Fast and Transparent Recovery for Continuous Availability of Cluster-based Servers

Rosalia Christodoulopoulou
Department of Computer Science, University of Toronto, Canada
roza@cs.toronto.edu

Kaloian Manassiev
Department of Computer Science, University of Toronto, Canada
kaloianm@cs.toronto.edu

Angelos Bilas
Department of Computer Science, University of Crete, Greece
bilas@csd.uoc.gr

Cristiana Amza
Department of Electrical and Computer Engineering, University of Toronto, Canada
amza@eecg.toronto.edu

Abstract

Recently there has been renewed interest in building reliable servers that support continuous application operation. Besides maintaining system state consistent after a failure, one of the main challenges in achieving continuous operation is to provide fast reconfiguration. The complexity of the failure reconfiguration mechanisms employed and their overheads depend on the type of platform that is being used as a server and the types of applications that need to be supported. In this paper we focus on providing support for shared-memory applications running on clusters of commodity nodes and interconnects. Achieving continuous operation for shared memory applications on clusters presents two main challenges. (a) The fault tolerance mechanisms employed should be *transparent* to applications and should have low overhead during failure-free execution. (b) When failures occur, reconfiguration should occur with minimum application disruption without requiring the full recovery of the failed node.

In this work we examine in detail the latter, i.e., (b), the failure reconfiguration path. We use a previously developed system [8] that achieves (a) by using dynamic replication of data to the memories of multiple nodes of the system during execution. We examine in detail how the runtime system can achieve minimum application interruption, when failures occur. We present the design and implementation of *FineFRC* (*Fine-grained Failure Reconfiguration on Clusters*), a runtime system for achieving continuous operation of shared memory applications on commodity clusters without requiring application instrumentation or human intervention. We present results using a working, 16-processor system that achieves subsecond failure reconfiguration times.

Categories and Subject Descriptors D.1.3 [*PROGRAMMING TECHNIQUES*]: Concurrent Programming—Distributed programming

General Terms Measurement,Performance,Experimentation

Keywords fault tolerance, fast failure reconfiguration, distributed shared memory, scalability, availability

PPoPP'06 March 29–31, 2006, New York, New York, USA.

1. Introduction

With the continued growth and use of the Internet, ranging from on-line multiplayer games to on-line transaction processing, it becomes challenging to meet expectations and requirements for server scaling and virtually perfect availability. For instance, in many cases, central servers that serve more and more online customers are required to operate continuously (24x7) with availability of at least 99.999%. These trends have prompted a departure from the traditional "treat failures as the unusual case" philosophy and researchers are encouraged to focus on tools and techniques to improve the performance of the recovery path, since failures should be treated as a fact of life.

Our goal is to completely hide component failures from applications that run on scalable servers. Although a range of different architectures are currently used as server systems, in this work we focus on providing support for generic shared-memory applications running on clusters of commodity nodes and interconnects. In order to provide applications with the illusion of continuous availability in this setup, there are two main challenges: (a) During failure-free execution, the system should be able to gather and maintain all required state transparently and with low overhead. (b) When failures occur, server recovery and reconfiguration should be both *transparent* and *fast*.

Our previous work [8] deals with (a) by supporting dynamic replication of all in-memory data to multiple nodes. In this previous work, we report overheads of implementing fault tolerance schemes during normal operation, but not the delays during recovery from a failure. Thus, similar to most other research work in this area, our previous work has focused on various techniques for minimizing the overheads during failure-free execution. Most of these previous research efforts follow two alternative approaches: (i) disk logging and periodic checkpointing mechanisms [13, 10, 27] and (ii) fast, in-memory failover techniques [1, 31]. In the former approach, fast recovery is limited by reloading the previous consistent checkpoint from disk and replaying the logs. In the latter approach, although the backup maintains a copy of the application state, consistency is not strictly enforced. Hence, the backup is either idle during failure

free system execution [1, 31] or could execute a different set of applications/tasks.

The recovery path has received far less attention, mainly because handling failures is in practice complex, involving many corner cases. In particular, implementing and evaluating a recovery scheme is challenging for shared memory applications that share information at fine granularities and time intervals. Thus, in practice, failure reconfiguration for many approaches is not completely automated and requires some type of human intervention.

Our previous work [8] presents a fault tolerance approach that achieves low overhead by using dynamic replication of application and system state during system operation. Our approach can be summarized as follows. We transparently capture all in-memory application state modifications by means of a shared virtual memory system, *GeNIMA* [5] and we maintain consistency of the in-memory state through dynamic replication on multiple nodes. All nodes in a cluster are used for running a single copy of the server application. To achieve low recovery overhead, we checkpoint system and application state at each system node. Checkpoints are taken in a coordinated manner between threads in each system node, but *asynchonously* across the nodes in the cluster in order to preserve scalability. Each node behaves as two virtual nodes, a primary and a backup. Each primary stores its checkpoint information into the memory of its corresponding backup node.

In this work, we present *FineFRC* (*Fine-grained Failure Reconfiguration on Clusters*), a system that builds on our previous work [8] and automatically reconfigures itself upon failures in the above context of shared memory applications on commodity clusters. *FineFRC* provides support for transparently recovering all in-memory state at low overhead. Our system is able to recover and reconfigure itself after any number of successive, single-node failures, under a fail-stop model. The recovery path involves mostly restarting the failed threads on a backup node from their last respective checkpoint saved in the backup's memory. The restarted threads see the consistent, in-memory state on the backup corresponding to the last checkpoint taken on the failed node. Fast recovery relies on the fact that our checkpointing mechanism maintains system state consistent at fine-grain intervals. Overall, the features of *FineFRC* are:

- It provides transparent support for shared memory applications running on commodity clusters of SMPs interconnected with low-latency, high-bandwidth interconnects.
- It supports multithreading within each cluster node, thus taking advantage of multiple per-node CPUs.
- There is no static division of nodes in primaries and secondaries, resulting in better system utilization for a single application.
- It achieves fast failure reconfiguration through fine-grained in-memory state replication and reconstruction.

