Fast Cluster Failover Using Virtual Memory-Mapped Communication

Yuanyuan Zhou*, Peter M. Chen+, and Kai Li*

* Computer Science Department
Princeton University
Princeton, NJ 08544
{yzhou, li}@cs.princeton.edu

+Computer Science and Engineering Division
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor, MI 48109-2122
pmchen@eecs.umich.edu

Abstract

This paper proposes a novel way to use virtual memory-mapped communication (VMMC) to reduce the failover time on clusters. With the VMMC model, applications' virtual address space can be efficiently mirrored on remote memory either automatically or via explicit messages. When a machine fails, its applications can restart from the most recent checkpoints on the failover node with minimal memory copying and disk I/O overhead. This method requires little change to applications' source code. We developed two fast failover protocols: *deliberate update failover protocol* (DU) and *automatic update failover protocol* (AU). The first can run on any system that supports VMMC, whereas the other requires special network interface support.

We implemented these two protocols on two different clusters that supported VMMC communication. Our results with three transaction-based applications show that both protocols work quite well. The deliberate update protocol imposes 4-21% overhead when taking checkpoints every 2 seconds. If an application can tolerate 20% overhead, this protocol can failover to another machine within 4 milliseconds in the best case and from 0.1 to 3 seconds in the worst case. The failover performance can be further improved by using special network interface hardware. The automatic update protocol is able to take checkpoints every 0.1 seconds with only 3-12% overhead. If 10% overhead is allowed, it can failover applications from 0.01 to 0.4 seconds in the worst case.

1 Introduction

Reliability and availability are of critical importance for applications where computer malfunctions have catastrophic results. Examples include aircraft flight-control systems, hospital patient monitors, and financial on-line transactional applications. These applications require computer systems to continue functioning in the presence of hardware and software failures. In the past, custom-designed, fault-tolerant hardware has been used to provide highly reliable and available systems. Tandem [2, 1, 32, 18] and Stratus [33] are examples of such systems. However, custom systems tend to be expensive and do not track technology trends as quickly as commodity hardware.

Recent efforts have focused on building fault-tolerant systems using clusters of commodity components. The Microsoft cluster service (MSCS) [37] and the Digital TruCluster [19] are two such examples. If a node crashes in a cluster, applications running on the node migrate transparently to another node in the same cluster. This is called the "failover" process. Failover allows the system to continue providing services, although the overall performance may degrade.

However, most existing fault-tolerant clusters take more than 10 seconds to failover applications [8, 37]. This is unacceptable for mission-critical applications such as aircraft flight-control systems. There are two main factors that contribute to this high failover delay. The first is that the failover process in these systems usually involves access to external shared storage, such as shared disks. At periodic checkpoint intervals, an application must write important state information and other data to a shared disk. When a node fails, its checkpointed data is reloaded from the shared disk into memory before continuing the application on another node. As a result, the memory state recovery performance is limited by slow disk accesses. The second factor is that existing cluster failover schemes limit the checkpoint frequency for applications that cannot afford the high checkpointing overhead associated with writing to disks. Existing cluster failover schemes have high worst-case recomputation costs, which is determined by the checkpoint interval time.

Another line of related research is in the area of non-volatile and persistent memory. For example, the eNVy system [38] uses non-volatile Flash RAM to serve as a permanent data repository. It can tolerate hardware faults such as power outages. The Rio file cache [15] makes ordinary main memory safe for persistent storage by enabling memory to survive machine crashes. The main advantage of using non-volatile or persistent memory is that it can reduce the overhead of accessing persistent storage by several orders of magnitude. However, these approaches have not addressed how to failover quickly. Although these systems preserve data if a node fails, applications running on the node will be unavailable until the node restarts or its memory is moved to another node. When using the traditional failover method, the nodes have to periodically write their state to shared storage. This eliminates the benefits of using non-volatile or persistent memory. The challenging question is what architectural support and algorithms are needed to achieve fast failover for clusters.

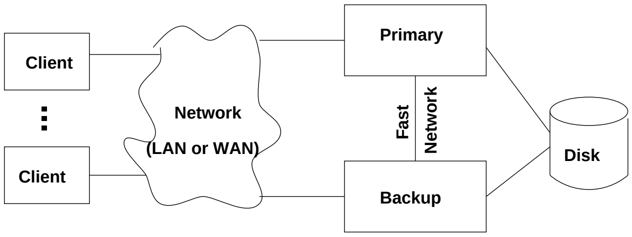

Figure 1: An example of highly reliable and available clusters

This paper proposes a novel way to use virtual memory-mapped communication (VMMC) to reduce the failover time on clusters. Our fast failover method includes two parts: fast checkpointing and fast failover. When a node fails, its applications can restart from the most recent checkpoints on the failover node where most of the stack, heap, and other memory data are already in place. In the best scenario, the node that takes over can immediately continue providing services with no memory copying or disk access overhead. Another feature of our method is application transparency. It requires little change to applications' source code. We developed two fast failover protocols: one requiring special network interface support and the other requiring no special network interface support.

To evaluate these protocols and investigate the design tradeoffs of the network interface hardware, we implemented two fault-tolerant PC clusters, one using SHRIMP network interfaces [14, 13] and the other using Myrinet [3, 9, 7] network interfaces. Both systems support virtual memory mapped communication. Our results with three transaction-based applications show that VMMC is a convenient and efficient communication mechanism to support fast failover on clusters. With no special network interface support, our system imposes 4-21% overhead when taking checkpoints every 2 seconds. If applications can tolerate 20% overhead, the deliberate update failover protocol can failover to another node within 4 milliseconds in the best case and from 0.1 to 3 seconds in the worst case. The failover performance can be further improved by using special network interface hardware. The automatic update failover protocol is able to take checkpoints every 0.1 seconds with only 3-12% overhead. If 10% overhead is allowed, it can failover applications from 0.01 to 0.4 seconds in the worst case.

2 Background

Building a reliable and available system usually involves redundant hardware components so that working components can take over when failures occur. Figure 1 shows the architecture for a simple cluster that provides reliability and availability. The primary node runs an application, while the backup node either stands by or runs other applications. The primary communicates with the backup through a network. The backup node can still access disk data when the primary fails because dual-ported disks are connected to both nodes.

