

# Raptor: Integrating Checkpoints and Thread Migration for Cluster Management

Hazim Shafi<sup>§</sup>  
IBM Research  
11501 Burnet Road M/S 9460  
Austin, TX 78758  
hshafi@us.ibm.com

Evan Speight  
Computer Systems Lab  
Cornell University  
Ithaca, NY 14853  
espeight@csl.cornell.edu

John K. Bennett  
Department of Computer Science  
University of Colorado  
Boulder, CO 80309  
jkb@cs.colorado.edu

## ABSTRACT

*Software distributed shared-memory (SDSM) provides the abstraction necessary to run shared-memory applications on cost-effective parallel platforms such as clusters of workstations. However, problems such as cluster component reliability and cluster management, which are not directly related to performance, need to be addressed before SDSM solutions can be widely adopted. This paper presents Raptor, a SDSM cluster management system based on checkpoint/recovery and thread migration. Raptor decouples the runtime system and application data from application threads, allowing efficient load balancing, resource allocation, and rollback recovery. There are two important features of the system. First, it reduces checkpoint overhead by only saving application-specific data that cannot be recreated at recovery time. Second, by integrating thread migration capability both at runtime or recovery, it allows the addition or removal of computing resources from a running application while adding little or no additional burden on the SDSM application programmer.*

Index Terms—Cluster management, thread migration, checkpoints, distributed shared-memory.

<sup>§</sup> Corresponding author.

# Raptor: Integrating Checkpoints and Thread Migration for Cluster Management

## ABSTRACT

*Software distributed shared-memory (SDSM) provides the abstraction necessary to run shared-memory applications on cost-effective parallel platforms such as clusters of workstations. However, problems such as cluster component reliability and cluster management, which are not directly related to performance, need to be addressed before SDSM solutions can be widely adopted. This paper presents Raptor, a SDSM cluster management system based on checkpoint/recovery and thread migration. Raptor decouples the runtime system and application data from application threads, allowing efficient load balancing, resource allocation, and rollback recovery. There are two important features of the system. First, it reduces checkpoint overhead by only saving application-specific data that cannot be recreated at recovery time. Second, by integrating thread migration capability both at runtime or recovery, it allows the addition or removal of computing resources from a running application while adding little or no additional burden on the SDSM application programmer.*

## I. INTRODUCTION

Software distributed shared-memory (SDSM) systems provide a relatively inexpensive entry into large-scale parallel computing by leveraging commodity uniprocessor or SMP nodes, networks, and operating systems. Much research has been dedicated to solving the performance problems of SDSM systems; however, there are additional obstacles that prevent the widespread adoption of SDSM clusters in production environments. Clusters are less reliable than single systems, simply because they contain more components that can fail. This is a significant drawback because applications that benefit from large-scale systems are typically long running ones, increasing their chances of encountering a failure. In addition, since each system in the cluster has its own operating system, many functions that have traditionally been handled by the operating system, such as resource management and process scheduling, cannot be easily performed. Ideally, the SDSM runtime system should provide a framework that allows relatively simple job submission, resource management, and reliability.

Solving the reliability problem requires an efficient checkpoint and recovery mechanism. The resource management problem requires an efficient and fine-grain mechanism for moving computation among cluster nodes.

We describe the design, implementation, and evaluation of Raptor, a set of mechanisms to address the reliability and system management problems associated with shared-memory parallel computing clusters. These mechanisms are possible because of the synergistic benefits of integrating thread migration and checkpoint/recovery in the following way. Our checkpoint mechanism decouples data checkpoints from computation (or thread) state, allowing the redistribution of a process' threads across multiple nodes when needed. In addition, it is possible to recover multiple process checkpoints in a single process. These features of Raptor enable the recovery of a failed SDSM application on a smaller cluster, the redistribution of load among existing nodes during parallel execution, and the addition of new nodes when more resources become available. Raptor's design objective is to enable Condor-like capability [15] for SDSM jobs that may be running on systems shared by other users. The set of mechanisms described in this paper enable such functionality. Raptor has been implemented in the context of the MyDSM\* software distributed shared memory system.

To our knowledge, there has been no prior 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., [5-7, 13, 14]) and thread migration (e.g., [10, 26, 28]) separately for SDSM systems. Previous work on adaptive clusters either requires the use of a specialized language [18], the ability to repartition computation at runtime [22], or more expensive process migration mechanisms that impose memory overheads [4, 22, 25].

The rest of the paper is organized as follows. Section II details our thread migration and checkpoint/recovery implementations, including their performance on a set of well-known shared memory applications. Sections III and IV describe the implementation of node removal, automatic single-node recovery, and online node addition. Programming environment issues are discussed in Section V. Related work is discussed in Section VI. Conclusions are presented in Section VII.

## II. THREAD MIGRATION AND CHECKPOINTS

Raptor's key feature is the tight integration of checkpoints and thread migration. In this section, we present a brief overview of our implementation. A more detailed presentation may be found in [1].