The main challenge in our approach stems from the requirement for *atomic* updates to global shared memory state. This requirement is hard to enforce if fast reconfiguration upon failures is also desired. Specifically, the absence of the detailed recovery logs used in traditional recoverable systems [4] is profitable for fast failure reconfiguration reasons. On the other hand, this complicates the design of the recovery phases for automatically reconstituting state consistency after a failure. In our work, we first examine in detail the recovery path in the system. We present the data structures, mechanisms, and actions required for recovering from failures. We build a working prototype that is able to recover from failures injected into the system while running generic applications. We run experiments on a 16-processor (8-node) system. This is, to the best of our knowledge, the first 16-processor prototype of a shared memory cluster that supports transparent system reconfiguration upon failures and is being demonstrated experimentally to provide fast recovery times. Finally, we use our prototype to measure the overheads associated with the recovery path.

In our evaluation we consider providing reliability and fast reconfiguration on recovery for two very different types of cluster servers: A generic compute server cluster, e.g., as used in Grid/-parallel computing for scientific applications and an in-memory transaction processing server cluster. As benchmarks for these architectures, we consider five Splash benchmarks for the compute server [14, 30] and the Vista transaction server [20] running the Vista benchmark suite for the in-memory transactional processing server. Our experimental results show that:

1. The average failure reconfiguration time is minimal (subsecond) for a range of experiments where we inject faults at different stages of server execution, including worst case scenarios.
2. Avoiding extensive application or system data undo or redo upon recovery is key for providing fast recovery.
3. The overhead of dynamic data replication and checkpointing during failure-free execution is 38% on average across all experiments we perform.

The rest of this paper is organized as follows: Section 2 presents the necessary background, focusing on the baseline distributed shared memory protocol and fault-tolerance enhancements that we build on for supporting transparent system recovery. Section 3 presents our system design and implementation for achieving subsecond recovery times. Sections 4 and 5 present our experimental setup, evaluation methodology, and results. Section 6 presents related work. Finally, we draw our conclusions in Section 7.

2. Protocol Background

In this section, we present the necessary background on our baseline distributed shared memory system, GeNIMA [5], and the failure-free case of the dynamic data replication scheme we use. A more detailed description of the latter can be found in [8].

2.1 GeNIMA

The base protocol we use is the GeNIMA [5] shared virtual memory system. GeNIMA is an invalidate-based protocol using lazy release consistency (LRC) for shared-memory accesses [15]. LRC centers around using synchronization operations present in the application (e.g., lock acquire and release pairs) in order to optimize update data transfers between processors for consistency maintainance. Application execution in each SMP node is partitioned into *time intervals*, deliniated by consecutive release operations that may be executed by any application threads in an SMP node. During each time interval all local page updates are recorded into a common *update-list*. When acquiring a synchronization object from a remote node, the local processor also fetches the list of updates which are needed for this synchronization step and *invalidates* the corresponding pages. A subsequent access to an invalidated page triggers a page fault that results in fetching the latest version of the page from its pre-assigned home node. Pages are fetched in their entirety.

One of the most important features of GeNIMA is its close integration with the system area network to reduce communication overheads. The most relevant techniques that we use in implementing our protocol are:

1. Automatic detection of writes to memory pages through memory protection violations [19].

2. Creation of *diffs*, which are encodings of the modifications to application state performed during a particular time interval [16].
3. Propagation of these updates for each shared memory page to a preset *home* node at the end of an interval [32].

2.2 Fine–grained Consistent Replication for Fault Tolerance

In our previous work [8], we extend the base GeNIMA protocol with fault tolerance mechanisms. These previous techniques build and maintain consistent replicas of system and application state during normal execution. In this section, we discuss the main design points of our approach.

We maintain two copies of (1) thread execution state, (2) application data state, and (3) protocol state for each node. Checkpoints of node state are initiated by our run-time system transparently to the application, whenever application updates of the node need to be made visible for consistency reasons to other nodes. Thus, the checkpoint interval at a node is the same as the consistency interval in the shared virtual memory protocol in our system. While the checkpoint interval is expected to be fine grained for most shared memory applications, extra checkpoints can be taken periodically if needed (e.g., for applications where consistency actions may not be necessary). Each checkpoint contains two phases: First, the execution state of each thread is saved remotely in the memory of the backup node. Second, the incremental data modifications since the previous checkpoint are sent out in the form of diffs. The two checkpoint phases should be conceptually atomic.

In order to allow the shared *application data* state to survive single node failures, we maintain application data replicas. The two issues related to replication of application state are (i) placement of the replicas and (ii) the way they are updated at checkpoints.

Each shared page is replicated at two distinct nodes called the primary and secondary home; if one copy of the page becomes inaccessible or corrupted due to a failure, then the other one can be used to allow the computation to continue. Shared page modifications performed by a node are propagated to both homes of the respective page, during regular consistency operations. The propagation of updates (diffs) is performed in two steps: first, all of the updates are propagated to the secondary homes and then to the primary homes. The two-phase scheme of update propagation ensures that all of the updates performed locally by a node within a given interval are replicated *atomically*. Because modifications are considered globally performed only after the first phase has been completed, the secondary home copy of a page is also called the tentative copy, while the primary home copy is called the committed copy.

A final, but simpler to manage, type of state is *protocol state*. Information about system resources such as shared locks, if any, is maintained in a similar fashion as application state. In particular, each runtime system resource, e.g., lock, is assigned a primary and a backup owner.

3. Achieving Fast Failure Reconfiguration

In this section, we present our approach for achieving sub-second recovery times. On-line reconfiguration in case of failures is based on the following principles: i) We ensure that during a checkpoint *both* application data state and local thread compute state are *atomically* replicated in the distributed memory of the cluster. (ii) After recovery, the in-memory state of the backup is consistent with the one in the failed primary and (iii) The failed threads are restarted on the backup node, so that the computation continues and completes successfully.

We assume that nodes are subject to *fail-stop* failures and we rely on the underlying communication layer to resolve transient network failures. Next, we describe each of these mechanisms followed by the description of the recovery path.