A typical way to implement a highly reliable and available system is to checkpoint the application's data periodically to shared disks [1]. When the primary fails, the backup node will reload the checkpoint data from shared disks and continue the application from the most recent checkpoint. Taking checkpoints more frequently generally increases overhead during normal operation, but reduces recovery time by minimizing the amount of lost work that must be redone. The automatic process for detecting a failure and switching to the backup node at the most recent checkpoint is called *failover*.

Figure 2 illustrates the failover process. A common way to detect failures is to use a periodic heartbeat mechanism [24]. Application C first runs on the primary, which periodically sends an "I'm alive" message to the backup. C checkpoints every T seconds. After C runs for a while, the primary dies at time x. When the backup discovers the absence of the heartbeat message from the primary, it first confirms the primary's failure using some voting mechanism. Once the failure is detected, the backup takes over. It first loads C's data from shared disks and then restarts C at its most recent checkpoint. Finally, the backup has to redo all the computation from the most recent checkpoint to the point where the primary failed. After these steps, the system can continue providing service to its clients.

Three factors are relevant when evaluating a fault tolerant cluster: *system overhead, failover delay,* and *application transparency.* System overhead is measured by two metrics:

- *Relative overhead*: the amount of overhead relative to the total execution time. This includes the overhead for taking checkpoints and any other operations that are necessary for correct failover. The overhead usually depends on how frequently checkpoints are taken. In general, the more frequently an application checkpoints, the more execution overhead is added. The traditional method of checkpointing to disks is inefficient. For example, even if less than 4 KBytes of data are modified during a checkpoint interval, it takes more than 10 milliseconds to write the data to a fast storage system [16]. Therefore, applications have to reduce the frequency of checkpoints in order to lower the percentage overhead below some acceptable threshold (e.g. 10%).
- *Maximum checkpoint time.* Some applications consider this an important metric because checkpoints may block, that is, no service is provided during a checkpoint. For example, it is unacceptable to block for more than two seconds for many on-line transaction processing systems because they require that every transaction finishes within two seconds.

The failover delay is an important metric of a fault-tolerant system, especially for mission-critical applications. This is because the delay defines the amount of time the service is unavailable. The failover delay includes the time to detect failure, recover memory state to the most recent checkpoint, and redo lost computation until the failure point. Most existing cluster systems take a long time to failover (more than 10 seconds), mainly for two reasons. First, applications must reload data from disk to recover the state at the most recent checkpoint. Second, most checkpoint intervals are large, which leads to a long redo process during recovery. For example, if the application checkpoints every 10 seconds, the backup has to redo 10 seconds of work (in the worst case) after it takes over from the primary.

Application transparency reflects how much applications need to be involved in recovery. The ideal case is full transparency, where any program written for an unreliable system can failover successfully. For example, the Stratus system [33] appears to users as a conventional single-node computer that does not require special application code for failure detection and recovery. However, many other existing

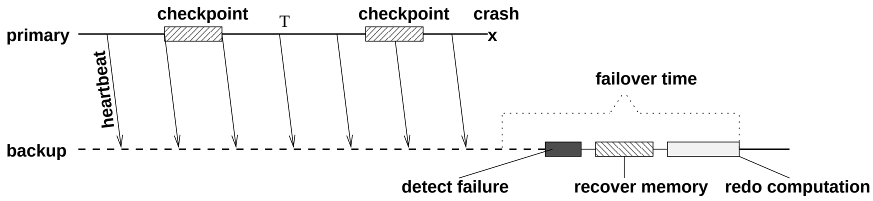

Figure 2: Failover process

fault-tolerant systems require applications to help checkpoint their state and/or assist during failures. In some systems [37, 19], applications must provide recovery scripts that are executed by the system during failover.

In this paper, we propose a new technique that not only reduces the system overhead and failover delay, but also requires very little change to applications' source code [29].

3 Fast failover

The main idea of our fast failover method is to mirror applications' virtual address space on the backup so that when the primary fails, the backup can take over quickly since it has most of the memory data in place. As in many other similar fault-tolerant systems, we also assume that the primary and backup fail independently.

In this section, we present our approach by first describing the virtual memory-mapped communication model and then presenting the two failover protocols. Lastly, we describe how to support transaction-based applications and failure detection.

3.1 Virtual Memory-Mapped Communication

A fast communication mechanism is needed to transfer data efficiently between the two application virtual address spaces (primary and backup) in order to replicate application data while minimizing interference with the computation. Among existing communication models, virtual memory-mapped communication (VMMC) [14, 9, 7] is a good candidate because it provides direct data transfer between virtual address spaces. With the VMMC model, the receiver exports variable-sized regions of contiguous virtual memory, called *receive buffers*, with a set of permissions. Any other process, with proper permission, can import the receive buffer to a *proxy receive buffer*, which is a local representation of the remote receive buffer.

Like most commodity network interfaces, VMMC supports *deliberate update* communication. This requires the sender to explicitly initiate a data transfer by specifying a local virtual address, a remote virtual address (proxy), and a transfer size. The data is delivered directly from the sender's to the receiver's virtual address space without memory copying.

Using a custom network interface (SHRIMP [14, 13]) VMMC also supports *automatic update* data transfers. This type of data transfer propagates updates automatically to the corresponding imported receive buffer. To use automatic update, a portion of local virtual memory is bound to an imported receive buffer such that all writes to the bound memory are automatically transferred to the remote receive buffer as a side-effect of local memory writes.

3.2 Automatic update failover protocol

The automatic update mechanism can transfer updates while the application is executing without requiring an explicit send call. The *automatic update failover protocol* (AU) is designed for a network interface with this automatic data propagation capability.

The AU protocol creates two processes when the application starts, one running on the primary and one running on the backup. The primary process does all the computation by executing the application code, while the backup process only handles requests from the primary process.

The virtual address space of the backup process is "mapped" to the primary process using the VMMC mechanism. The mapping is established by having the backup process export its data segment (as receive buffers) to the primary process. The primary process imports and binds these receive buffers to the corresponding addresses in its data segment. By doing this, updates to the primary process's data segment will be transferred to the backup process's data segment while the application is executing. That is, a local write to the data segment of the primary process will be automatically propagated to the corresponding address of the backup process.

The checkpoint procedure is quite simple in this protocol. Because all data segment modifications since the last checkpoint have been automatically propagated to the backup process, the primary only needs to send the current execution environment at the checkpoint. The execution environment includes important register values, instruction counters, stack pointers, etc. The checkpoint procedure completes after the primary receives an acknowledgment from the backup. Waiting for an acknowledgment is necessary for the primary to ensure that all data is in place at the backup.