---

\* "MyDSM" is a pseudonym to protect the identities of the authors.

### A. Thread Migration

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) [21] intended to contain thread-private data. 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 "shell". Shell threads are created by the runtime system during initialization as described below.

To avoid stack relocation errors, we ensure that the thread stacks at all participating nodes 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 MyDSM 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. This wastes an insignificant amount of resources when fewer threads are active, as each suspended thread costs only a single page of committed stack storage by default and the OS structures required for managing it. 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. Thread context and stack information is discovered by a combination of Win32 calls and access to user accessible (though scarcely documented) OS structures. Correctness issues that arise due to memory ordering models or SDSM system coherence mechanisms were addressed as described in [1].

File I/O operations in the presence of thread migration are challenging. In MyDSM, the Win32 calls that access files are intercepted by Detours [9] 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, file pointer location etc.). The file sharing mode is altered when necessary to ensure that files may be reopened at another node. This information is then transmitted to the new node as part of the thread migration message and used to reopen the appropriate files and to set their file pointers appropriately. Raptor addresses such issues as ensuring that file handles have the same semantics on both systems and flushing files before migration.

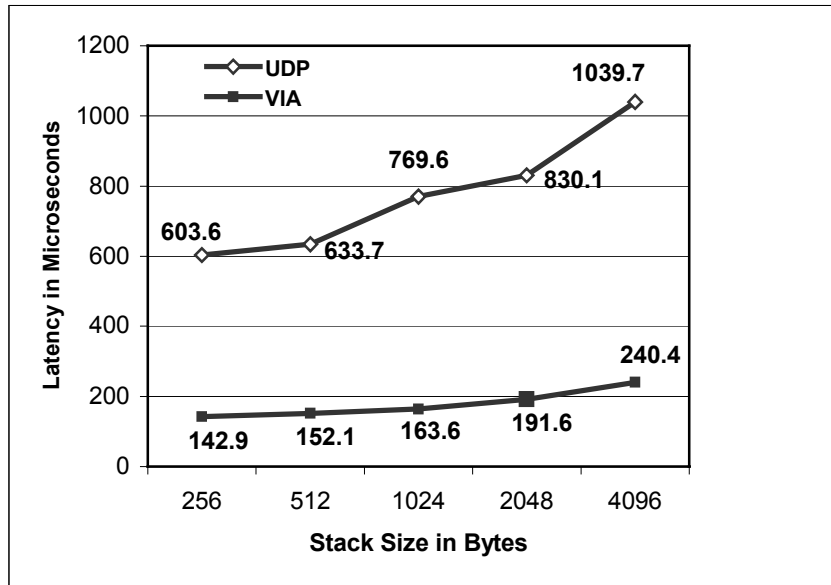
Migrating threads in MyDSM may be performed either explicitly by application programmers or automatically as part of the Raptor extensions. Since Raptor does not require that programmers alter existing applications, the

runtime system automatically performs thread migration when required at barriers. For those few applications that do not include barrier synchronization, providing thread migration capability is the subject of ongoing work.

### B. Performance of Thread Migration

The performance measurements presented throughout this paper were gathered on a network of 4-way SMP 500MHz Pentium III Xeon systems running Windows NT SP5 and connected by both Fast Ethernet and Emulex VIA-based NICs.

We measured the performance of thread migration by executing 1000 back-to-back thread migrations between two nodes while varying the thread's stack size. At the receiver, an acknowledgement message is sent back to the originating node for reliability purposes in the UDP implementation. This acknowledgement was not required for the reliable VIA measurements.



**Figure 1: Thread migration latency vs. stack size for UDP and VIA.**

The average thread migration latency was 1.04 ms for UDP on Fast Ethernet and 240.4  $\mu$ sec on Emulex VIA, both using 4KB stacks (see Figure 1). This latency is comprised of the Windows NT overheads to acquire the thread's context and stack, all required synchronization, stack copies, and network messages. The figure also shows the effect of varying the stack size on thread migration latency for both UDP and VIA. As expected, the thread migration latency increases with larger stack sizes. The cost of a thread migration using 4KB stacks in MyDSM is

very similar to the cost of fetching a single 4KB page in the SDSM system, which is 0.937 ms and 0.279 ms for UDP and VIA, respectively. Our thread migration implementation may be used to reduce communication costs in SDSM systems similar to [10, 26] by moving computation to data, although this is not a focus of this work. If the cost of thread migration is low, there will be more opportunities to improve performance by reducing data communication.

