Dynamic Data Replication for Tolerating Single Node Failures in Shared Virtual Memory Clusters of Workstations

Rosalia Christodoulopoulou
Department of Computer Science
University of Toronto
Toronto, Ontario M5S 3G4, Canada
roza@cs.toronto.edu

Angelos Bilas
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario M5S 3G4, Canada
bilas@eecg.toronto.edu

Abstract

In this paper we investigate how shared memory clusters can take advantage of replication to tolerate single system failures. We start from a shared virtual memory protocol (GeNIMA) that has been optimized for low-latency, high-bandwidth system area networks. We propose a set of extensions that maintain shared data consistent in the presence of failures and support SMP nodes. Our scheme uses dynamic data replication to guarantee that no *shared* data is lost when a failure occurs. A failing node is removed from the system and the rest of the nodes recover dynamically and can continue with application execution. We deal both with data consistency and lock synchronization issues. Our approach leverages the low initiation overhead operations provided by modern system area networks as well as the availability of network bandwidth to guarantee data consistency in the presence of failures, and the low-latency operations for dealing with lock synchronization issues.

We have implemented the proposed scheme on a cluster of 32, Intel-based dual-processor systems interconnected with a Myrinet network. We are currently evaluating the performance implications of our protocol extensions.

1 Introduction

Clusters of high-end workstations and PCs have a number of advantages over more traditional multiprocessor systems. They follow technology curves well since they are composed of commodity components. They exhibit shorter design cycles and lower costs than tightly-coupled multiprocessors and they can benefit from heterogeneity. However, one of the most important aspects of modern clusters, especially in the context of commercial applications, is the potential for providing highly-available systems since component replication (e.g. memory, processors, network bandwidth) is not as costly as in other architectures. Moreover, due to the success of the shared address space (SAS) abstraction, many new applications are being developed for this abstraction. The majority of low-end servers are symmetric multiprocessor (SMP) systems and most vendors are designing larger scale hardware cache-coherent machines, targeting both scientific and commercial applications. These trends make shared memory clusters an attractive approach to providing availability for many classes of applications.

Our work addresses the problem of introducing fault-tolerance in modern shared virtual memory (SVM) clusters of symmetric multiprocessors (SMPs) used for parallel computation. In particular, it investigates how an existing all-software SVM protocol can be extended to tolerate efficiently single, fail-stop, node failures. In this paper, we specify the memory consistency guarantees provided by our protocol extensions and we present how these remain valid in the event of single node failures.

Our goal is to tolerate failures without introducing prohibitive overheads to the failure–free execution of applications. For this purpose, at this stage, we provide to applications semantics that guarantee that *no global* data is lost in the presence of failures. Extending our work for local thread data (stack) is the subject of our future work that will explore the tradeoff between the additional cost and the stronger semantics.

We employ existing protocol and communication layer mechanisms to replicate shared application and protocol data and to maintain their consistency in the presence of failures. Our scheme tolerates single node failures by taking advantage of the architectural features of modern clusters and the ability of SVM to provide transparent control over application shared data.

In particular, the main implementation axes of our work are: (i) We use low overhead communication operations to maintain replicas of global application and protocol data structures. These operations for the most part involve only low host–initiation overheads, whereas there is no host–processor intervention on the remote node. (ii) Our extensions support SMP nodes. (iii) We extend an existing SVM protocol to provide data replication without requiring specialized hardware support, non-volatile memory, disk storage, or other forms of stable storage. (iv) All system nodes are used for application processing and there is no separation of nodes to application and backup systems. (v) The semantics of our API in the presence of failures are provided to applications transparently, without requiring application modifications. (vi) The overheads incurred by our approach are the additional memory needed for maintaining replicas of shared data and the additional messages required to maintain their consistency. (vii) We eliminate extensive logs by essentially using short independent checkpoints at release operations, enhancing system scalability.

Our approach is novel in that it manages data replication dynamically throughout application execution. Unlike

our work, most research efforts that address the problem of fault-tolerance in this context, use logging to stable storage, shared disks, or non-volatile and persistent memory. In our approach, reliability is achieved by exploiting the *redundancy* that is inherent in a distributed system due to the presence of multiple processing elements and multiple copies of shared data. In particular, we employ *data replication* to dynamically maintain duplicates of shared memory and critical protocol information in the volatile memories of distinct machines that in case of failure, are used to maintain memory consistency. To tolerate single failures at any point in time, it suffices to maintain at least two copies for each shared page and for the global SVM protocol data structures.