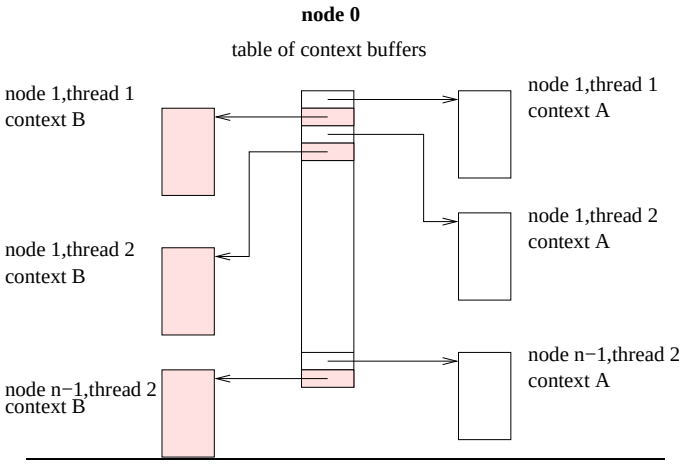

Figure 1. Example of exported address space for checkpointed thread contexts on node 0, assuming n nodes with two compute threads per node.

3.1 Thread State Checkpointing

Each active thread on the checkpointing node saves their local execution state remotely onto the memory of the backup node. A consistent snapshot of *all* thread state is needed to ensure atomicity of the compute checkpoint and data checkpoint, since the effects of each thread's execution have already been incorporated in the memory used by the application within the node.

A single thread's state consists of two pieces of data, the thread's execution context and its stack. A node's execution checkpoint consists of the saved context and stack of all of the compute threads running locally on that node, the *checkpoint timestamp* and other book-keeping information to assist recovery. Upon the end of a consistency interval in a node, the system creates a new checkpoint. Consistency intervals are deliniated either a) with a remote lock release performed by any of the local threads or b) with a global barrier. The data structures required for maintaining thread state replicas consistent are:

Context Buffers: During initialization, each node *exports* an address space, called *context buffer*, where a remote thread can save its checkpointed execution context. While initially each node serves as the backup node for only one other node, in the case of multiple successive failures, the maximum number of contexts for which a node can serve as a backup, corresponds to the number of compute threads on all nodes but one, itself (see Figure 1). Thus, to handle multiple successive failures, a node statically allocates space for up to as many contexts as the number of compute threads on all nodes but one. Each remote thread uses *two* context buffers, alternating bewteen them when saving its context, in order to handle the case of failure *while* checkpointing.

Stack Buffers: Similarly to contexts, during initialization, each node allocates a portion of its address space, called *stack buffer*, where remote threads can save their checkpointed stack.

Checkpoint Timestamp: Each checkpoint is assigned a timestamp. The timestamp contains the consistency status of shared memory at each node and a checkpoint counter. The consistency status indicates how many updates by other nodes have been incorporated to this node's shared data and how many checkpoints this node has performed.

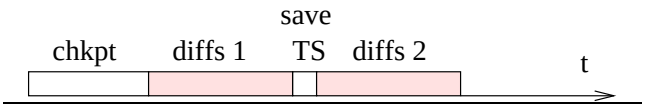

Figure 3. Checkpoint phases: thread state checkpointing (chkpt), diff propagation to secondary page homes (diffs 1), saving of checkpoint timestamp on backup node (save TS), diff propagation to primary page homes (diffs 2).

To checkpoint the state of each thread, we create in each system node a special thread, the *checkpoint thread*. The checkpoint thread is, in general, suspended and is only resumed when a local checkpoint is triggered. The checkpoint thread's primary role is to save the execution contexts and the stacks of the compute threads remotely on the backup node and coordinate the checkpointing actions. The separate checkpoint thread is required because the execution of the compute threads needs to be temporarily *interrupted* while their stacks are being saved.

Any of the regular compute threads can trigger a checkpoint by signaling the checkpoint thread (see Figure 2). At that point, the checkpoint thread resumes and in turn signals all of the compute threads to save their execution context (structure `sigcontext`) and suspend execution. Then, the checkpoint thread performs the following operations:

1. It commits all updates sent by other nodes so that they become part of the checkpointed application state.
2. It saves the stack of each compute thread remotely, at the corresponding stack buffer on the backup node.
3. It saves the context of each compute thread remotely, at the corresponding context buffer on the backup node, along with the stack size, the *checkpoint timestamp*, and a checkpoint-related thread-specific data structure.

Finally, the update propagation phase of the *application* state is further split in two sub-phases corresponding to sending the local diffs first all to secondary homes and then to primary homes (see Figure 3). In between the two phases of update propagation, the thread that initiated the checkpoint (*leader thread*) saves the *checkpoint timestamp* on the backup node. This timestamp is used to identify the status of update propagation at the point of failure and determine the appropriate actions for recovery.

3.2 Ensuring Atomicity of a Checkpoint

Next, we describe how the two checkpoint phases i.e., thread state and application state checkpointing are made atomic. The execution of all compute threads in a node is interrupted while the local updates are committed and the thread stacks are saved remotely. This guarantees that there are no updates intervening between the local commit of updates and saving the thread contexts and stacks by the checkpoint thread.

Application data updates are sent directly to the backup node from the pages within which they occur, without any extra buffering. This is achieved with the remote DMA capabilities of the underlying communication layer and is important for reducing protocol overhead. For this reason and to ensure consistency, we need to disallow further modifications to the respective pages until the diff propagation phase is complete. For the same reason, the primary home of a page cannot write directly to the page it hosts, but only to a working copy of the page. Its changes are committed to the permanent page replica hosted locally, by means of diffs, as for all other nodes.

Finally, we use the saved *checkpoint timestamp* (Figure 3) to determine whether or not a checkpoint has been interrupted by a failure and if yes, which of the two checkpoints on the backup node is the last valid checkpoint to be used for restarting the threads, as discussed next.

