barrier release message to all nodes. Upon receiving this message, all remaining nodes wait for the data and threads to arrive from the node being removed. After performing all the coherence actions included in the barrier release message, the node being removed immediately migrates its threads to the remaining nodes before initiating data migration. All application threads remain suspended until this operation completes.

Data migration requires sending each page containing valid data and any modifications to the remaining nodes in the system so that computation may proceed correctly after node removal. The actions performed at the to-be-removed node are very similar to those done at checkpoint creation with respect to gathering the required shared memory data to migrate (refer back to Figure 12). The most significant difference is the means of transferring the data to the remaining nodes. This is accomplished by sending the page status blocks, actual data pages, and any twins or diffs to the rest of the nodes. To minimize bandwidth requirements while maximizing throughput, two optimizations are implemented. First, we include as much information as possible in the maximum allowed message size according to the underlying communication layer. Second, to
minimize the number of messages sent, the UDP implementation utilizes multicast when sending the data migration messages. Each destination node performs recovery actions that are identical to those performed for checkpoint-based node removal (see Figure 18). To eliminate the possibility of overrunning message buffers at the recipients, the node to be removed waits for acknowledgements before sending subsequent data migration messages. Once all data migration messages complete, a node removal completion message is sent to the remaining nodes, allowing all application threads to proceed. The online node removal process is summarized in Figure 23.

Figure 23: Online Node Removal Process.
Some synchronization issues arise with online node removal. Specifically, since Brazos uses a distributed lock manager implementation for application locks, a mechanism for handling the removal of a lock manager node is required. This is handled in our system by redistributing the lock managers as soon as the first data migration message is received by assigning managers in a round-robin fashion while skipping the removed node. For barriers, it is also necessary to change the barrier flags to reflect the new number of nodes that constitute the new cluster (represented by a bitmap) and the total number of threads to wait for at a barrier. This problem is handled by changing the barrier bitmap when data migration starts. The number of user threads is automatically adjusted as application threads from the removed node arrive at their destinations. No races similar to those discussed in Section 4.3.3 are introduced, because online node removal occurs at barriers.

<table>
<thead>
<tr>
<th></th>
<th>SOR</th>
<th>Water</th>
<th>3DFFT</th>
<th>Barnes-Hut</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset Used</td>
<td>4000×4000</td>
<td>4096 Mol</td>
<td>$2^6 \times 2^7 \times 2^6$</td>
<td>32K Bodies</td>
</tr>
<tr>
<td>Total Removal Time</td>
<td>2.22 sec.</td>
<td>1.88 sec.</td>
<td>0.63 sec.</td>
<td>0.52 sec.</td>
</tr>
<tr>
<td>Data Migration Time</td>
<td>2.20 sec.</td>
<td>0.96 sec.</td>
<td>0.60 sec.</td>
<td>0.38 sec.</td>
</tr>
<tr>
<td>No. Data Messages</td>
<td>978</td>
<td>421</td>
<td>256</td>
<td>155</td>
</tr>
<tr>
<td>Total Data</td>
<td>53.71 MB</td>
<td>23.46 MB</td>
<td>14.43 MB</td>
<td>8.85 MB</td>
</tr>
<tr>
<td>Ave. Message Size</td>
<td>57584 bytes</td>
<td>58437 bytes</td>
<td>59114 bytes</td>
<td>59880 bytes</td>
</tr>
<tr>
<td>Number of Pages</td>
<td>6857</td>
<td>2781</td>
<td>2461</td>
<td>986</td>
</tr>
<tr>
<td>Bytes Per Page</td>
<td>8213 bytes</td>
<td>8846</td>
<td>6149 bytes</td>
<td>9413 bytes</td>
</tr>
<tr>
<td>Bandwidth Seen</td>
<td>24.44 MB/s</td>
<td>24.38 MB/s</td>
<td>24.14 MB/s</td>
<td>23.01 MB/s</td>
</tr>
</tbody>
</table>

Table 5: Online Node Removal Performance Statistics.
6.3.2 Online Node Removal Performance