As in most checkpointing systems, an undo log must be generated on the backup before its data is overwritten with new values propagated from the primary. The undo logs are used to roll back to the most recent checkpoint after application failover. We use copy-on-write to create the undo log on the backup. All user pages are initially write protected. The first write attempt to a page triggers a page fault on the primary. The page fault handler sends an undo log request to the backup. The backup then makes a duplicate of the faulted page and acknowledges the primary. Once the primary receives the backup's reply, it unprotects the page and restarts the faulting instruction. At checkpoints, all dirty pages are write protected to catch write attempts in the next checkpoint interval.

The backup takes over once it detects that the primary has failed. It first rolls back all the changes since the most recent checkpoint by copying undo logs back to the corresponding pages. After the backup recovers the memory state to the most recent checkpoint it redos any lost computation and then starts executing the application.

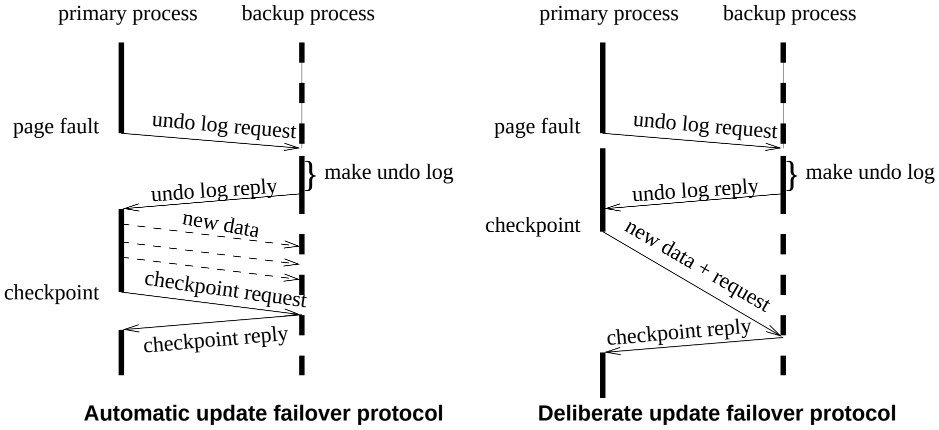

Figure 3: The failover protocols

There are two exceptions of data that is not mapped between the two processes. First, a special region of the backup's address space is not mapped because it contains all the data used in our failover library and the underlying VMMC libraries. Second, the stack segment is not mapped because our operating system (Linux) does not provide a separate stack to handle signals. Copy-on-write is difficult to implement since the copy-on-write handler itself needs to write to the stack. Some operating systems provide a separate stack to handle signals. Instead, we use a shadow-page mechanism for the stack. The shadow buffer mechanism avoids making a full copy of the whole stack by having two receive buffers on the backup for the stack. The two receive buffers are used for receiving the entire primary stack in an alternating fashion.

Besides the application's user data, system states also need to be recovered in order to successfully failover an application. Examples of system states include open files/sockets, file positions, and page protections. Many techniques for recovering system states have been developed in other fault-tolerant systems. For example, one can intercept all system calls and provide a virtual interface to the application [17]. In order to redo the computation on the backup when the primary fails, the failover system must be able to reproduce signals, messages, and any other external events. One simple solution to this problem is to checkpoint before every non-abortable event [17]. An alternative solution is to log those events and replay them after the backup takes over [36, 8, 23, 34]. Our prototype implementation uses the second approach and is able to recover all operating system states which are necessary to failover applications used in our experiments.

3.3 Deliberate update failover protocol

The *deliberate update failover protocol* (DU) is similar to the automatic update protocol. The fundamental difference between the two protocols is that the DU protocol transfers the primary data to the backup using explicit messages. The DU protocol incurs higher data-transfer overhead, but it can be implemented on most existing cluster systems.

At a new checkpoint, the primary process sends the backup all pages that were modified since the last checkpoint. Unlike the AU protocol, the DU protocol does not allow computation during checkpoint-data transfer. Therefore, the checkpoint procedure in the DU protocol may impose more overhead than the AU protocol. On the other hand, the DU protocol can hide the undo log overhead by creating undo logs on the backup in parallel while the primary is computing. The page-fault handler proceeds as soon as it sends the undo log request to the backup. If the communication mechanism guarantees reliable data transfer, there is no need for the primary to wait in the page fault handler since the data on the backup is not updated until the next checkpoint. At checkpoints, the primary process must check whether the backup has finished creating the undo log before propagating the new data.

Figure 3 illustrates the two protocols performing a checkpoint. The AU protocol has to wait in the page fault handler to make undo logs but does not need to send new data at checkpoints. On the other hand, the DU protocol allows page faults to proceed without blocking but has to send updates at checkpoints.

3.4 Optimizations for Transaction-Based Applications

Two special function calls, "ignore_region" and "register_recover_function" are provided to transaction-based applications to enable better performance. The "ignore_region" function is used to tell the failover system to ignore some regions when making undo logs. The "register_recover_function" call is used by applications to register a special function which is called after the backup takes over. With these two calls, applications themselves can make undo logs for some data regions and recover them with the registered recovery function when the primary fails. For example, a database application can choose to ignore its data buffer. When the backup takes over, it only rolls back the application's control data. The recovery function provided by the application is called to rollback the data buffer. This mechanism may improve overall performance because database applications usually create undo logs and recover dirty data more efficiently.

For transaction-based applications, the "ignore_region" optimization has less of an effect on the DU protocol than the AU protocol. Although it is unnecessary to make undo logs for ignored regions, all modifications to these regions still need to be propagated to the the backup. The benefit for the DU protocol is minimal because it must still use page faults to find dirty pages and at the next checkpoint, it transfers data for each modified page whether it is ignored or not. The AU protocol has no overhead for replicating ignored regions to the backup because all modifications are propagated automatically.

Our failover system uses the 1-safe commit mechanism [22]. In a 1-safe design, the primary process goes through the standard commit logic and declares completion when the commit record is written to the local log. This mechanism provides the same throughput and response time as a single-node design. However, the 1-safe design risks the loss of transactions. This risk is higher with the DU protocol than with the AU protocol because AU propagates updates automatically as a side-effect of local writes to the log. Moreover, the risk can be completely eliminated with the AU protocol by having the primary wait for the backup's reply before declaring completion. To achieve similar a result with DU requires slightly higher overhead because the DU protocol needs to explicitly transfer the new log records to the backup first.