3.3 Failure Recovery Path

A node failure is detected by some system node during communication with the failed node. After a failure has been detected, recovery takes place. First, the backup node identifies the execution point at which the failure occurred and the first valid checkpoint taken before the failure. A valid checkpoint is an atomically taken checkpoint, i.e., not interrupted by a failure. If a failure occurs *during* diff propagation in our checkpointing scheme, then, upon recovery, our system uses the second page replica to restore consistency among the two home replicas of the affected pages. Usually a checkpoint is much shorter than the interval between checkpoints. Hence, the probability of a failure occurring during the time a checkpoint is taken is low. In this unlikely case, recovery may involve undoing or reapplying partially propagated diffs, depending on whether the failure occurs during the first or second part of diff propagation respectively.

In contrast, in the general case where a failure does not occur during diff propagation, recovery is fast and only consists of reconfiguration action, reassigning new page and lock homes where necessary, and resuming the failed threads on the backup node. Reconfiguration actions and any necessary restoration of shared memory consistency in the case of failure during checkpointing, are performed by all nodes in a distributed fashion. Subsequently, the compute threads that used to run on the failed node are resumed on the backup node where their execution state and stack had been saved. Execution is resumed at the point where the last valid checkpoint was originally created by the failed node. Next, we discuss in detail the process of failure detection and then the recovery procedure.

3.3.1 Failure Detection and Diagnosis

A failure is eventually detected by some thread in one of the following situations:

- In an internal barrier, where a thread reaches an internal barrier and waits for some local thread which is behind because it has already detected a failure and is participating in the recovery process.
- In a read/write operation, where a thread fails to perform a remote read/write operation on the failed node, or awaits for a specific data item that never gets updated.
- While spinning on a condition variable, where a thread that is waiting for too long checks if any associated remote node has failed, or whether a local recovery process due to the failure of a remote node has already been initiated.

After the failure of a node F is detected, simple recovery actions are executed on all nodes as follows:

1. On each node, all local threads reach an internal barrier.
2. The backup node of F, F', determines the valid checkpoint of F that will drive recovery.
3. All of the nodes synchronize at a global barrier.
4. Each node runs the recovery handler, which reconstructs application and protocol state.
5. The backup node resumes locally the threads that were running on the failed node.
6. All of the nodes synchronize at a global barrier and resume normal execution henceforth. Next we discuss in more detail the steps that require further explanation.

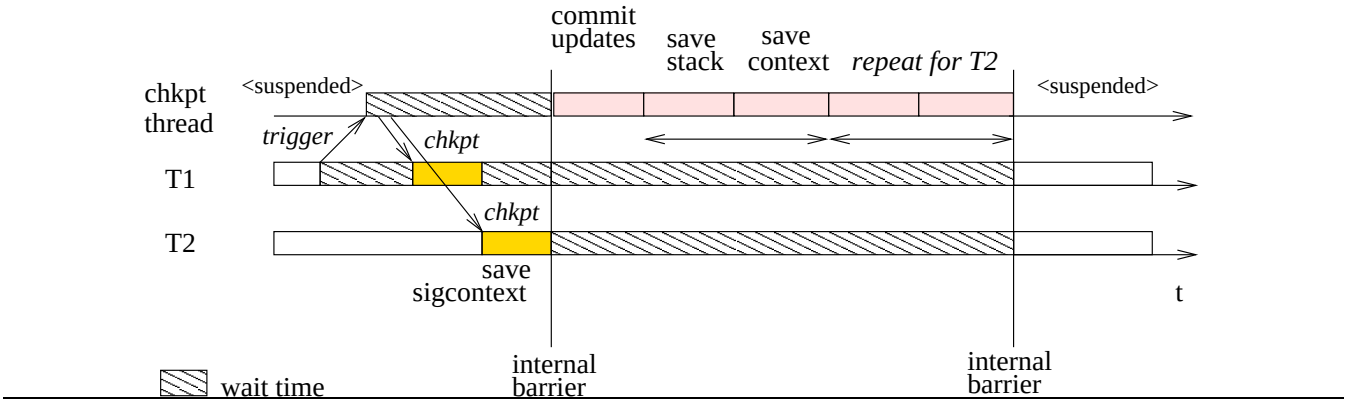

Figure 2. Checkpointing scheme for thread execution state.

After detecting a failure, the backup node for the failed node performs failure diagnosis in Step 2. The backup node identifies the execution point at which the failure occurred. Then, it determines which checkpoint among the two saved thread execution checkpoints must be used for recovery.

We distinguish two different types of failures. A failure of node F may occur before or after F has fully propagated at least to one set of home nodes the local updates of the interval in which the latest checkpoint was taken.

- In the former case (before propagation), recovery is based on the *older* checkpoint and the failed threads resume execution from the beginning of the interval in which the latest checkpoint was taken.
- In the latter case (after propagation), recovery is based on the *latest* checkpoint: the failed threads resume execution from the beginning of the interval that immediately follows the latest checkpoint.

The problem of identifying which consistency point to use for recovery is equivalent to the problem of deciding whether the checkpoint was completed atomically or was interrupted due to a failure. Our system uses the *checkpoint timestamp* saved by the failed node on its backup in between the two phases of update propagation to resolve this problem. More specifically, the backup node reads this timestamp and compares it with the timestamp of the latest checkpoint. If the two timestamps match, then the latest checkpoint is used for recovery; otherwise, the failure occurred while the update propagation was in progress and the old checkpoint is used instead.

3.3.2 Failure Recovery Procedure

After a failure has been diagnosed, recovery takes place in Step 4. Recovery mainly consists of two actions: a) shared memory reconfiguration and, if necessary, restoration of consistency and b) protocol state, namely lock, reconfiguration. Both actions are performed by all nodes in a distributed fashion.