To measure the performance of online node removal in Brazos, we ran our four application benchmarks on four nodes with four threads each (a total of 16 threads). We then manually issued an online node removal request. Table 5 summarizes the performance characteristics of the facility. The total online removal time ranged between 0.52 and 2.22 seconds. These numbers include the total time elapsed from the arrival of the "leave" barrier arrival from the node being removed until all data and thread migration activity completes. The bulk of this time is spent migrating data (between 0.38 and 2.20 seconds for our applications) or waiting for the barrier. We also gathered statistics on the number of pages transferred, the number of messages and their average sizes, and the network bandwidth observed during the data migration operation. As the figure shows, our implementation was effective at utilizing network bandwidth by maximizing the message sizes to approach the maximum allowed by our network (about 64KB). The effective bandwidth seen during the data migration operation was about 24MB/s for all the applications. This is considerably less than the maximum bandwidth of the underlying VIA network, primarily due to the small acknowledgement messages that need to be exchanged in addition to the intervening processing (including copying) that occurs at the recipient nodes. The average amount of data transferred per page ranged from 4136 to 9413 bytes and reflects the number of twins and/or diffs that are included per page. On our experimental platform, a page and a twin are 4096 bytes in size, and diffs can have a maximum of 8192 bytes. This aspect of the data migration operation is application dependent.
Chapter 7  Node Addition

The integration of thread migration in Brazos allows new or replacement computing resources to be brought on-line with minimal interruption of running applications. This situation can arise when a failed node is repaired and placed back in service (e.g., after automatic failure recovery or online node removal), or when one or more additional computing nodes become available during the execution of a long-running application that may benefit from more processing capacity. In such a situation, the application is started on a certain number of nodes with a number of threads that exceeds the number of available processors. The same is true when a node is removed, causing existing processors to become overloaded with computation threads. The performance degradation caused by overloading processors with threads is application dependent, because it depends on the amount of extra communication introduced when more threads exist on each system. The overhead of scheduling threads is relatively small.

Without thread migration, it is only possible to add computing nodes by interrupting the running application, repartitioning it, and creating as many threads as necessary in the resulting processes. This can be time consuming, and requires a programming language that allows frequent repartitioning of applications. Our approach is different, because it does not require the interruption of computation on existing nodes. In our system, the only action necessary to add new computing nodes is to notify existing ones to migrate some of their threads to a “shell” process on a new system. Besides the migration of the
computation threads, it is necessary to ensure that the data accessed by the threads on their new node can proceed without violating coherence. Since the Brazos runtime system already supports the automatic migration of shared memory pages among processes participating in a computation, this issue is resolved by transferring the necessary coherence state information to the new node before it receives the computation threads. Finally, before joining an existing cluster, the new node has to initialize the runtime system and any read-only application variables. These operations are performed off-line without interrupting the running application. We describe the implementation of node addition in detail in the following sections.

7.1 Network Communication

The first challenge encountered while implementing online node addition was finding a mechanism by which a joining node can notify existing nodes of its intent to join. This section describes the mechanism that we implemented for this purpose.

The first step is to create a Brazos configuration file that includes the existing nodes as well as the new node, which is marked as being added. This configuration is used to start the application on the joining node. Once the runtime system is initialized, the joining node can contact the existing cluster. The mechanism by which the new node contacts an existing cluster depends on the networking protocol being used. For the datagram protocol (UDP), a joining node can simply join the appropriate multicast groups using the
same multicast addresses and port numbers that are currently in use by the running cluster. This allows the new node to contact the others by sending a “join” request to all currently running systems informing them of its intent to join the computation and of the number of threads that it would like to receive from each system. For VIA, processes are required to establish virtual interface connections before communication may take place. Establishing VI connections requires knowledge of the host names and discriminator values for each connection. Since a running node cannot have a-priori knowledge of the identity of a joining node, it was necessary to develop a mechanism for informing the running systems of this information before VI connections may be established. We decided to implement a runtime system Command Thread that is created during runtime system initialization and does one of two things. For normal nodes (i.e., those that are run using a standard configuration), this thread creates a socket, binds it to a predetermined port, and waits for command messages to be received. For joining nodes, the command thread performs two functions. First, it prompts the user for the number of threads to transfer from each existing node. Second, it creates a socket used to communicate with the command threads on other nodes. The command thread is created after the runtime system is initialized on the joining node and immediately contacts the existing nodes using the host information stored in the configuration file. At the recipients, the incoming node's host name is discovered from the incoming message and stored in the configuration. An acknowledgement is sent back to the joining node. Then, the joining node's host name is converted to the format used by VIA. At this point, all the nodes execute procedures to establish the necessary virtual interfaces with the joining
node in parallel, without interruption ongoing computation. For the UDP implementation, the existing nodes join the multicast groups needed to communicate with the joining process. At the end of this phase, the application initialization phase is initiated on the joining node (see Figure 24).