### *C. Checkpoint and recovery*

Checkpoints in Raptor support rollback recovery of a distributed process. Each node in the system performs a consistent checkpoint, and all such checkpoints are used to recover a failed MyDSM application. In addition, there are three important features of Raptor checkpoints. First, there is no inherent association between data, computation threads, and cluster configuration in the checkpoint and recovery process. This is important as it allows additional functionality during recovery. The thread migration support in Raptor is a crucial component of this feature. In fact, computation checkpoints are treated just like thread migration operations. Second, to minimize the runtime overhead of creating checkpoints, disk I/O is largely removed from the critical path. Third, checkpoints may be initiated automatically using time intervals or explicitly under program control. The latter is desirable if application programmers can leverage information about program behavior to reduce runtime overheads.

In Raptor, each application requires a configuration file that specifies the number of nodes required as well as the desired (but not guaranteed) initial distribution of threads among nodes for a parallel application. The runtime system is responsible for creating the shared-memory pool, network connections, thread migration, and checkpoint threads. Since our goal is to provide flexible recovery support with cluster reconfiguration, we save only application data, some file I/O information, and computation threads. All MyDSM nodes must have access to at least one shared file system to access the program executable. The common file system is also used for application data and checkpoint files.

One of the main challenges of creating data checkpoints in a SDSM system is deciding the point at which shared-memory pages, runtime system management structures, and application threads are in a stable state. It is difficult to create checkpoints while coherence activities (e.g., invalidations, page-fill operations, etc.) are in process due to the need for there to be a “coherent snapshot” from which to recover the state of the system in the event of a failure.

To avoid these issues, checkpoints are created at barrier synchronization points. Checkpoints store all application shared-memory data (including all runtime system structures that maintain page states). To minimize the amount of data saved, page guarding is used to identify pages that have been allocated but never accessed. Such pages are excluded from the checkpoint to reduce the size of the checkpoint file. Although it may be possible to reset the guarded state of pages between checkpoints to enable incremental or log-based implementations, we currently perform full checkpoints. Finally, some MyDSM runtime system structures were modified to avoid saving redundant information kept in the page state structures that may be recreated during recovery. Computation and file I/O states are saved in a manner that is almost identical to the thread migration facility. During recovery, which also occurs at barriers, each node opens its checkpoint file and applies the contents to the local copy of all shared-memory pages, setting page protections as necessary. Thread contexts are set and stacks are copied before application threads are awakened and allowed to exit from the recovery barrier. At that point, rollback recovery is completed.

Checkpoint creation and recovery is managed by a Checkpoint Agent thread that is created during runtime system initialization. This thread is awakened when all threads arrive at a checkpoint barrier. These barriers are flagged either explicitly by the programmer, automatically by a timer-based scheme, or by external request (e.g., from a cluster management console). Checkpoint Agents are also responsible for recovery operations. During recovery, the checkpoint agent checks the integrity of the checkpoint file by comparing its “table of contents” information to its size. This is necessary to identify problems introduced by the file I/O overhead reduction techniques described next.

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 address the first issue as follows. Each node creates its checkpoint file locally to maximize file system throughput. The best file system write throughput was achieved using the sequential scan file access mode while allowing file system buffering mechanisms to remain active, although previous work has indicated that good write throughput can also be achieved using hardware write caches [20] (the disk drives used in our experimental platform do not support hardware write caching). Since node failures can render access to local files impossible, it is necessary to copy the checkpoint files to remote file servers

accessible by all nodes. To reduce checkpoint creation overheads, all of these steps must 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. The responsibility for I/O completion therefore lies with the file system. We chose this scheme because it allows the overlap of computation with I/O operations, resulting in a substantial reduction in checkpoint overhead compared to a previous implementation [1]. Once the local checkpoint is closed and application threads are resumed, the Checkpoint Agent copies the checkpoint file to a network file system before accepting any new requests for checkpoints. To avoid losing a checkpoint due to failure during I/O operations, the runtime system retains two checkpoint files per node and alternates between them as new checkpoints are created. By comparing the size of a checkpoint file against its header information, we detect corrupt or incomplete files during recovery and revert to the older checkpoint. Raptor configuration files allow users to require file I/O completion prior to exiting the checkpoint barrier if they are willing to accept the performance penalty.