The first action requires that shared pages affected by the failure are reconfigured: for each page whose primary or secondary home used to be the failed node F, a new primary or secondary home is assigned according to a primary-backup mapping scheme. Next, consistency is restored among the two home copies of updated shared pages through the following operations: Every node reads the checkpoint timestamp T which was saved by the failed node F on its backup node F' in between the two phases of update propagation. This timestamp indicates the latest updates that were successfully propagated by F to at least the first set of their homes

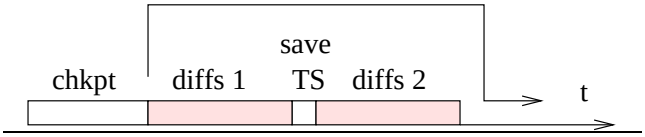

Figure 4. Threads resume execution from last checkpoint.

and are considered permanent. Any other updates with timestamp greater than T, must have been sent by F on a later unsuccessful attempt and must be undone.

Each node examines the updates it received from F by examining the timestamps of shared pages. For pages with timestamp greater than T, there is a possible inconsistency between the secondary home copy of the page (corrupted copy) and the primary home copy (valid copy). Consistency is restored by having the primary home of every such page overwrite the secondary home copy of the page with its own valid home copy. For pages with timestamp less or equal to T, there is a possible inconsistency between the secondary home copy of the page (updated copy) and the primary home copy (partially updated or stale copy). Consistency is restored by having the secondary home of every such page overwrite the primary home copy of the page with its own updated home copy.

In this section, we discuss the reconfiguration of the protocol state. The most important part of protocol state is locks. For each lock whose primary or secondary home used to be the failed node F, a new primary or secondary home is assigned according to a predefined primary-backup mapping scheme. If a node becomes the new primary or secondary home for a lock, it copies the associated lock data structures from the secondary or primary home of the lock, respectively.

Finally, in Steps 5 and 6 the compute threads that used to run on the failed node resume execution on the backup node. Figure 4 depicts how the failed threads resume execution on the backup node. The leader thread that had initiated the checkpoint resumes immediately at the point of the last successful checkpoint, skips the propagation phase, which had been successfully completed before the failure, and continues execution. The rest of the threads resume execution at the point where each thread was originally checkpointed, which for all threads except the leader thread is not apriori determined.

4. Experimental Setup

In this section we describe the platform and benchmarks we use in our experiments.

VMMC Operation	Overhead
1-word send (one-way lat)	8μs
1-word fetch (round-trip lat)	22μs
4 KByte send (one-way lat)	52μs
4 KByte fetch (round-trip lat)	81μs
Maximum ping-pong bandwidth	118 MBy/s
Maximum fetch bandwidth	118 MBy/s
Notifi cation	18μs
Remote lock acquire	53.8μs
Local lock acquire	12.7μs
Remote lock release	7.4μs

Table 1. Basic costs for the VMMC communication layer.

Application	Problem Size
FFT	1M points
LU-contiguous	1024×1024 matrix
WaterNsquared	4096 molecules
WaterSpatialFL	4096 molecules
WaterSpatial	4096 molecules
RadixLocal	4M keys

Table 2. Applications and problem sizes used for our performance evaluation.

4.1 Platform

We perform all experiments on an 8-node (16-processor) cluster equipped with 400MHz, 2-way Pentium-II SMP nodes running Linux and interconnected with the Myrinet SAN [6]. Cross-node communication is based on a user–level communication library, Virtual Memory Mapped Communication (VMMC) [11].

VMMC provides direct data transfer between the sender's and receiver's virtual address spaces. More specifically, VMMC provides remote deposit and remote fetch operations. These operations allow for data transferred between two nodes to be fetched from and deposited to specified virtual addresses in the host's main memory without interrupting the remote host processor or copying between communication buffers. These features dramatically reduce latency compared to traditional TCP/IP based implementations. VMMC also tolerates transient network errors by using packet retransmission, and guarantees FIFO message delivery. Table 1 shows the cost of basic communication operations on our cluster. Noticeably, the communication layer provides a one way, end–to–end latency of around 8μs.

4.2 Compute Server Benchmarks

For our performance evaluation of a compute server running scientific applications, we use the SPLASH-2 [30, 14] application suite. The specific applications and problem sizes that we use are (Table 2): FFT (1M points), LU-contiguous (1024×1024 matrix), WaterNsquared (4096 molecules), WaterSpatialFL (4096 molecules), WaterSpatial (4096 molecules), and RadixLocal (4M keys).

4.3 In-memory Transactional Server Benchmarks

Vista is a lightweight in-memory transaction processing server [20]. To the best of our knowledge, Vista is the fastest open-source transaction system available. As such, it provides a good "stress test" for a cluster-based server. It has also been used by previous studies on fault tolerance and availability [1, 20].

Vista achieves high performance by avoiding disk I/O. Instead, it relies on the presence of a reliable memory system [7] layer to achieve persistence. Besides avoiding disk I/O, the assumption of the reliable memory underneath allows considerable simplification

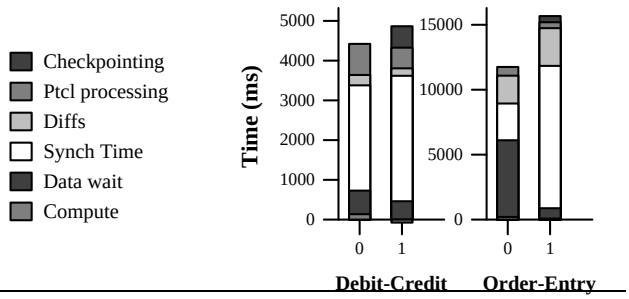

Figure 6. Execution time breakdown for Vista on 4 system nodes, for both the unreliable (left bar) and the reliable (right bar) protocols.

in the implementation of transaction semantics [20]. The original Vista system, relies on the Rio file-system [7] to protect main memory against its two common causes of failure, power failures and operating system crashes.

In contrast, our cluster server does not rely on the presence of the specialized Rio file system. Instead, we implement the recoverable and highly available memory layer using our techniques for data replication, remote checkpointing, and resuming of threads. Because Vista does not store its data on disk, but rather keeps its data structures and database tables in memory, in the original system, the data remains safe when the machine fails, but it is unavailable until the machine recovers. *FineFRC* uses data replication to provide both reliability and data availability.