Application initialization is necessary to ensure that all static or heap-allocated read-only variables are correctly initialized. This initialization is performed by special thread that calls the UserMain() function with a global thread id of (-1) and a local thread id of (0). To ensure that this initialization proceeds correctly, the following guidelines should be followed:

1. Memory allocation should be skipped. This is because the new node will inherit the memory allocation information from existing nodes (see Section 7.2).
2. The node addition initialization phase ends with the call to the recovery barrier. All initialization should be completed at this point. Note that this condition is also required for checkpoint/recovery.
3. The initialization thread's global thread id (-1) can be used to prevent the execution of code that should be excluded. The local thread id (0) should be used to ensure that a section of code is executed.
4. It is not necessary to initialize any shared memory locations during node addition.
The rest of the node addition process proceeds as follows. When the initialization thread reaches the recovery barrier, it issues a “Join Cluster” request to the existing nodes and waits for an acknowledgement. The barrier manager node responds and includes some shared memory allocation information (see Section 7.2). The joining node exits the recovery barrier without waiting for it. At the following application barrier, which we referred to as the “Join Barrier”, the barrier manager waits for all existing nodes to arrive.
Once all nodes have arrived, a message carrying shared memory coherence information is sent to the joining node. After updating its coherence information, the joining node sends an acknowledgment to the barrier manager, which will then continue to release all nodes in the system. Before exiting the join barrier, all nodes will migrate the number of threads requested by the joining node (as specified in the initial connect request message). This process is summarized in Figure 25.

7.2 Memory Allocation

The initialization of the shared memory pool and its associated data structures is performed by the Brazos runtime system when the application program is started. Once this initialization is complete, user programs can allocate shared memory as needed. In the case of node addition, the shared memory regions that have been allocated on the existing nodes (and the pointers to those shared regions) are not initialized on the new node. It is therefore necessary to notify the new node of existing memory allocations so that migrating threads do not encounter unhandled memory exceptions due to null pointer accesses. A mechanism for transferring shared memory allocation information is necessary to resolve this problem. To address this, we modified the shared memory allocation routines in Brazos. For each shared memory allocation, we save the address of the shared memory region allocated and the virtual address of the variable that will receive the shared memory region's address, which is passed as an argument to the allocation routine. The only exception to this is when the pointer itself resides in shared
memory. In that case, it is not necessary to save the allocation information because such pointers will be brought in when the shared memory pages in which they reside are faulted on.

The shared memory allocation information is bundled in the acknowledgement sent to the joining node on a reply to a join request (see Figure 25). Upon receiving this information, the new node sets each pointer to the appropriate value before proceeding. When threads arrive at the joining node, they will access the correct shared memory locations and fault on the first access to a shared memory page. The mechanism used to ensure that the page fault handler will send requests to the appropriate nodes in the system is the subject of the next section.
Figure 25: Node Addition Process (after connection establishment).

7.3 Memory Coherence

Ensuring consistent shared memory is one of the most complex issues (from the standpoint of implementation) in adding a node to an executing program. All runtime
system data structures that are used to track modifications to shared memory pages must be brought up-to-date on the new node. This is essential since the threads at the new node will access shared memory pages that need to be updated by requesting data from one or more other nodes. There are two relevant sets of information kept for each shared memory page: the set of nodes that have copies of a page (the copyset), and the set of nodes that have modified the page (the invset). When a page fault occurs, the new node relies on the information in these sets in order to know 1) the identity of nodes to which requests for pages should be made, and 2) when it has received all of the necessary page data and modifications. In order to update the joining node's copysets and invsets, the barrier manager sends this information to it. This is done after all other nodes have arrived at the join barrier and updated the invsets and copysets to reflect the modifications performed at all nodes before the barrier. To ensure that the coherence information is applied at the joining node, the barrier manager waits for acknowledgements from it before releasing the rest of the nodes from the barrier. The runtime system structures in Brazos were modified to improve the performance of this process by keeping the copysets and invsets in contiguous arrays, instead of as individual fields in the page structures. When the nodes exit the join barrier, the required number of user threads is then migrated to the new node and computation proceeds.

7.4 Synchronization

The Brazos API provides synchronization primitives in the form of barriers and locks. Both of these synchronization mechanisms present problems when a new node is added.
In the case of barriers, it is required that all nodes use the same barrier instance at the same point in a program. The barrier implementation in Brazos uses two barrier instances, and alternates between them at barrier calls. In order to guarantee that the new node will use the same barrier instance as the others once it joins the cluster, this information is encoded as part of the “Join Cluster” acknowledgement message.