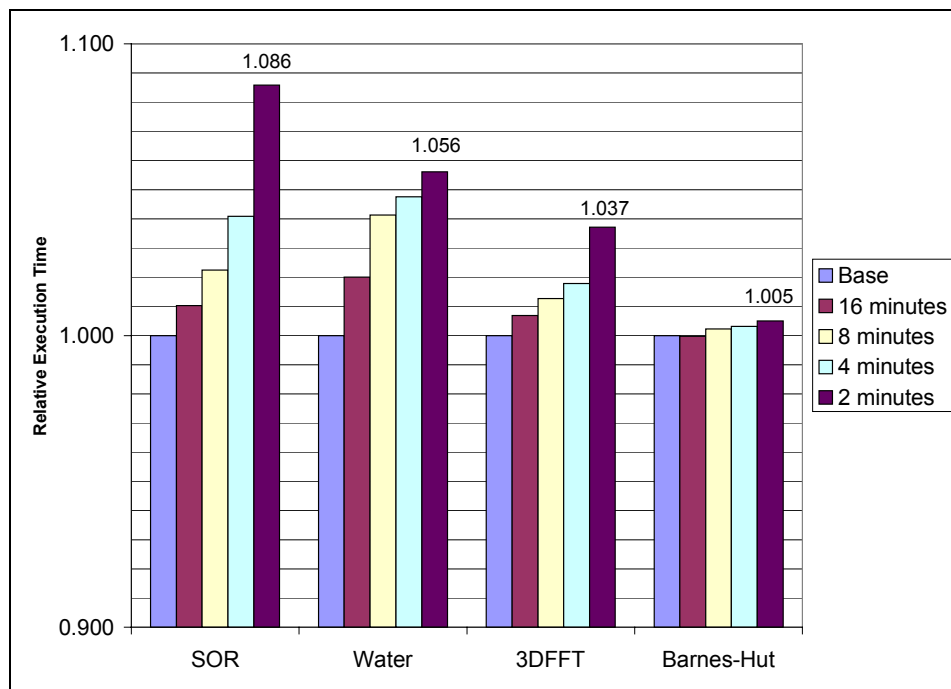
*D. Checkpoint Performance*

	SOR	Water	3DFFT	Barnes-Hut
Problem Size:	4000×4000	4096 Mol	$2^6 \times 2^7 \times 2^6$	32K Bodies
Iterations:	10000	40	1500	500
Processors:	16	16	16	16
Nodes:	4	4	4	4
Threads/Processor:	1	1	1	1
Shared Memory Size	122 MB	45 MB	88 MB	12 MB

**Figure 2: Benchmark applications**

Four parallel applications (see Figure 2) were used to measure the overhead of the checkpoint facility: SOR (successive over-relaxation); Water from SPLASH [23]; 3DFFT from the NAS benchmarks [2]; and Barnes-Hut also from SPLASH. These applications are representative of typical scientific shared-memory parallel applications that may perform well on SDSM systems. This set includes applications with both regular and irregular data access patterns, which can affect the amount of coherence data generated by the runtime system and stored in checkpoints.

These applications have varying shared-memory footprints and are all iteration-based, which allowed us to increase their execution times arbitrarily to include enough data points in our evaluation. All of the measurements were performed on a 4-node, 16-processor cluster interconnected using VIA. Checkpoint files were written to local disks and copied to a network file server over Fast Ethernet. 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 3 shows the execution time overhead of generating checkpoints at the respective intervals compared with the base execution time with checkpoints disabled.



**Figure 3: Checkpoint execution time overhead vs. checkpoint interval.**

As indicated in Figure 3, the overhead of generating checkpoints was less than 9% of the total execution time for all applications using 2-minute intervals. The overheads are considerably lower when more reasonable intervals are used. Figure 4 provides additional parameters that are useful for the performance evaluation. First, 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. The table also shows the total recovery time for each of the four applications, both with and without runtime system and application initialization. Including initialization, the recovery times 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 between checkpoints, the applications will spend some time (half of the checkpoint interval on average) to reach the state that they were in before failure occurred. Previous work has shown that recovering logs can also be time consuming (refer to Section VI). Two points are worth noting regarding the growth of checkpoint overheads with reduced intervals. First, the amount of data saved is a function of the barrier instance at which checkpoints take place. For applications that have few computation phases, checkpoints are typically similar in size, resulting in near linear growth of checkpoint overheads. Second, the number of checkpoints taken does not always grow by the interval ratios because even if the interval has expired, the application has to reach the next barrier before a checkpoint is taken.

	SOR	Water	3DFFT	Barnes-Hut
Recovery from Network Disks	33.83 sec.	14.39 sec.	18.69 sec.	3.21 sec.
Total Recovery Time	43.33 sec.	17.21 sec.	22.24 sec.	7.64 sec.
Bandwidth seen during recovery	3.65 MB/s	3.42 MB/s	4.84 MB/s	3.87 MB/s
Average Checkpoint Size	76.95 MB	32.68 MB	34.77 MB	6.64 MB
Shared Memory Total Size	122 MB	45 MB	88 MB	12 MB
Process Size	312 MB	248 MB	226 MB	198 MB
Checkpoint vs. Process Size	24.66 %	13.18 %	15.38 %	3.35 %

**Figure 4: Checkpoint and Recovery Statistics.**

Raptor performs recovery in two situations. It recovers applications when systems fail or when applications have to be restarted due to rescheduling or resource availability. The distributed nature of SDSM applications requires flexible mechanisms to redistribute computation as needed to react to cluster utilization and availability. The following sections describe the means by which Raptor addresses these issues.