In our experiments, we use the Debit-Credit and Order-Entry benchmarks provided with Vista [20]. These benchmarks are variants of the widely used TPC-B and TPC-C benchmarks. Debit-Credit (TPC-B) models banking transactions [33]. The database consists of a number of branches, tellers, and accounts. Each transaction updates the balance in a random account and the balances in the corresponding branch and teller. Each transaction also appends a history record to an audit trail. The Debit-Credit benchmark differs from TPC-B primarily in that it stores the audit trail in a 2 Mbytes circular buffer in order to keep it in memory.

Order-Entry (TPC-C) models the activities of a wholesale supplier who receives orders, payments, and deliveries [34]. The database consists of a number of warehouses, districts, customers, orders, and items. In both Debit-Credit and Order-Entry we issue transactions sequentially and as fast as possible. They do not perform any terminal I/O in order to isolate the performance of the underlying transaction system. Both benchmarks exhibit very fine grained transactions (on the order of tens of μs on our system) and write little data (on the order of tens to hundreds of bytes per transaction).

5. Experimental Results

We first examine overall execution results for the reliable versus the unreliable system, respectively, highlighting overheads during failure-free execution. Then, we examine recovery times.

5.1 Failure-free Overheads

Figures 5 and 6 present a comparison of the execution time for the unreliable and reliable protocols for the Splash and Vista benchmarks, with one and two compute threads per node.

We present a breakdown of the execution time in six basic components: compute time, data wait time, synchronization time, the time to create and transfer diffs to home nodes, protocol processing and checkpointing time. This breakdown shows the relative impact of different protocol aspects on system performance for the reliable protocol versus the unreliable one. Synchronization time includes

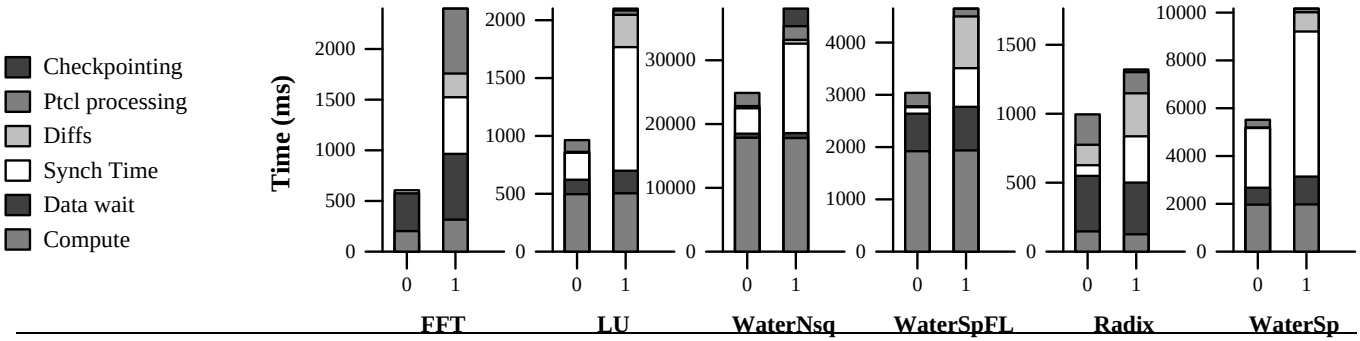

Figure 5. Execution time breakdown for SPLASH-2 on 8 system nodes, for both the unreliable (left bar) and the reliable (right bar) protocols.

both intra- and inter-node synchronization. Diff time includes the time necessary for computing and sending diffs during all checkpoints. The checkpoint overhead includes the checkpoint trigger overhead due to signaling between the compute threads and the checkpoint thread, the per-thread cost of saving the thread context locally and the time necessary for the checkpoint thread to save the contexts and stacks remotely. Protocol time consists of the remaining execution time.

Comparing the overall execution time of the two protocols across all graphs, we see that the main source of overhead compared to the base system is an increase in the synchronization and diff portion time, while the overheads due to thread state checkpointing are negligible. For the diff portion, the increase comes from the need to propagate diffs to two sets of homes and due to the additional diff creation by the home nodes themselves (Section 3.2). For the synchronization portion, the increase comes from the need to coordinate thread actions during checkpoints, such as disallowing concurrent writes to the specific pages involved in the checkpoint, and other synchronization needed for maintaining atomicity of protocol state.

The overhead in FFT is relatively high, compared with all other applications and configurations. The busy-waiting nature of barriers in GeNIMA and the large number of diff flushes during barriers in FFT create contention between protocol and application actions in the two-thread configuration (Section 5.2): diffs for roughly a thousand application pages need to be sent because the barrier constitutes a consistency point, hence a checkpoint.

Overall, overheads of reliability actions taken during failure-free execution are 38% on average for all applications considered. Across both application suites, checkpoints occur with high frequency ranging from 2 checkpoints per second to 124 checkpoints per second in the Splash benchmarks and above 800 checkpoints per second in the Vista benchmarks, obviating the need for protocol-induced periodic checkpoints. In the Splash benchmarks, checkpoints are driven by consistency actions due to the high level of data sharing inherent in the applications. In the Vista benchmarks, while actual sharing is low, transactions are very fine-grain, on the order of a few to tens of μs, and each transaction's commit automatically triggers a checkpoint.

5.2 Recovery Time

The recovery overhead depends on a number of parameters that can be classified as static and dynamic. The static parameters are fixed for a particular run and are: the application pattern, the problem size, and the number of nodes. The first two determine the number of locks and shared pages used. The third affects synchronization time as well as the distribution of shared data (pages and locks) to homes.

There are two main dynamic parameters which depend on the exact point at which a failure occurs: the number of updates that need to be undone during the recovery and the home distribution of the corresponding updated pages.

The worst-case scenario (in a single run) for the number of updates that need to be undone is when the failure occurs in the interval with the greatest number of updates and immediately after all of the diffs have been sent to the first set of homes, but exactly before the write of the checkpoint timestamp that indicates the end of the first phase of diff propagation. This means that, during recovery, the largest possible number of updates need to be undone.