The second and more complex problem encountered relates to lock management. Brazos uses a distributed lock scheme that statically assigns lock managers to nodes within the cluster in a round-robin fashion. This assignment is performed in an identical fashion on every node in the cluster. When a thread needs to acquire a lock that is currently held by a remote process, an acquire message is sent to the appropriate lock manager process. When a new node is added to the system, the default lock initialization scheme would attempt to include the new node as a lock manager. This fails because its mapping of managers is different from that on the “old” nodes. In addition, a simple solution that forces the new node to distribute the lock managers among the original number of nodes might work for adding a single node, but would fail in general with the sequential addition of multiple nodes. The solution to this was to redistribute all lock managers at the join barrier.
7.5 Performance of Node Addition

We measured the performance of node addition by running the four benchmark applications with three nodes initially and distributing a total of sixteen threads among them. This resulted in a single node with six threads and two others with five threads each. We measured the latency of node addition for adding one node and requesting a total of four threads from the existing nodes; two from the six thread node, and one each from the five thread nodes. We measured the total elapsed time from sending the connection request until the last application thread arrived at the new node, including the delay incurred while waiting for all nodes to arrive at the join barrier. In addition, we measured the amount of time spent initializing the application. Table 6 shows that node addition can be accomplished in a short amount of time, ranging from 0.3223 to 2.7734 seconds for our benchmark applications. Since thread migration, connection request messages, and coherence data exchanges are fast operations, most of the node addition latency is spent initializing the application and waiting for the nodes to reach the join barrier. This variably component of node addition depends on the application and the time at which join requests are made.

<table>
<thead>
<tr>
<th></th>
<th>SOR</th>
<th>Water</th>
<th>3DFFT</th>
<th>Barnes-Hut</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time</td>
<td>0.3223 sec.</td>
<td>0.4486 sec.</td>
<td>2.1188 sec.</td>
<td>2.7734 sec.</td>
</tr>
<tr>
<td>App. Initialization Time</td>
<td>0.0023 sec.</td>
<td>0.0100 sec.</td>
<td>0.0008 sec.</td>
<td>1.3234 sec.</td>
</tr>
</tbody>
</table>

Table 6: Performance of Node Addition.
Chapter 8  Related Work

This chapter discusses previous work in fault tolerance, thread migration, and adaptability on software distributed shared memory systems. Since there appears to be no previous work that combines thread migration and checkpoint/recovery techniques, related work is divided into three parts. First, we present other systems that utilize thread migration. Then we discuss various fault tolerance techniques proposed for software distributed shared memory systems. Finally, we discuss previous work that supports the addition and removal of nodes on SDSM system.

8.1  Thread Migration

There are several other software distributed shared memory systems of which I am aware that utilize thread migration: Millipede [24], D-CVM [55], Nomad [38], Ariadne [37]. We discuss these systems and point out the similarities and differences between our approaches.

Millipede is a distributed shared-memory system implemented on homogeneous networks of systems composed of Intel x86 based computers running Windows NT. Millipede’s developers proposed and implemented a user-level thread migration mechanism that is similar to the one that we have implemented, as discussed in Chapter 4. In particular, they use the same method of ensuring that all thread stacks reside at
identical virtual addresses across all systems running a distributed process. In addition, they impose the same restrictions that we do on the usage of shared memory by threads. Specifically, they require that all non-stack data accessed by a user thread be allocated in shared memory, even if that memory is logically private (i.e., only accessed by a single thread throughout a program’s execution).

Millipede’s runtime system uses thread migration to reduce communication costs. It accomplishes this by keeping track of all accesses made by a thread that result in inter-node communication (i.e., page faults). They observed that migrating a thread incurs a lot of page faults that are serviced by remote nodes without accounting for the frequency and number of local pages that are accessed may result in poor placement of threads. Poor thread migration decisions result in an increased amount of inter-node communication to the previously local pages and can be remedied by returning the offending thread(s) to their initial host, resulting in what is referred to as a ping-pong effect. Making poor decisions improves future migration decisions by providing additional sharing information that would have otherwise remained unknown to the runtime system.

There are a few differences between their approach and ours. First, their implementation provides the automatic migration of user threads and does not discuss issues related to the explicit migration of threads by programmers. They also do not discuss how their system reacts to sudden changes in communication patterns that may be impossible to react to in
a timely fashion in order to affect the reduction of communication sought. Second, their system only supports thread migration under a strict or strong consistency memory model, sequential consistency [32]. Brazos supports both release consistency and scope consistency, which allows it to outperform systems using strong consistency models by delaying and aggregating communication. Thread migration in Brazos can be used under both memory consistency models without violating correctness, although some restrictions are imposed. Finally, Millipede does not use thread migration for node removal or addition and does not provide a checkpoint/recovery facility to handle failures in the cluster.