3.5 Primary Node Failure Detection

Primary node failure detection can be implemented using the classic heartbeat mechanism [37]. The primary machine periodically sends a sequenced message to the backup, over a network that is marked for internal communication. The backup raises a failure suspicion event after it detects a number of consecutive missing heartbeats from the primary. Once a failure suspicion is raised, a standard voting protocol can be used to confirm that the primary process is out of service, because the primary machine is down, the application process has crashed, or the link from the primary to the network is broken.

To implement a fast failover system, it is very important to have a failure detection mechanism that detects failures quickly and precisely. Most existing systems use alarm signals to periodically generate heartbeats. The main drawback of this method is that the alarm handler may not be triggered on exact intervals due to the operating system's process scheduling. As a result, the backup must wait for a long detection period before triggering a failure suspicion. The second drawback is the context switch overhead which limits the heartbeat frequency. For example, the Microsoft Cluster Service sends a heartbeat every 1.2 seconds and the detection period for a failure suspicion is 7.2 seconds [20].

Our system implements failure detection on the network interface. The network interface on the primary node periodically injects the network with a heartbeat message for the backup. When the network interface on the backup detects a number of consecutive missing heartbeats, it starts the failure suspicion process. Once the primary failure is confirmed, it generates an interrupt for the backup process to take over from the primary. In our prototype implementations, the primary sends heartbeats to the backup every millisecond and the detection period for a failure suspicion is 3 milliseconds.

4 Testbeds

To evaluate the two fast failover protocols, we require a platform that support virtual memory-mapped communication. We constructed two PC clusters, one connected with SHRIMP network interfaces [14, 13] and the other with Myrinet[3, 9, 7] network interfaces. The SHRIMP cluster provides both automatic update and deliberate update virtual memory-mapped communication, whereas the Myrinet cluster can only support deliberate update. However, the Myrinet cluster can better represent new generation systems than the SHRIMP cluster. Both systems run the Linux operating system. Table 1 compares the system configuration and basic operation costs on these two clusters.

	SHRIMP	Myrinet
CPU speed (Mhz)	Pentium 66	Pentium Pro 200
L2 Cache (KBytes)	256	512
DRAM (MBytes)	64	256
1 word transfer (μs)	8	25
4K Page Transfer (μs)	180	87
4K Page Copy (μs)	90	68
Page Fault (μs)	80	20
Page Protection (μs)	22	10

Table 1: System configuration and basic operation costs on the SHRIMP and Myrinet clusters.

4.1 SHRIMP Cluster

The custom network interface is the key system component of the SHRIMP cluster. It connects each PC node to the routing backplane and implements hardware support for virtual memory-mapped communication (VMMC) [14, 13]. The network interface hardware consists of two printed circuit boards because it connects to both the memory bus and the I/O bus. The memory bus board, called "snoop logic", simply snoops all main-memory writes, passing address and data pairs to the network interace.

4.2 Myrinet Cluster

The Myrinet cluster can only support deliberate update virtual memory-mapped communication [9, 7]. Myrinet is a high-speed local-area network or system-area network for computer systems. A Myrinet network is composed of point-to-point links that connect hosts and switches. The network link can deliver 1.28 Gbits/s bandwidth in each direction [3]. The PCI network interface is composed of a 32-bit control processor called LANai (version 4.1) with 256 KBytes of SRAM (Static Random Access Memory). The LANai processor is clocked at 33 MHz and executes a LANai Control Program (LCP) which supervises the operation of the DMA engines and implements a low-level communication protocol.

5 Performance Evaluation

To evaluate our fast failover method, we implemented the two failover protocols as user level libraries on our testbeds. The failover libraries use periodic timer signals to trigger checkpoints. The libraries are also responsible for detecting failures and automatically failing over to the backup.

The goal of our experiments is to answer three questions:

- How much is the system overhead?
- What is the maximum time for a checkpoint?
- How much does it take to failover applications from the primary to the backup?

We conducted several experiments with a variety of applications. The results on both platforms are similar. Therefore, we only present the performance evaluation on the SHRIMP cluster. We then discuss the impact of different system configurations. Finally, we use the results with two SPEC95 applications to discuss the requirement for the network interface design.

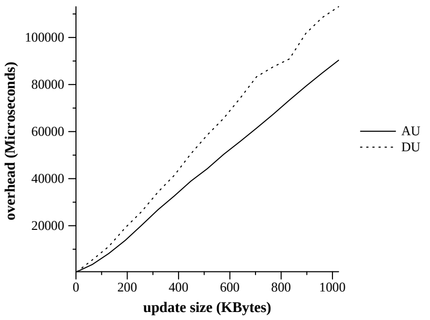

Figure 4: Checkpoint overhead vs. update size per checkpoint on the SHRIMP cluster with the AU and DU protocols.

5.1 Applications

The first set of experiments use three transaction applications: Debit-Credit (based on TPC-B), Order-Entry (based on TPC-C) and Postgres. The system successfully fails over all three applications with almost no modification to the applications' source code.

Debit-Credit and Order-Entry are based on TPC-B and TPC-C, which are two widely used transaction-processing benchmarks. We ran the two benchmarks on top of a lightweight, main-memory transaction library, Rio Vista [27]. The Rio Vista is modified slightly by inserting two function calls, "ignore_region" and "register_recovery_func", to improve the performance (See Section 3.4). We chose the largest possible database sizes which fit in main memory.

Postgres [35] is an object relational database developed at University of Berkeley. This application does not assume stable main memory. When a transaction commits, it flushes all dirty data modified by the transaction to disks with synchronous writes. The numbers are collected with the Postgres Wisconsin Benchmark.

5.2 Microbenchmark

To measure the system overhead, we first use a microbenchmark to quantify the relationship between the absolute system overhead and working set size for the two failover protocols. In this microbenchmark, every checkpoint interval modifies some pages by writing the first word of each page. We vary the number of modified pages from 1 page (4 Kbyte) to 256 pages (1 Mbyte). Figure 4 shows the relationship between the number of bytes touched per checkpoint interval and the overhead incurred per interval, which includes both the checkpoint cost and page fault handling time.