Moreover, the home distribution of the updated pages affects the overall time to undo updates at each individual node, since the restoration of a corrupted copy might be done using a local copy operation or a remote write depending on the primary home placement. Thus, we are interested in the measurement of the recovery time in two different failure cases: one that happens during diff propagation, whose extreme scenario we mentioned above, and another that happens at any other point in the execution, when recovery consists only of reconfiguration actions and the resumption of failed threads on the backup node. This time should be invariable across different failure cases.

To examine the cost of recovery we inject failures at representative points during system operation and measure the cost of each recovery action. Tables 3 and 4 present the recovery cost in several representative cases. For the Splash benchmarks, the faults are injected as follows: in WaterNsquared, during barrier synchronization, in FFT during diff propagation and, in LU during diff propagation. FFT and LU are also the applications that require the largest amounts of update data movement during checkpoints, so we can consider the respective recovery times as worst case scenarios. For the Vista benchmarks, the failures are injected at a random point during execution outside of synchronization operations and checkpoints. The runs are performed on a 4-node (8-processor) system.

The overhead is broken into the following components: inter-node synchronization, recovery actions performed exclusively by the backup node, reconfiguration of homes for system locks, initialization/restoration of lock-related data structures, reconfiguration of page homes and initialization/restoration of orphan/corrupted home pages, and resumption of failed threads on the backup node. The final row shows the number of pages that need to be restored in the given recovery session.

Overall, recovery overhead is fairly low, and in all cases including the worst case scenarios in FFT and LU, we measure it to be below 600 msec.

6. Related Work

There is a large body of previous work in fault tolerance in a number of research areas. A survey of rollback-recovery protocols for message passing systems is presented in [12]. More relevant for our work is the survey of recoverable distributed shared virtual mem-

Application: Recovery Action	WaterNsquared Cost (ms)	FFT Cost (ms)	LU Cost (ms)
Internode synch	20.307	0.013	0.091
Backup processing	0.043	0.037	0.102
Reconfigure locks	0.114	0.004	0.085
Reconfigure pages	0.104	162.813	53.366
Resume threads	0.393	399.778	399.692
Total	20.961	562.645	453.336
Restored pages	0	1024	98

Table 3. Indicative examples of recovery cost for the Splash benchmarks.

Application: Recovery Action	Debit-Credit Cost (ms)	Order-Entry Cost (ms)
Internode synch	3.121	90.962
Backup processing	0.074	0.050
Reconfigure locks	193.971	195.000
Reconfigure pages	0.158	78.690
Resume threads	0.080	0.108
Total	197.404	364.81
Restored pages	0	0

Table 4. Indicative examples of recovery cost for the Vista benchmarks.

ory systems presented in [21]. Previous work that has examined various aspects of recovery in software shared memory systems includes [27, 10, 31, 17, 1, 18, 26]. In all these cases, the focus has been on protocol extensions for logging and checkpointing that enable coarse-grain system recovery. Also, most studies do not examine the recovery path and do not provide results for recovery overheads. In our work, we demonstrate how systems can support continuous operation with below-second recovery times through dynamic replication of system state.

Our work is based on the system presented in [8]. Our previous work has focused on the failure free execution. In this work we examine in detail the recovery actions, we build a working prototype that recovers from failures, and provide measurements of the recovery and reconfiguration time.

A good taxonomy of various systems is presented in [24]. Based on this taxonomy the scheme we use is a *backward error recovery* scheme that uses replication to distinct volatile memories for storage protection, performs uncoordinated checkpoints across nodes and coordinated inside each node, and achieves separation of checkpoint and working data with full duplication but by incremental propagation of modifications.

Our approach shares similarities with the Tandem NonStop architecture and the Guardian operating system [3]. Both approaches use independent components operating asynchronously, where any failed component can be replaced by another component. Furthermore, both systems can tolerate a single failure. However, the two approaches target different platforms. The NonStop architecture involves special hardware components including ECC memory. In contrast, our system is implemented with commodity components and uses runtime support to maintain two consistent copies of system state during execution.

Recently, there has been more work on techniques that allow hardware shared memory servers to transparently recover from faults [25, 24]. The emphasis in these cases is placed on the failure-free case and available results are based mostly on simulation. The proposed approaches are log-based and require modifications to the host processors.

Previous work in fast failover [31, 1] has addressed issues related to maintaining low-overhead backups of in-memory state. However, in these cases, the primary and backup nodes are not able to support a single copy of the same multithreaded application. Our work uses the same fail-over concept, however, allows the primary and backup nodes to transparently share application state, significantly broadening the range of server applications that can benefit.

Recent work in virtualization layers [2, 28] aims at providing higher level abstractions on top of hardware platforms. Although this work has mostly targeted single-node systems, it is currently being extended with migration capabilities towards multi-node systems for tolerating system faults. The issues that arise in fast failover techniques for this purpose are similar to many of the issues discussed in our work. Zap [23] is a system that allows sub-second migration of a group of processes from a Linux system to another. The main technique is to provide a virtualization layer that completely decouples process state from the underlying system. Although Zap and *FineFRC* share the same goal, the approaches differ. In particular, Zap does not specifically support parallel applications or maintain data copies on a cluster.

Finally, the currently renewed interest in improving server reliability is manifested by the emergence of a large number of commercial cluster management systems, such as Microsoft MSCS [29], NCR Lifekeeper [22], and Veritas Firstwatch [9] that provide fault-tolerance functionalities. Such products differ from our approach in that they employ fail-over to backup servers to provide continuous service operation non-transparently, at the application level.

7. Conclusions

In this work we investigate building commodity, transparently reliable servers that support continuous availability. Our approach uses dynamic state replication during execution to guarantee system recovery when a failure occurs. We focus on the recovery path and present the protocol actions required for transparent and efficient recovery. We present results for a working prototype that reconfigures itself after failures injected during system operation.