The performance of thread migration in Millipede as quoted in [24] appears to be at least an order of magnitude slower than it is on Brazos. Although no specific details were given for the speed of the Pentium processors used to make their measurement of migration latency (70ms using Fast Ethernet and 4KB stacks), we observe latencies of about 1ms on our experimental platform using Fast Ethernet (see Section 4.5). It is difficult to understand the performance differences without acquiring and analyzing their implementation; however, one important difference could be their use of the MGS library that automatically makes migration decisions, which might be introducing an extra layer of latency during thread migration. The differences in our experimental platforms cannot justify the 70-fold improvement seen in our implementation.
Finally, Millipede imposes some restrictions on thread-to-thread synchronization as well as on system calls. Our implementation imposes the same restrictions on system calls and system objects; however, our distributed locking and barrier implementations allow threads to migrate and synchronize as long as they do not hold a synchronization object during the migration process. Our current implementation will ignore user thread migration requests for threads that are holding user locks.

The D-CVM (Dynamic Coherent Virtual Machine) DSM system uses thread migration to dynamically redistribute computation threads to nodes to reduce communication and improve load-balancing. The system relies on a correlation tracking mechanism to direct the redistribution of threads. Instead of solely relying on the tracking of page faults as in Millipede, the D-CVM system contains an active thread tracking mechanism. This mechanism allows the tracking of sharing among local threads by both serializing thread execution and adding per-page access counters for each thread in a SDSM process. The addition of this tracking mechanism is purported to avoid the ping-pong or thread-thrashing effect that was observed in the Millipede implementation.

We followed D-CVM's approach to coherence during thread migration. Their system uses a Multiple-Writer [9] and LRC (Lazy Release Consistency [26-28]) coherence protocol; therefore, care has to be taken by programmers to ensure correctness when threads are migrated. D-CVM solves this problem by only allowing migration at fixed
synchronization points (barriers). Our approach is to also allow migration at other points in the program, as long as certain rules are followed (refer to Section 4.3.2).

They discuss two approaches to the stack address/pointers problem: re-mapping or scanning of addresses at the destination node, or Reserving the same addresses on all nodes for threads. The second approach is similar to that used in Millipede and in Brazos. The first approach requires scanning the stack data of a migrating thread and translating all pointers to stack data to the correct addresses on the new host. D-CVM solves the stack address problem by reserving stacks on all node, just as we do in Brazos. D-CVM handles static and heap allocated thread-private data by disallowing “any non-shared data that is processor dependent” [55], in other words, they require that all such data reside in shared memory. Finally, they quote some thread migration performance results on three platforms: IBM SP2 using 66.7MHz Power2 processors over a 40MB/s SP2 switch, DEC 275 MHz 21164 Alpha processors using a 155Mbps ATM, and a SUN UltraSPARC system using Ethernet. The migration costs for the reserved stack solution ranged from 1583μsec on the IBM SP2 with a stack size of 1704 bytes to 2786μsec on the UltraSPARC with a stack size of 1280 bytes. Our implementation is faster than both platforms, but there are many differences in hardware and in the thread implementation (they use the NewThreads [29] user-level thread library). This shows that our kernel level thread migration implementation on Windows NT is competitive with user level implementations despite the fact that operating system calls have to be performed to implement migration in our case.
Nomad is a software distributed shared memory system that supports thread migration [38]. They propose thread migration to address both communication and load balance problems. They use thread migration at page faults instead of requesting and receiving the pages from other nodes as is typically the case with other SDSM systems. When a thread faults on a page, the node that it resides on copies the top and bottom pages of the thread's stack into a request message that is sent to the node containing the page. The node receiving the request either accepts the thread or sends the page back to the requesting node according to some heuristic referred to as a bidding system. The main principle behind the bidding system is that migration is only used when a thread's bid is higher that a page's value. Bids and values are determined using knowledge about the sharing behavior exhibited by the page and various system resource parameters. Timing results on a Nomad implementation running on Sun Microsystems SparcStation 2 workstations using 40MHz SuperSPARC processor and 10Mbps Ethernet (UDP) show a thread migration cost of 8ms and a page migration cost of 15ms.

Another system that supports thread migration is Ariadne [37]. It is a user level thread library that runs on hardware and software distributed shared memory systems. It supports inter-process thread migration and uses it to implement its library on distributed systems. The system's initial goal was to support parallel process-oriented simulation environments; therefore, the library allows user-defined thread schedulers.
Acquiring a thread’s stack and context is simple in Ariadne since this information is kept in a data structure maintained by the runtime system. Ariadne’s approach to the stack pointer problem is to scan and re-map pointers at the destination node. Although this approach has its drawbacks, as we discussed earlier, it appeared to be reasonable since their system might support hundreds of thousands of user threads at a given time. A large number of threads makes it wasteful to pre-allocated or dedicate stack space on all nodes as is done on the Brazos, Millipede, and D-CVM DSM systems. A significant drawback of Ariadne’s re-mapping policy is that it might result in incorrect execution, because of the re-mapping of data values and or the inability to re-map addresses kept in registers. Thread migration performance measurements on a network of Sun SPARCstation 5 systems were in the range of 11-16ms for a roundtrip thread migration operation. They attribute most of the latency to message overheads rather than thread packing or unpacking functions. The Ariadne implementation uses a dedicated thread for packing and unpacking threads, similar to the Migration Agent thread in Brazos.