The minimum overhead is 185 μs per checkpoint for both protocols. This overhead is achieved with only touching a variable in the stack. Most of the 185 μs is used to send the stack to the backup. Since our implementation does not use copy-on-write for stacks because of the Linux limitation, both protocols have to explicitly transfer the stack to the backup (See Section 3.2. No page faults occur in this case.

The overhead with the AU protocol increases slower than the DU protocol when the working set increases. The AU protocol has almost constant checkpoint cost, but it takes 220 μs to handle the write protection signal for each page modified. The 200 μs includes 80 μs to invoke a page fault, 100 μs to make an undo log on the backup and 20 μs to change the page protection. With the DU protocol, each modified page adds 290 μs of the overhead, including 180 μs for sending the page to the backup, 80 μs for triggering a page fault and 30 for handling the page fault. Since the cost for each modified page using the AU protocol is smaller than the DU protocol, the AU curve grows 30% slower than DU.

In general, there is no fixed relationship between absolute overhead (time) and relative overhead (fraction of execution time). Real programs that touch a larger amount of data in an interval are likely to spend more time on computing than programs that touch only a small amount of data. Hence their checkpoint overhead will be amortized over a longer period of time. The main factor in determining relative overhead is locality. Programs that perform more work per touched page will have lower relative overhead than programs that touch many pages without performing much work. Therefore, the relative overhead is application dependent.

5.3 Relative Overhead

Figure 5 shows the relative overhead with different checkpoint interval lengths for the three applications. The automatic update failover protocol is able to take checkpoints every 0.1 seconds while adding only 3-12% overhead. The deliberate update protocol has higher overhead than the AU protocol with the two TPC benchmarks and similar overhead with Postgres.

With the Debit-Credit and Order-Entry applications, failover can be efficiently supported with the automatic update protocol. For example, the AU protocol has only 5% overhead when the two applications checkpoint every 0.1 seconds. If the tolerable overhead is 10%, the application can afford to checkpoint every 0.05 seconds. As checkpoints are taken less frequently, the relative overhead drops because fewer page faults occur.

The deliberate update protocol imposes higher overhead than the AU protocol on the Debit-Credit and Order-Entry applications. For example, the protocol has 66% overhead when checkpoints are taken every 0.1 seconds in the Debit-Credit application. To reduce the overhead below 10%, the application has to increase the checkpoint interval to more than 3 seconds. The DU protocol overhead is mostly spent on transferring dirty pages in the database buffers to the backup process. Since these buffers are "ignored" for making undo logs, modification to those buffers imposes no overhead with the AU protocol. However, with the DU protocols, each dirty page in the database buffers adds 80 μs to trigger the page fault and 180 μs to send the page to the backup.

In the Debit-Credit application, the relative overhead with the deliberate update protocol remains constant when varying the checkpoint interval from 0.01 to 0.1 seconds. The main reason is that this application modifies data records randomly. Although the number of checkpoints decreases by some factor, the number of pages modified during each interval is increased by the same factor. Small amount of data reuse happens after the checkpoint interval time is greater than 0.1 seconds and increases gradually with the interval time. As a result, the relative overhead drops as checkpoints are taken less frequently. The percentage overhead decreases earlier in Order-Entry than in Debit-Credit because the Order-Entry application exposes better data locality than the Debit-Credit application.

Both protocols have low overhead with Postgres. The relative overhead of both protocols is less than 12% with this application when the system is checkpointing every 0.1 seconds. Since Postgres does not assume stable main memory,

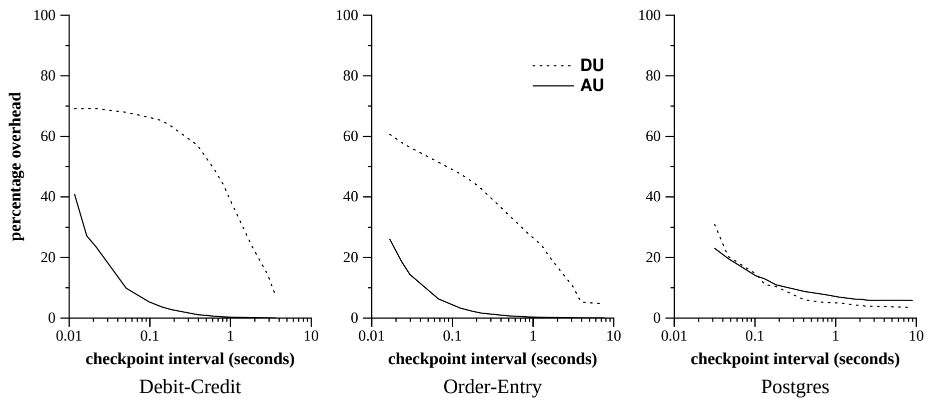

Figure 5: Relative overhead for varying intervals with three transaction based applications on the SHRIMP cluster. The Debit-Credit and Order-Entry benchmarks run on top of the Rio Vista main memory database [27].

it flushes dirty data to disks with synchronous writes at commit time. Accessing disks is usually two orders of magnitude slower than transferring data to remote memory. Therefore, in this application, the total execution time is dominated by disk I/Os rather than the checkpointing overhead.

5.4 Maximum Checkpoint Time

The maximum time for a checkpoint is very small in the automatic update protocol (See Figure 6). The reason is that most of the application data is automatically propagated to the backup as a side-effect of local writes in this protocol. At each checkpoint, the primary first sends the backup its important register values and the entire stack and then write-protects all un-ignored regions. The primary can proceed as soon as the backup replies. Since the cost of all these operations are independent of the number of modified pages, the absolute checkpoint time remains almost constant with different checkpoint interval time. The maximum time for a checkpoint is always less than 10 milliseconds with the AU protocol for all applications and checkpoint frequencies.

In contrast to the AU protocol, the maximum time for a checkpoint in the DU protocol is very high and mostly increases with the length of checkpoint intervals. In the DU protocol, dirty pages are transferred to the backup at checkpoints using explicit messages. Therefore, DU has to spend 180 μs for every page that is modified during the last checkpoint interval. In general, the number of dirty pages increases when the system checkpoints less frequently. The maximum and average time for a checkpoint are similar in the two TPC applications. But the maximum time is significantly higher than the average time in Postgres because writes are not evenly distributed across the whole execution time in this application.

5.5 Failover Delay

The worst case happens when the primary fails at the end of a checkpoint interval, that is, right before the primary propagates the last piece of data to the backup. In this scenario, the backup not only needs to roll back all changes, but also has to redo all the lost computation from the most recent checkpoint to the point when primary fails.