### III. NODE REMOVAL

In this section, we describe three mechanisms: node removal using checkpoint and recovery, automatic single node failure detection and recovery, and online node removal. Node removal using checkpoint/recovery 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 recovery on the remaining nodes from single node failures without any user intervention. Finally, the ability to reconfigure parallel applications to use fewer resources is a desirable feature in a shared cluster 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.

#### *A. Node Removal Using Checkpoint/Recovery*

The integration of thread migration within our checkpoint/recovery implementation allows us to recover a node's data and threads independently and optionally on different nodes. The only extra functionality required is the processing to ensure that all data at the removed node are applied in a manner consistent with the runtime system's coherence mechanisms. The alternative to thread migration would require migrating the process and re-creating it on one of the existing nodes [22]. 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 in the absence of replacement nodes 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 termination after data migration is performed as an alternative to process migration, then the application has to be repartitioned on-the-fly because the computation threads of the dead process cannot be adopted by the remaining nodes.

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. Recovering the memory contents of the removed node is a somewhat complex process. The runtime system must identify the shared 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 were modified on the removed node, the runtime system applies those changes to

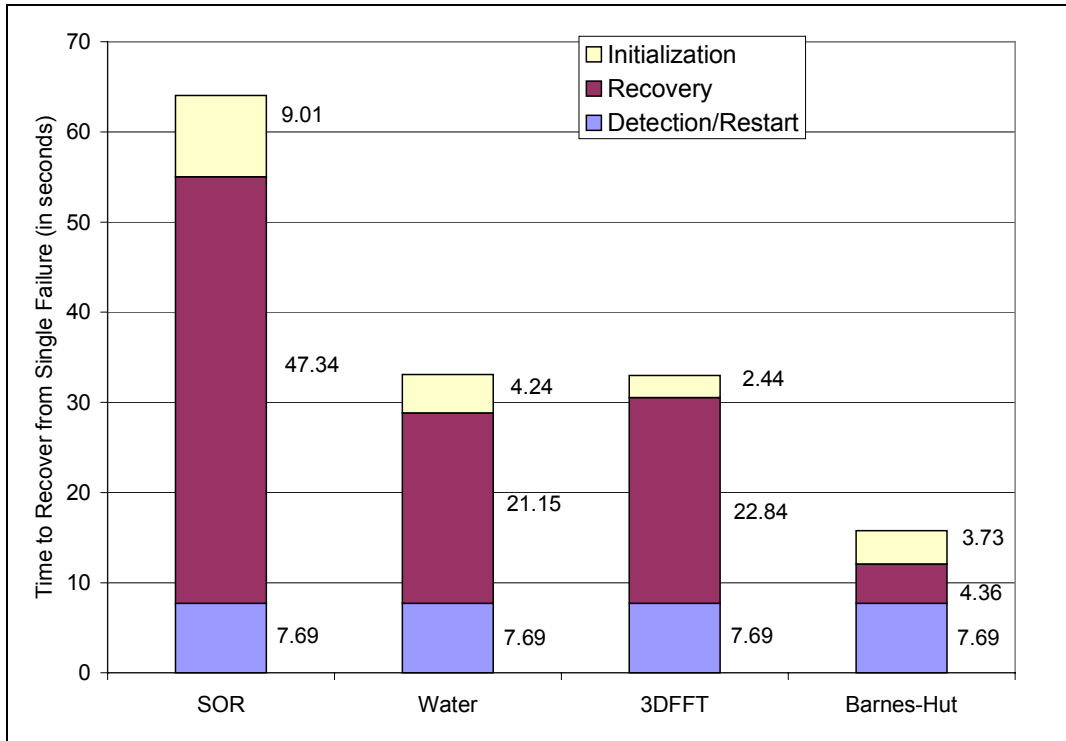
other copies of the page that reside on other nodes to bring their shared page data up-to-date with respect to the removed node. Thread recovery during node removal is simple. The threads of the removed node are distributed in a round-robin fashion on our prototype implementation. This is accomplished by performing a thread migration using the removed node checkpoint file as the source of the thread(s). A more sophisticated thread distribution heuristic that uses load and system configuration information (possibly through querying an extended version of the MyDSM remote access service) can be employed to optimize this operation. In such an implementation, the system will query the MyDSM services on cluster nodes to determine a distribution of threads that maximizes the performance of the application to be recovered without degrading running applications if possible. The current implementation can result in overloading certain nodes with threads while leaving processors on other nodes idle. The performance of node removal using checkpoints will be discussed within the context of automatic recovery.

### *B. Automatic Recovery*

We implemented an automatic recovery mechanism for single node failures. Several new capabilities were required. First, a failure detection mechanism is needed. Second, once a failed node is identified, the remaining nodes must be shutdown and restarted from the last checkpoint without user intervention. In addition, the current implementation relies on a single node to perform the failure detection and to initiate the automatic recovery process.