There is a large body of research related to process and thread migration on non-DSM distributed computing environments. We present some of the most significant efforts in this area next.

Emerald [25] is an object-oriented programming environment that supports object mobility among systems. Objects in the Emerald environment are free to migrate among nodes in a distributed system. Emerald provides primitives to fix and locate objects. It
takes about 27.9ms to send a remote invocation message and receive its reply on a network of 4 DEC MicroVAX II workstations connected with a 10Mbps Ethernet. In more recent work [52], the Emerald system was extended to allow the migration of objects and threads among heterogeneous computer systems. In order to overcome differences in stack structures, they introduced the concept of a bus stop, which simplifies the problems associated with migrating threads by only performing these actions at predetermined points in a program. Since the bus stops are at predetermined locations, it is possible to reconstruct the thread stacks when they arrive from a machine with a different architecture. They handle differences in data representation on the different architectures (e.g., big or little-endian systems) by converting objects to a common format during object migration.

The Amber system [10], which is the successor to Emerald, also supports the migration of threads in a distributed computing environment. Amber is an object-oriented programming environment designed to utilize networks of multiprocessor workstations. All objects in an Amber system are mobile and are free to move around the distributed system. The major difference compared to Emerald is that Amber provides a consistent address space for objects no matter which actual system they reside on. In Emerald, it was necessary to perform an address translation on different machines. The fact that all nodes use the same addresses also makes it easier to implement thread migration because it avoids the problem of having to translate addresses upon migration. Performance
results were gathered on a network of Firefly multiprocessor systems. A remote invocation and return consumed 8.3ms.

Spring [17] is a distributed, object-oriented, multi-threaded operating system that supports object invocations across multiple systems (or domains). This is accomplished by creating multiple threads, one each for the domains involved. These threads are then used to host the call locally. An interesting aspect of their implementation is that it is not necessary to copy stacks during inter-domain calls.

Active Threads [57] is a user-level thread library that includes support for migration. The library provides programmers with an API that includes functions specific to thread migration. One of the main goals of the Active Threads package is performance and they achieve it mainly by utilizing an efficient user-level communication package based on Active Messages [13] (a variant of the Berkeley AM library). Their solution to the stack pointer problem is similar to ours; they reserve the same stack address space on all nodes. Thread migration latency was measured on a network of SPARCstation-10 multiprocessor workstations, each containing 4 50MHz HyperSPARC processors and a Myrinet network interface connected to the SBUs. The one-way latency of a 5-word message on their system is 17μsecs. A bulk transfer of 1KB takes 560μsecs and is limited by SBus bandwidth. Thread migration latency is about 1.1ms for a 2KB thread. Thread migration latency is lower than 1.1ms for both Fast Ethernet and VIA on our experimental platform.
8.2 Fault Tolerance

There is a significant number of checkpoint/recovery implementations on software distributed shared memory systems. In this section, we compare our implementation to the most significant previous work in the area.

Costa et al. implement a two-level [56] checkpoint mechanism. They use a lightweight logging mechanism to support single node failures and perform occasional consistent checkpoints to implement multiple node recovery. Lightweight logging uses the coherence messages of TreadMarks to maintain the necessary information needed to recover from single node failures. For every SDSM process, the logging system maintains a set of lists that contain information about every message that was sent to or received from every other node in the system. This tracking of all the coherence traffic allows their system to replay or reproduce the shared memory state a failed process up to the point of failure. By distributing this information on the nodes in the cluster, the system can recover from a single failure. The recovery mechanism relies heavily on the vector timestamps used in the TreadMarks LRC implementation in order to determine the order of writes seen by the failed process. The implementation also requires maintaining some information that may not be needed under normal circumstances (e.g., diffs). The performance results in [12] show that the overhead of maintaining the logs is very small, almost negligible. 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). Because the replay time is considerable, it is difficult to justify adding such a log-based recovery mechanism to the system. The consistent checkpoint facility was implemented using an existing checkpointing tool called Libckpt [41]. No performance results were given for consistent checkpoints, although the paper presents some figures using independent checkpoints to save the logging information usually kept in volatile memory. Their log-based checkpoint overheads are 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 them. Unfortunately, they do not list the overhead of their consistent checkpoint mechanism. Our consistent checkpoint facility’s overhead using two-minute intervals is lower than their log-based implementation (for the same dataset sizes). Our rollback recovery mechanism allows faster recovery, because it does not consume a significant amount of time replaying memory operations from a log.