Table 2 gives the worst case failover delay with the two protocols under the assumption that the application can tolerate 10% or 20% of relative overhead. Although our applications can successfully failover from the primary to the backup and continue providing services, it is impossible to measure the worst case failover delay. Therefore, we calculate the worst case delay by adding the basic failover cost and the maximum time to make undo log and redo lost computation. The basic failover cost in our system is 4 milliseconds. The undo time is computed by multiplying the average cost for copying a page with the maximum number of page faults during a checkpoint interval. The recompute time is the smallest interval time to achieve the given relative overhead.

Application	Time in seconds			
	DU		AU	
	10%	20%	10%	20%
Debit-Credit	3.817	3.070	0.055	0.055
Order-Entry	4.069	3.224	0.071	0.027
Postgres	0.465	0.153	0.462	0.096

Table 2: Worst case failover delay for two different relative overheads (10% and 20%) on the SHRIMP cluster.

The automatic update protocol can failover within 0.01-0.4‘seconds while adding 10% of the overhead. If applications can tolerate 20% overhead, the failover time can be reduced to 0.01-0.1 seconds. The failover time with the deliberate protocol is 3-4 seconds for the two TPC benchmarks because the 10% overhead can only be achieved when checkpoints are taken less frequently than every 3 seconds. The DU protocol can failover within 0.01 to 0.4 seconds for the Postgres application.

5.6 Impact of Configuration Scaling

Starting from the point study on the SHRIMP cluster, one can roughly project the results on systems with better system configuration such as faster processor, higher memory bandwidth or higher network bandwidth. We only consider applications with uniform updates because they provide a lower bound of the relative checkpoint overhead for our failover scheme. The TPC benchmarks are two such applications.

For a transaction-based application with uniform updates, we use R to represent the transaction rate on a platform A. Then the computation time for each transaction is $\frac{1}{R}$. Suppose each transaction modifies P pages and the set of pages modified by each transaction is distinct. We

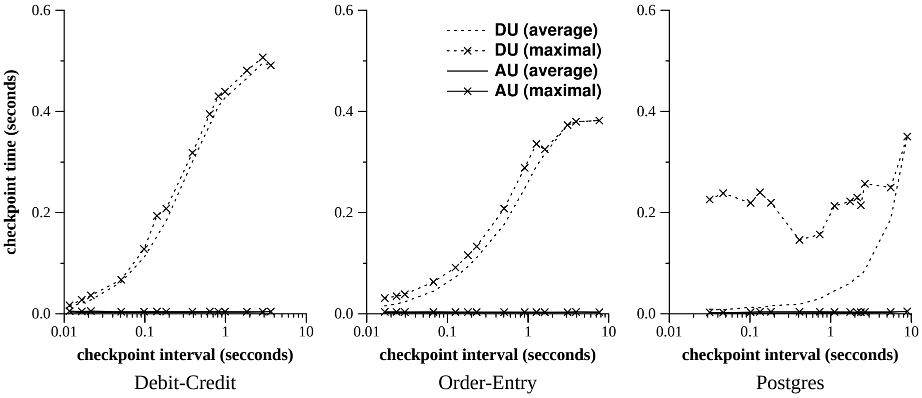

Figure 6: Maximum and average time for a checkpoint with three transaction based applications on the SHRIMP cluster.

assume that the system checkpoints every t seconds and the checkpoint overhead for each modified page on this platform is T, then the overhead for each transaction is $P \cdot T$. The relative overhead with this checkpoint frequency is

$$\frac{P \times T}{\frac{1}{R} + P \times T}$$

If the system can tolerate θ relative overhead, the failover system has to satisfy

$$T \leq \frac{\theta}{R \times P \times (1 - \theta)}$$

For the DU protocol, the checkpoint overhead for each modified page mainly consists of two parts: operating system overhead T_{os} and the communication overhead T_{comm}. T_{os} includes the time to trigger page fault and changing page protections. T_{comm} includes the time to transfer the modified page at checkpoints and send a 1-word undo log request to the backup.

Running an application on a new platform only changes the values for T and R. For example, on the Myrinet Cluster, one can use the numbers on Table 1 to calculate T_{os} and T_{comm} to be 30 μs and 112 μs, respectively. Therefore, T equals 142 μs on Myrinet, which is almost half of the cost on SHRIMP. We then measure the transaction rates without linking with the failover library. The Order-Entry benchmarks run at 11600 transactions per sec on the Myrinet cluster, which is twice higher than the transaction rate on the SHRIMP cluster (6000 transactions per sec). Substituting these numbers to the first formula described above, one can easily conclude that the deliberate update failover protocol has similar relative overhead on both platforms. The transaction rate with Debit-Credit benchmarks on Myrinet is also twice that on SHRIMP, therefore this application has the similar relative failover overhead on both platforms. Our experiments on the real platforms verify this analysis (See Figure 7).

A similar analysis can be used to project results on other systems with different configurations. However, this method only works for simple and regular applications like TPC benchmarks. For other applications that have irregular update patterns or non-deterministic computation, it requires a more complicated and accurate model to do the projection.

5.7 Network Interface Requirement

Besides the three transaction based applications, we have also used two SPEC95 applications: Go and Tomcatv. The

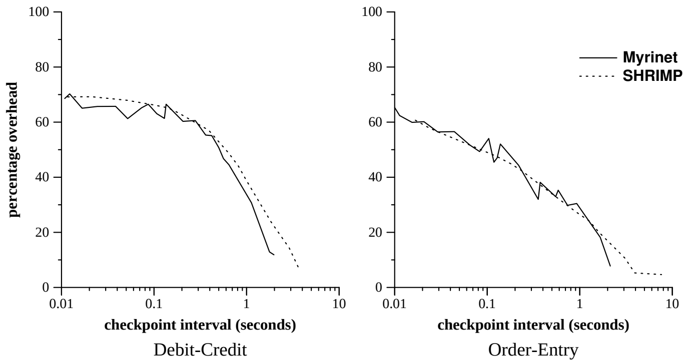

Figure 7: Comparison of relative checkpoint overhead on the SHRIMP and Myrinet clusters with the DU protocol.

Go is an integer benchmark, whereas Tomcatv is a highly vectorizable double precision floating point FORTRAN program. Although these applications are not mission-critical applications, they reveal certain properties that other applications do not show.