The Raptor prototype relies on heartbeats for failure detection. More robust communication mechanisms (e.g., ISIS [3]) should be used in production environments. To minimize the number of messages required to detect failures, we employ multicast heartbeats when available (e.g., in the UDP implementation), 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 Heartbeat Thread that is responsible for sending multicast heartbeat messages at administrator-defined intervals (10 seconds by default). If all replies are not received within a specified timeout period, the heartbeat message is resent to the non-responding node(s). A node that does not respond after two retries is identified as failed. 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 MyDSM service (daemon) on the first

node in the configuration file. The application restarts by performing recovery on the remaining nodes, recovering the data and thread(s) of the failed node as described in the previous section.



**Figure 5: Performance of Checkpoint-Based Node Removal.**

To measure the performance of node removal, we ran the benchmarks on four systems, each running four threads while enabling automatic failure detection. We then intentionally terminated a process running on one of the systems and measured the time required to recover the application on the remaining nodes. We also measured the time it takes to detect the failure and start the recovery process. We average measured failure-to-recovery initiation time 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 5, which shows the total single failure recovery time subdivided into the detection/restart, checkpoint recovery, and runtime system/application initialization components for each application.

### C. Online Node Removal

Online node removal may be initiated manually by system users, automatically as a reaction to a system logout or shutdown event, or by an UPS (uninterruptible power supply) upon power failure detection. The key to the success

of this implementation is the ability to migrate application threads and data from a removed node to the remainder of the cluster. Online node removal involves three components: a mechanism for initiating node removal, a means of moving data to other nodes while preserving coherence, and a way of distributing threads to the remaining nodes. Our design objective was to implement an efficient mechanism that can handle all the situations in which a node might be removed from the system. All parallel application programs that run on MyDSM 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, including user interrupts such as when users hit the Control-C keyboard combination, as well as shutdown and logout events. Online node removal can be initiated when one of these events is detected and will postpone shutdowns or logouts until removal is complete. On power failures, node removal can be initiated by a UPS-triggered shutdown.

Once the event handler sets the removal request flag, node removal proceeds as follows. At the next application barrier, 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. 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. The node being removed sends its page status blocks, data pages, and any page modifications 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. Each destination node performs recovery actions that are identical to those performed for checkpoint-based node removal. Once all data migration messages complete, a node removal completion message is sent to the remaining nodes, allowing all application threads to proceed.

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

**Figure 6: Online node removal performance statistics.**

The performance of online node removal was measured by running the four benchmark applications on four nodes with four threads each (a total of 16 threads). We then manually issued an online node removal request. Figure 6 shows the total online removal time ranged between 0.52 and 2.22 seconds. These numbers include the total time elapsed from the arrival at the removal barrier until all data and thread migration activity completes. The bulk of this time is spent migrating data. The figure also shows some useful statistics on network utilization, average bytes transferred per page, etc.

#### IV. NODE ADDITION

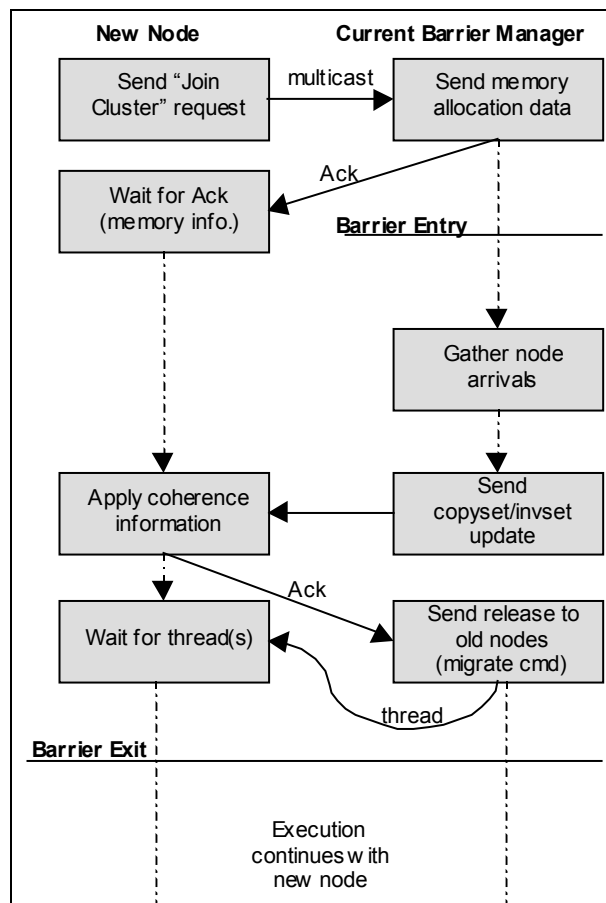
The integration of thread migration in MyDSM allows new or replacement computing resources to be brought on-line with minimal interruption of running applications. After node removal, processors at the remaining nodes may be overloaded with computation threads, and node addition can be used to improve performance in such cases when more resources become available. In addition, an application may be initially started on a cluster with a number of threads that exceeds the number of available processors in anticipation of future resource availability. In principle, node addition in MyDSM only requires the migration of threads from existing nodes since shared data is automatically migrated by the SDSM runtime system when needed. To ensure that coherence is not violated, the necessary coherence state information is sent to the new node before it receives the computation threads. Finally,

before joining an existing cluster, the new node initializes the runtime system and any read-only application variables. These operations are performed off-line without interrupting the running application.