Brazos differs from TreadMarks in the way it maintains coherence. Brazos does not require timestamp vectors and would require additional support in order to maintain the kind of information that was used to implement the low-overhead logging scheme discussed above. In addition, since we intend to use checkpoints to move processes around, we require both fast checkpoints and fast recovery.
Cabillic et al. [7] designed a consistent checkpoint facility for a distributed shared memory system that is similar to ours in many aspects. They perform checkpoints at barrier synchronization points that are annotated by the programmer. Their checkpoints require the copying of pages and page descriptors (page information blocks) only and do not require saving diffs because their DSM system, MYOAN [8] implements sequential consistency and an invalidation-based coherence protocol similar to IVY [34]. 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 to perform the checkpointing, similar to our checkpoint thread; however, their system requires sending messages to the server, whereas our system uses a thread per process and communication is through hardware shared memory, which is much more efficient. In order to optimize the performance of the checkpoint facility by minimizing the amount of blocking time while the checkpoint is committed to storage, they use three enhancements. First, they use incremental checkpoints. They use two schemes to track pages that have been modified since the last checkpoint in order to know which pages to save. The *accurate* scheme uses a dirty bit in each page's descriptor and the checkpoint server examines the page tables and only copies pages that have been modified. This requires maintaining multiple checkpoint files for recovery. The *estimate* scheme avoids the overhead of the accurate method by saving all pages that are owned by a process and is read-write, but only read-only pages that have the dirty bit set. In this case, the dirty bits are set when the owner of a page changes and is reset when the page is saved to disk. The second optimization used avoids blocking while checkpoints are written to disk.
especially since messages are used to communicate with the checkpoint server. They use a *pre-copying* or a *copy-on-write* [14, 35] method for this purpose. Pre-copying uses an implementation that does not wait for acknowledgements and copy-on-write protects pages against writes while the server is reading them. The final optimization they use is called *page pre-flushing*. This method attempts to use the load imbalance in applications to perform some of the checkpointing once the process reaches a barrier and while it is waiting for other processes to reach the same barrier. The processes send a message to the checkpoint server with all the pages that it owns. Since this list may change depending on the access patterns of the other processes, the page directory is not written until all the processes reach the barrier. Pages that change ownership will be discarded from the checkpoint according to the contents of the page directory.

The above consistent checkpoint system was implemented on an Intel Paragon system consisting of 56 computing nodes [23] and running the Paragon/OSF1 operating system. Each node contained two Intel i860 microprocessors, 16MB of main memory, and a network interface to a grid topology interconnection network. Performance measurements of the execution time overhead of the non-optimized checkpoint tool on four applications (Mp3d, Matmult, MGS, and Radix) ranged between 8% and 47%. The performance of the *estimate* incremental checkpoints was measured only on Mp3d (8.7%) and Matmult (6.5%). Interestingly, the performance was slightly worse for Mp3d but almost 50% better for Matmult.
Kermarrec et al. implement a recoverable distributed shared memory system called ICARE [31]. 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.

There has been no previous work on managing parallel SDSM applications on a pool of resources using Windows NT or 2000. However, there has been a large body of work on the managing of sequential applications and some work on message-passing parallel applications running in such environments. One of the most notable distributed computing management systems is Condor [36]. Condor manages a pool of privately owned computing resources and a pool of applications. Its most important feature is that it attempts to schedule applications on idle machines and to migrate them as soon as user activity is detected on the host in use. The migration of applications is enabled using a checkpointing facility that is linked in with applications that run using Condor. The checkpoint facility used by Condor is implemented on multiple UNIX platforms. Condor does not support parallel shared memory jobs. More recently, a checkpoint facility called CoCheck [53] has been implemented to support parallel message-passing applications using MPI.

Previous researchers have developed checkpoint facilities for Windows NT processes [20, 51], although none of these work in a software distributed shared memory
environment. Huang et al. implement a recovery facility for NT processes, called NT-SwiFT [20]. 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. [51] implemented a general-purpose checkpoint facility for non-distributed 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 creation time was twenty one seconds for a 50MB process.

8.3 Adaptability and Process Migration

The Microsoft Clustering Service (MSCS) is a facility designed for the Windows NT 4.0 operating system that supports failure-detection, cluster-join, as well as fail-over mechanisms. MSCS can be used with non-modified binary executables, but it also defines an API that can be used by programmers to benefit from all the features of the service. Although the clustering service currently only supports a two server cluster, it performs many of the functions that I propose to support in the future including network
and system fail-over; however, MSCS cannot handle the kinds of reliability issues that I address for software distributed shared memory systems.