Figure 8 shows the relative overhead with these two applications on the SHRIMP cluster. Both protocols have less than 10% of the overhead for Tomcatv. The reason is that this application involves lots of floating point computation for each modified page. With the Go application, however, the automatic update failover protocol imposes much larger overhead than the deliberate overhead. For example, when the application checkpoints every 0.2 seconds, the relative overhead of the AU protocol is up to 13.2%, whereas the DU protocol has only 6.7% of the overhead. The main reason lies in the SHRIMP network interface design.

On the SHRIMP network interface, an Outgoing FIFO is used to provide flow control for automatic update. To prevent overflow, the network interface generates an interrupt when the amount of data in the FIFO exceeds a programmable threshold. The operating system then de-schedules all processes that perform automatic update until the FIFO drains sufficiently. On the SHRIMP cluster, the minimal Outgoing FIFO capacity is 1K bytes to prevent the case that the FIFO is quickly filled before the CPU recognize the threshold interrupt. This number is calculated by multiplying the memory write bandwidth by the hardware interrupt invoking time.

Although the outgoing FIFO in this system actually has 32 Kbytes, the number of interrupts due to flow-control is

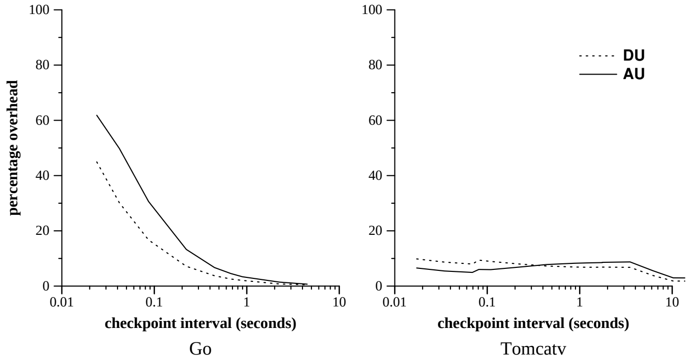

Figure 8: Relative overhead for varying intervals with two SPEC95 applications on the SHRIMP cluster

still significantly high when memory writes are very intensive such as in the Go application. When a threshold interrupt happens, the kernel has to spin for average 1.4 milliseconds to wait for the FIFO to drain to a special water mark, for example, half of the FIFO capacity. As a result, in some cases, more than 30% of the total execution time is spent on this flow control delay. This explains why the automatic protocol has much more overhead than the deliberate protocol with the Go application.

The mismatch between the FIFO filling speed and draining speed is the main reason for the high frequency of threshold interrupts and long spinning time. The memory bandwidth on the SHRIMP cluster is 50 Mbytes/sec, whereas the data transfer bandwidth is 25 Mbytes/sec, limited by the I/O bus bandwidth. Because the automatic update failover protocol needs to propagate most of data writes to the backup, it requires the network interface to reduce the bandwidth mismatch of the propagation datapath, especially for applications with burst writes such as the Go application.

Another limitation that our testbed imposes is that the AU mechanism requires using write-through caching strategy for all mapped virtual memory with AU operations. For certain applications, it is well known that write-through caching strategy requires more memory bandwidth than write-back. For example, in the application Tomcatv, the DU protocol has slightly less overhead than the AU protocol when checkpoints are taken more frequently than every 0.5 second. One way to avoid the write-through problem is to to use the write-back caching strategy for the AU mechanism of a virtual memory-mapped network interface. To make this method work well, it requires an efficient mechanism to flush dirty cache lines of a virtual address space at a checkpoint.

6 Related work

In the past, custom-designed fault tolerant hardware has been used to provide highly reliable and available systems. The state-of-art Tandem system [2, 1, 32, 18] is the first commercially available system designed specifically for online transaction processing applications. The Stratus system [33] presents the same inputs to two processor boards and uses a comparison logic on each board for failure detection. All these systems are custom-designed, therefore are very expensive and do not track technology trends well.

Recently, the fault tolerant system research has been focused on building highly reliable and available systems with a cluster of commodity hardware. For example, Microsoft cluster service (MSCS) [37, 20] takes a phased approach to build highly reliable and available clusters of PCs. Another similar system is the Digital TruCluster [19]. These systems usually take more than 10 seconds to failover an application and require applications to be responsible for checkpointing memory states to shared disks. This paper presents the novel approach of using virtual memory-mapped communication to reduce the failover time for clusters with minimal modifications to existing applications.

Other systems exploit a bus or broadcast network to implement fault-tolerant processes on top of an operating system. The work described in [4, 8, 30, 12] exemplify this approach. The most recent work of the Hypervisor based fault tolerance [5] uses a software layer that implements virtual machines to coordinate replicas on the primary and backup machines. Their system slows down applications performance by a factor of two. Although our study shares some similar issues with these systems, our system propagates data from the primary to backup either automatically or explicitly using virtual memory mapped communication mechanism.

Checkpointing and log-and-replay [10] are the two main techniques for reconstructing the state of a failed process. Checkpointing has been used for many years [6, 26] and in many systems [31, 25, 29, 17]. In this paper, we checkpoint to remote memory for fast failover.

PERSEAS is another work [28] similar to our study. It is a transaction library based on reliable main memory provided by mirroring the data at the remote memory. It differs from our work because it does not support failover of process state. It only guarantees that the database won't be lost when the machine crashes.

Previous results on the SHRIMP system [14, 13] show that automatic update outperforms deliberate update in few cases and the blocked mode with automatic update does not matter for most of applications. Our study creates a very good example for demonstrating the benefit of the automatic update hardware and exposes different design tradeoffs of the network interface.

7 Conclusions and Limitations

This paper presents a novel way to use the virtual memory-mapped communication model to reduce the failover time for cluster systems. Our system mirrors applications' virtual address space to the backup node either automatically or explicitly. The prototype implementations on two PC clusters with three transaction-based applications show that both protocols work quite well. Even the DU failover protocol that requires no special network interface support imposes less than 4-21% overhead when taking checkpoints every 2 seconds. If applications can tolerate 20% overhead, it can failover within 4 milliseconds in the best case and from 0.1 to 3 seconds in the worst case. This demonstrates that VMMC is a convenient and efficient communication mechanism to support fast cluster failover.