After the new process has completed the application and shared memory initialization, the process sends a “Join Cluster” request to the existing nodes and waits for an acknowledgement. A designated manager node responds with a message that includes shared-memory information usually exchanged during shared-memory allocation in normal SDSM runtime system operation. At the following application barrier, which we refer to as the *join barrier*, the 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 acknowledgement to the 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) to the new node. This process is summarized in Figure 7.

#### *A. Performance of Node Addition*

We measured the performance of node addition by running the four benchmarks with 3 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 arrives 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



**Figure 7: Node addition process (after connection establishment).**

Figure 8 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.

	SOR	Water	3DFFT	Barnes-Hut
Dataset Used	4000×4000	4096 Mol	$2^6 \times 2^7 \times 2^6$	32K Bodies
Node Addition Total	0.3223 sec.	0.4486 sec.	2.1188 sec.	2.7734 sec.
Addition Init. Time	0.0023 sec.	0.0100 sec.	0.0008 sec.	1.3234 sec.

**Figure 8: Performance of node addition.**

## V. PROGRAMMING MODEL

We discuss programming related issues to enable Raptor functionality in this section. Applications written for MyDSM are linked with the runtime system's library. The runtime system includes all support for transparent checkpoints, Raptor-initiated thread migration, recovery, node addition, and removal. There are two simple requirements that applications have to comply with in order to benefit from Raptor functionality. First, since recovery depends on rerunning the application's initialization code, we require a barrier to exist between the end of initialization and the start of the parallel execution section. This was already the case for the four applications discussed here. During recovery, this barrier initiates the runtime system's recovery process. Second, for thread migration, all data that may be modified after initialization has to be in shared-memory to ensure accessibility from any node if threads are migrated. This is also commonly done by application programmers and does not result in a performance penalty unless there is sharing. Such sharing will occur when a thread is migrated and needs to access otherwise private data that remained on the original node. The runtime system automatically handles data migration.

It is important to observe that the environment in which Raptor is likely to be used is one in which programmers are probably willing to recompile and slightly modify code in order to benefit from the extra functionality. There is plenty of anecdotal evidence at national laboratories and research institutions about the extent to which scientists and programmers are willing to modify their code for reliability. Our requirements are very simple and often do not require any modifications to application code.

## VI. RELATED WORK

To the best of our knowledge, there has been no previous work that combines and integrates thread migration and checkpoint/recovery to the extent presented here; however, there are numerous previous efforts that are related. Several other SDSM systems utilize thread migration, but not within checkpoints and recovery mechanisms [10, 16, 17, 26]. Our thread migration latencies are better than most reported systems, although it is difficult to make objective comparisons due to differences in platforms used for the various implementations.

There are numerous checkpoint/recovery implementations on SDSM systems [5, 7, 11, 14, 27]. None of these implementations integrate thread migration functionality during checkpoint or recovery. There are several checkpoint facilities for Windows NT processes [8, 24], although none of these work in a SDSM system. Huang et

al. implement a recovery mechanism called NT-SwiFT [8]. It includes the *Winckp* library that can be used for rollback-recovery of NT applications. Srouji et al. [24] implemented a general-purpose checkpoint facility for non-distributed multi-threaded Windows NT processes.

A study on the TreadMarks [12] SDSM system describes an implementation that uses process checkpoint and migration facilities to support adaptive and configurable cluster-based computing using OpenMP [22]. Their implementation uses an existing process checkpoint facility called *libckpt* [19] to checkpoint and recover SDSM 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, it is possible to 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 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. Using thread migration and decoupling of data and computation in checkpoints provides a substantially more efficient means of achieving the same objectives without requiring special programming environment. Thus, we believe that Raptor's approach is simpler and more efficient for cluster management applications.

## VII. CONCLUSIONS

We have described the design and implementation of Raptor, a system that integrates efficient user-level thread migration and checkpoint/recovery mechanisms. Additionally, we have shown that the integration of these two mechanisms facilitates the online addition and removal of computational nodes with minimal disruption to running applications. Performance numbers measured on a real system demonstrate 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 the decoupling of data and computation in checkpoints, and runtime support for thread migration in SDSM systems enable all the functionality required for managing shared parallel computing cluster resources while providing efficient support for reliability, job scheduling, and load-balancing.