A recent study on the TreadMarks [28] DSM system describes an implementation that uses process checkpoint and migration facilities to support adaptive and configurable cluster-based computing using the OpenMP shared memory programming environment [45]. Their implementation uses an existing process checkpoint facility called libckpt [41] to checkpoint and recover DSM processes. When a node needs to be added to a cluster, the program has to be repartitioned for the new number of processes and a new process is created. When nodes are removed, they utilize two options. First, they can migrate the data from the soon-to-be-removed process to other processes if there is enough time to do so (as specified by the programmer). Second, in cases where node/process removal has to be immediate, the victim process is checkpointed and migrated to one of the remaining nodes. This has the effect of overloading a node with multiple processes and can have substantial performance penalties due to limited memory resources, resulting in excessive virtual memory swapping activity. As I have shown in this thesis, using thread migration provides a much more efficient means of achieving the same objectives without requiring the repartitioning of applications or the possible overloading of systems.
Chapter 9  Conclusions and Future Work

In this dissertation, we described the design and implementation of a system that integrates efficient user-level thread migration and checkpoint/recovery mechanisms. We have also described the means by which this integration facilitates the online addition and removal of computational nodes with minimal disruption to running applications. In addition, we demonstrated the efficiency of our system in recovering from failures, detecting and automatically recovering from single node failures, and adding and removing computing nodes. We believe that integrating these mechanisms in SDSM systems overcomes the difficulties encountered when using clusters for parallel computing; namely the increased susceptibility to failure, and the lack of operating system oversight over cluster resources.

9.1  Future Work

There are many opportunities for enhancements to our system and for further research in this area. We briefly describe some of these opportunities next.

The most important contribution of this work is that it provides the building blocks needed to implement a mostly automated parallel application scheduling facility on clusters of multiprocessors. We envision a system that can automatically discover available resources for a given application, schedule the execution of the application on
those resources, and adapt to varying loads and system resources with a minimum amount of user or administrator intervention. Besides the mechanisms described in this dissertation, the Brazos system includes other features that can facilitate the implementation of such a system. The Brazos service, which is currently used for initiating the execution of distributed processes, can be enhanced to include system configuration and load monitors. These features can be used for discovering resources and adapting to changes in loads and configurations while minimizing interruptions to running applications. Such integration of mechanisms provides clusters with the tools necessary to compensate for the lack of a cluster operating system. The integration of Brazos with existing software solutions for high-availability and fail-over, such as the Microsoft Clustering Service, can also improve the availability and utility of such environments.

Although our system attempts to handle the file I/O problem by maintaining enough information about files in order to reopen them after thread migration, or during recovery, our implementation does not guarantee correctness in a number of cases. We do not checkpoint file contents, but we do maintain the last file pointer location used for accessing the file. This poses a problem when applications perform random writes to a file, because we use a rollback recovery mechanism. After such files are reopened during recovery, it is possible for application threads to read stale data, or data that was written to the files after the checkpoint was performed. In addition, one of the difficulties of writing I/O intensive applications on a cluster is that the files are not managed by a single
filesystem. This makes it difficult to allow multiple threads to update the same file, because, unless those threads resided on the same node, the transparent movement of the file pointer after each access will not be performed and file data may be overwritten. To overcome these issues, a distributed filesystem that allows the same files to be concurrently accessed from multiple machines while maintaining a single file handle and pointer is needed. Having such a filesystem would ease the programming difficulties. In addition to that, it may be necessary to maintain file access logs, so that rollback recovery can be applied to files as well as shared memory.

Instead of requiring the continuous creation of checkpoints and the resulting execution time overhead, it is possible to enhance the system by allowing users to request checkpoint creation at arbitrary points in time. For example, checkpoints can be created before scheduled maintenance operations. This requires a simple addition to our current implementation, which depends on the command thread to receive asynchronous checkpoint requests. Such requests would be handled at the barrier immediately following the receipt of the checkpoint creation notification. It is also possible to perform such checkpoints outside of barriers by synchronizing all user threads at predetermined points in the runtime system code. Once all threads in the cluster are suspended, the checkpoint can take place.

As we mentioned in Section 6.2, the current failure detection mechanism can be improved using a more sophisticated distributed membership algorithm that can handle
network partitioning situations without depending on a single resource for failure
detection and recovery. In addition, we would like to provide the functionality described
in this work to message passing programs and programs that integrate both shared
memory and message passing. The objective will be to leverage existing runtime system
support in order to reduce the overhead of creating checkpoints for message passing
programs, possibly by utilizing the underlying mechanisms used to maintain coherence
across the cluster.

Finally, implementing our system on a network of heterogeneous computer systems
presents a large number of interesting challenges. This is primarily due to the differing
architecture platforms, which result in differences in stack representation, data
representation, and processor contexts.
Bibliography