Our results also show that the AU failover protocol has less overhead and failover delay time than the DU failover protocol. The AU protocol is able to checkpoint every 0.1 seconds with only 3-12% overhead. If 10% overhead is allowed, it can failover applications within 0.01 to 0.4 seconds in the worst case. This indicates that some special network interface hardware support for automatic update is very useful for building fault-tolerant cluster systems. Besides the SHRIMP cluster, the automatic update failover scheme can also be modified for use in clusters built with other network hardware, such as MemoryChannel [21], Memnet [11] and etc.

We learned that although virtual memory-mapped communication supports fast failover for many applications, care must be taken when designing the automatic update mechanism. We found that the automatic update end-to-end bandwidth must be high enough to reduce the flow control overhead for applications that have intensive burst writes.

Our study has several limitations. The first limitation is that we only implemented a primary/backup process pair which can failover on a single failure. Although it is easy to extend our approach to a full cluster, we have not yet evaulated it on large-scale clusters. In this paper, we focused on processor failures since many techniques have been proposed to support failover when the disk or network fails. Another limitation is that we were not able to experiment with the proposed automatic update mechanism that allows the use of a write-back caching strategy. We believe this approach will improve the performance of the applications that did not work well with the current automatic update mechanism.

8 Acknowledgement

This project is sponsored in part by the Scalable I/O Initiative Effort under DARPA grant DABT63-94-0049 and grants from Sandia National Lab and Lawerance Livermore National Lab. We are grateful to Stefanos N. Damianakis and Scott C. Karlin for helping with the system setting up. This paper also benefits greatly from Stefanos N. Damianakis and Sanjeev Kumar's valuable suggestions.

References

[1] Bartlett et al. A NonStop Kernel. *SOSP'81*.

[2] Joel F. Bartlett. *A NonStop Operating System*. Tandem Computers, Inc., 1977.

[3] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[4] A. Borg, J. Baumbach, and S. Glazer. A Message System Supporting Fault Tolerance. In *SOSP'83*.

[5] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-Based Fault Tolerance. *ACM TOCS*, 14, February 1996.

[6] K.M. Chandy and C.V. Ramamoorthy. Rollback and Recovery Strategies for Computer Programs. In *IEEE Transactions on Computers*, pages 546–556, June 1972.

[7] Y. Chen, S. Damianakis, C. Dubnicki, and K. Li. UTLB: A Mechanism for Address Translation on Network Interfaces. In *ASPLOS*, 1998.

[8] Borg et al. Fault Tolerance Under UNIX. *ACM Transactions on Computer Systems*, 7, February 1989.

[9] C. Dubnicki et.al. Software Support for Virtual Memory-Mapped Communication. In *IPPS'96*.

[10] E. N. Elnozahy et.al. A Survey of Rollback-Recovery Protocols in Message Passing Systems. Technical Report TR 96-181, Carnegie Mellon University, 1996.

[11] G. S. Delp et.al. Memory as a Network Abstraction. *IEEE Network*, 5, July 1991.

[12] Greg Minshall et.al. An Overview of the NetWare Operating System. In *USENIX'94*.

[13] M. Blumrich et.al. Design Choices in the SHRIMP System: An Empirical Study. In *ISCA'98*.

[14] M. Blumrich et.al. A Virtual Memory Mapped Network Interface for the Shrimp Multicomputer. In *ISCA'94*.

[15] Peter M. Chen et.al. The Rio File Cache: Surviving Operating Systems Crashes. In *ASPLOS'96*.

[16] Peter M. Chen et.al. RAID: High-performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[17] Peter M. Chen et.al. Discount Checking: Transparent, Low-Overhead Recovery for General A pplications. Technical report, University of Michigan, July 1998.

[18] R. Chillarege et.al. Challenges in Designing Fault-Tolerant Systems. In *FTCS'91*.

[19] W. M. Cardoza et.al. Design of the TruCluster Multicomputer System for the Digital UNIX Environment. *Digital Equipment Corporation Technical Journal*, 8(1), May 1996.

[20] W. Vogels et.al. Scalability of the Microsoft Cluster Service. In *Proceedings of the 2nd USENIX Windows NT Symposium*, 1998.

[21] R. Gillett, M. Collins, and D. Pimm. Overview of Network Memory Channel for PCI. In *Proceedings of the IEEE Spring COMPCON '96*, February 1996.

[22] Jim Gray and Andreas Reuter. *Transaction Processing: concepts and techniques*. 1993.

[23] Yennun Huang and Yi-Min Wang. Why Optimistic Message Logging Has Not Been Used in Telecommunication Systems. In *FTCS'95*.

[24] Katzman J, A and et.al. A Fault-tolerant multiprocessor system system. *United States Patent 4,817,091*, March 89.

[25] David Johnson and Willy Zwaenepoel. Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing. In *PODC'88*.

[26] Richard Koo and Sam Toueg. Checkpointing and Roolback-Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, 13(1):23–31, January 1987.

[27] David E. Lowell and Peter M. Chen. Free Transactions With Rio Vista. In *SOSP'97*.

[28] A. E. Papathanasiou and E. P. Markatos. Lightweight Transactions on Networks of Workstations. In *ICDCS'98*.

[29] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent Checkpointing under Unix. In *Proceedings of the 1995 Winter USENIX Technical Conference*, 1995.

[30] M. L. Powell and D. L. Presotto. Publishing: A Reliable Broadcast Communication Mechanism. In *SOSP'83*.

[31] Kenneth Salem and Hector Garcia-Molina. Checkpointing Memory-Resident Databases. Technical Report CS-TR-126-87, Department of Computer Science, Princeton University, 1987.

[32] Siewiorek and Swarz. *The Theory and Practice of Reliable Systems Design*. Digital, Bedford, 1982.

[33] D. P. Siewiorek and R. S. Swarz. *Reliable Computer Design and Evaluation*. Digital Press, Burlington, MA, USA, 1992.

[34] J. H. Slye and E. N. Elnozahy. Supporting Nondeterministic Execution in Fault-Tolerant Systems. In *FTCS'96*.

[35] Michael Stonebraker. The Postgres DBMS. In *SIGMOD'90*.

[36] Robert E. Strom and Shaula Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, 3(3):204–226, August 1985.

[37] Werner Vogels and etc. The Design and Architecture of the Microsoft Cluster Service. *FTCS'98*.

[38] Michael Wu and Willy Zwaenepoel. eNVy: A Non-Volatile, Main Memory Storage System. In *ASPLOS'94*.