## REFERENCES

- [1] Anonymous. *Reference omitted to preserve the anonymity of the author(s)*.
- [2] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS Parallel Benchmarks. NASA Ames RNR-91-002, August 1991.
- [3] Kenneth Birman and Thomas Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, vol. 5, pp. 47-76, 1987.
- [4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, Y. C. E. Zhou, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. *ACM SIGPLAN Notices*, vol. 30(8), pp. 207-216, 1995.
- [5] G. Cabillic, G. Muller, and I. Puaut. The Performance of Consistent Checkpointing in Distributed Shared Memory Systems. *Proceedings of the 14th Symposium on Reliable Distributed Systems*, pp. 96-105, September 1995.
- [6] G. Cabillic, T. Priol, and I. Puaut. MYOAN: An Implementation of the KOAN Shared Virtual Memory on the Intel Paragon. IRISA, Research Report 812, March 1994.
- [7] Manuel Costa, Paulo Guedes, Manuel Sequeira, Nuno Neves, and Miguel Castro. Lightweight Logging for Lazy Release Consistent Distributed Shared Memory. *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pp. 59-73, October 1996.
- [8] Yennun Huang, P. Emerald Chung, Chandra Kintala, Chung-Yih Wang, and De-Ron Liang. NT-SwiFT: Software Implemented Fault Tolerance on Windows NT. *Proceedings of the 2nd USENIX Windows NT Symposium*, pp. 47-55, August 1998.
- [9] Galen Hunt and Doug Brubacher. Detours: Binary Interception of Win32 Functions. *Proceedings of the 3rd USENIX Windows NT Symposium*, pp. 135-144, July 1999.
- [10] Ayal Itzkovitz, Assaf Shuster, and Lea Shalev. Thread Migration and its Applications in Distributed Shared Memory Systems. *Journal of Systems and Software*, vol. 42, pp. 71-87, 1998.
- [11] P. Keleher. Lazy Release Consistency for Distributed Shared Memory. Ph.D. dissertation, Department of Computer Science, Rice University, Houston, TX, 1995.
- [12] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. *Proceedings of the 1994 USENIX Winter Technical Conference*, pp. 115-131, January 1994.
- [13] A. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, and I. Puaut. A Recoverable Distributed Shared Memory: Integrating Coherence and Recoverability. *Proceedings of the 25th International Symposium on Fault-Tolerant Computing Systems*, pp. 289-298, June 1995.
- [14] A.-M. Kermarrec, C. Morin, and M. Banatre. Design, Implementation and Evaluation of ICARE: An Efficient Recoverable DSM. *Software Practice and Experience*, vol. 28(9), pp. 981-1010, 1998.
- [15] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor: A Hunter of Idle Workstations. *Proceedings of the 8th International Conference on Distributed Computing Systems*, pp. 104-111, June 1988.
- [16] Edward Mascarenhas and Vernon Rego. Ariadne: Architecture of a Portable Threads System Supporting Thread Migration. *Software - Practice and Experience*, vol. 26, pp. 327-356, 1996.

- [17] Scott Milton. Thread Migration in Distributed Memory Multicomputers. The Australian National University, Technical Report TR-CS-98-01, February 1998.
- [18] OpenMP Architecture Review Board. *OpenMP C and C++ Application Program Interface*. Version 2.0 ed., 2002.
- [19] J. S. Plank, M. Beck, G. Kingsley, and Kai Li. Libckpt: Transparent Checkpointing under Unix. *Proceedings of the USENIX Winter 1995 Technical Conference*, pp., January 1995.
- [20] Erik Reidel, Catharine van Ingen, and Jim Gray. A Performance Study of Sequential I/O on Windows NT 4. *Proceedings of the 2nd USENIX Windows NT Symposium*, pp. 1-10, August 1998.
- [21] Jeffrey Richter. *Advanced Windows*. 3rd ed., Microsoft Press, 1997.
- [22] Alex Scherer, Honghui Lu, Thomas Gross, and Willy Zwaenepoel. Transparent Adaptive Parallelism on NOWs using OpenMP. *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 96-106, May 1999.
- [23] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Stanford University, Technical Report CSL-TR-91-469, April 1991.
- [24] Johnny Srouji, Paul Schuster, Maury Bach, and Yulik Kuzmin. A Transparent Checkpoint Facility on NT. *Proceedings of the 2nd USENIX Windows NT Symposium*, pp. 77-85, August 1998.
- [25] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, vol. 2(4), pp. 315-339, 1990.
- [26] Kritchalach Thitikamol and Pete Keleher. Thread Migration and Communication Minimization in DSM Systems. *Proceedings of the IEEE*, vol. 87(3), pp. 487-497, 1999.
- [27] N. H. Vaidya. A Case for Two-Level Distributed Recovery Schemes. *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 64-73, May 1995.
- [28] Boris Weissman, Benedict Gomes, Jurgen Quittek, and Michael Holtkamp. Efficient Fine-Grain Thread Migration with Active Threads. *Proceedings of the 12th International Parallel Processing Symposium*, pp. 410-414, March 1998.