Our approach results in sub-second recovery times for all cases we measure. Using dynamic replication of system and application state during execution results in an average 38% increase of program execution time, which we believe is acceptable for many application areas, given the fast recovery provided by our system. Furthermore, we expect that the applications presented are examples of "stress-tests" for our system and this overhead can be significantly reduced for applications with low to moderate sharing needs which allow the system better control over the checkpoint frequency.

Our experience shows that there is a need for a better understanding of the tradeoffs between designing systems for performance and designing for reliability. Communication protocols in system area networks have been highly optimized for data transfers. In particular, messages are not buffered at the host in order to eliminate software data copies out of the critical path. Instead, the memory that the application is writing on is itself used as the send buffer. This makes atomicity for the remote replication of updates in the presence of failures complex to implement correctly in a system using remote data copies for reliability purposes.

Overall, however complex the system design may be, our work demonstrates that using commodity, transparently reliable servers for continuous operation is a promising approach for building low-cost application servers.

Acknowledgments

We thank the anonymous reviewers for their comments and Reza Azimi for his help with VMMC. We thankfully acknowledge the support of Natural Sciences and Engineering Research Council of

Canada, IBM, Canada Foundation for Innovation, Ontario Centers of Excellence, the European Commission FP6 HiPEAC Network of Excellence, and Research and Technology, Greece.

References

[1] C. Amza, A. L. Cox, and W. Zwaenepoel. Data replication strategies for fault tolerance and availability on commodity clusters. In *Proc. of the Int'l Conference on Dependable Systems and Networks*, 2000.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *19th ACM Symposium on Operating Systems Principles*, Oct 2003.

[3] J. Bartlett, W. Bartlett, R. Carr, D. Garcia, J. G. R. Horst, R. Jardine, D. Lenoski, and D. McGuire. Fault tolerance in Tandem computer systems. Technical Report TR-90.5, Tandem, 1990.

[4] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[5] A. Bilas, C. Liao, and J. P. Singh. Accelerating shared virtual memory using commodity ni support to avoid asynchronous message handling. In *The 26th Int'l Symposium on Computer Architecture*, May 1999.

[6] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, Feb. 1995.

[7] P. Chen, W. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio file cache: Surviving operating system crashes. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 74–83, Oct. 1996.

[8] R. Christodoulopoulou, R. Azimi, and A. Bilas. Dynamic data replication: An approach to providing fault-tolerant shared memory clusters. *Proc. of The 9th IEEE Symposium on High-Performance Computer Architecture (HPCA9)*, Submitted for Publication, 2003.

[9] V. S. Corp. Veritas firstwatch. http://www.veritas.com.

[10] M. Costa, P. Guedes, M. Sequeira, N. Neves, and M. Castro. Lightweight logging for lazy release consistent distributed shared memory. In *Proc. of the Operating Systems Design and Implementation Conference*, pages 59–73, Oct. 1996.

[11] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. Vmmc-2: Efficient support for reliable, connection-oriented communication. In *Proc. of the Hot Interconnects Symposium V*, Aug. 1997.

[12] E. N. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

[13] IBM. High availability with DB2 UDB and Steeleye Lifekeeper. IBM Center for Advanced Studies Conference (CASCON): Technology Showcase, Toronto, Canada, Oct 2003.

[14] D. Jiang, H. Shan, and J. P. Singh. Application restructuring and performance portability across shared virtual memory and hardware-coherent multiprocessors. In *Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming*, June 1997.

[15] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pages 13–21, 1992.

[16] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, Jan. 1994.

[17] A.-M. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, and I. Puaut. A recoverable distributed shared memory integrating coherence and recoverability. In *Proc. of the 25th Annual Int'l Symp. on Fault-Tolerant Computing (FTCS-25)*, pages 289–298, 1995.

[18] J. Kim and N. Vaidya. Analysis of failure recovery schemes for distributed shared-memory systems. *IEE Computers and Digital Techniques*, 146(3), May 1999.

[19] K. Li. Ivy: A shared virtual memory system for parallel computing. *Proceedings of the 1988 International Conference on Parallel Processing*, 2:94–101, August 1988.

[20] D. Lowell and P. Chen. Free transactions with Rio Vista. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Oct. 1997.

[21] C. Morin and I. Puaut. A survey of recoverable distributed shared virtual memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(9):959–969, 1997.

[22] NCR Lifekeeper. http://www.ncr.com.

[23] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *OSDI'02*, Dec. 2002.

[24] M. Prvulovic, Z. Zhang, and J. Torrellas. Revive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proceedings of the 29th Int'l Symposium on Computer Architecture*, May 2002.

[25] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. Safetynet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proceedings of the 29th Int'l Symposium on Computer Architecture*, May 2002.

[26] M. Stumm and S. Zhou. Fault tolerant distributed shared memory algorithms. In *Proc. of the 2nd IEEE Symposium on Parallel and Distributed Processing*, pages 719–724, December 1990.

[27] F. Sultan, T. D. Nguyen, and L. Iftode. Scalable fault-tolerant distributed shared memory. In *Proc. of Supercomputing*, 2000.

[28] VMware. Vmware ESX Server Storage Area Networks. http://www.vmware.com/, 2003.

[29] W. Vogels, D. Dumitriu, K. P. Birman, R. Gamache, M. Massa, R. Short, J. Vert, J.Barrera, and J. Gray. The design and architecture of the Microsoft cluster service. In *Proceedings of the 1998 Fault Tolerant Computing Symposium*, pages 422–431, 1998.

[30] S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. Methodological considerations and characterization of the SPLASH-2 parallel application suite. In *Proceedings of the 23rd Int'l Symposium on Computer Architecture*, May 1995.

[31] Y. Zhou, P. Chen, and K. Li. Fast cluster failover using virtual memory-mapped communication. In *Proc. of the Int'l Conference on Supercomputing*, June 1999.

[32] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation*, pages 75–88, 1996.

[33] Transaction Processing Performance Council. TPC Benchmark B Standard Specification, August 1990.

[34] Transaction Processing Performance Council. TPC Benchmark C Standard Specification, August 1996.