In our design and implementation we use a state-of-the-art cluster interconnected with a high–bandwidth, low–latency system area network (SAN). Previous research in system area networks has led to the development of a number of user-space communication systems that can achieve performance close to the hardware limit [7, 16, 23, 24, 18, 3]. The base communication layer in our system is such a user–level communication system, Virtual Memory Mapped Communication (VMMC) [7]. VMMC is of particular interest because it provides direct communication between the sender's and receiver's virtual address spaces. Thus, data can be replicated with minimum communication cost and minimum interference with the rest of the computation. As a result, not only does VMMC provide a high-performance communication model that matches well our replication strategy, but also enhances scalability because it enables computing nodes to be used as backup nodes while performing useful computation.

We use as our starting point the GeNIMA shared virtual memory system [4], that has been optimized for this type of interconnects. GeNIMA is a home-based protocol [26, 19] that was shown to provide scalable performance to the 64–processor level [9]. We extend GeNIMA by introducing additional operations to guarantee consistency in the presence of failures. GeNIMA, as well as our extensions use low–overhead direct remote operations (read, write) provided by VMMC.

The rest of this paper is organized as follows. Section 2 introduces the guarantees we provide, describes the architecture of our system and specifies our assumptions. It also discusses briefly the potential of our proposed solution. Section 3 gives a concise overview of the original SVM protocol that is our starting point. Sections 4 and 5 present our protocol extensions and modifications. Section 6 covers most related research work. Finally, section 7 concludes with a summary of our proposed scheme.

2 System Model and Discussion

In this section we describe in more detail the guarantees we provide, the system we use and our assumptions, and we discuss various aspects of the proposed solution.

Guaranteed Semantics: As in the original SVM protocol, in a failure-free execution, the extended SVM protocol guarantees memory consistency at synchronization points. In the case of failure, it guarantees that, if a node F fails, then after all necessary recovery actions are complete:

1. Shared memory is release consistent.
2. All shared writes executed by some thread in F *before* the last release of F, have been performed (i.e. have become visible) at the corresponding home nodes in the system.
3. No shared write executed by a thread in F after its last release has been performed at a node other than F.
4. Any locks acquired by the failing node will be released.

The first three rules are related to shared memory consistency, whereas the last one is related to lock synchronization. These rules imply that, although after recovery the non-failed nodes are able to continue execution, the failed-node may not be present. In fact, our current implementation does not deal with recovering the failed node. It just excludes it from any further activity.

The motivation for our current approach is two-fold. First, it incurs lower overhead by not requiring to worry about local data. More importantly, many applications are written in a way that is very easy to adjust for this model. For instance, applications that employ work queues to dynamically assign tasks to threads will just assign uncomputed tasks to available threads.

It is part of future work to evaluate the overhead of using replication to also save local thread stacks and to guarantee the recovery of the failed process as well.

System architecture: The system architecture we use is a shared virtual memory cluster of symmetric multiprocessors (SMPs) that are connected by a high–speed system area network (SAN) with low bit error rates. Modern SANs provide very low latency for small messages (less than 10 μs) and bandwidth in the order of 100's of MBytes/s, limited by PCI bus implementations [8, 1, 5]. Cross-node SVM communication is usually based on user–level communication subsystems, that deliver to applications near–hardware performance.

In particular, our cluster is composed of 32 Intel–based, dual–processor systems, interconnected with a Myrinet SAN [5]. The communication layer we use is Virtual Memory Mapped Communication (VMMC) [7]. VMMC provides direct data transfer between the sender's and receiver's virtual address spaces, it tolerates transient network fabric errors by using packet retransmission, and guarantees FIFO message delivery between any two processes in the system.

VMMC operations return an error if a remote node is unreachable. When an error is returned, it is guaranteed that any subsequent communication with the same node will not be performed and it will return with error. However, it is not guaranteed that all previous operations have completed successfully, unless a response is received by the remote host. For instance, the reply of a successful remote fetch operation would guarantee that all previous operations have succeeded as well. These send/receive semantics are similar to TCP sockets.

Failure model: We assume that nodes are subject to failures and *fail-stop*, that is, they fail only by stopping and do not exhibit any incorrect behavior while operating. We consider *single–node* failures only. We assume that individual process or other software failures exhibit themselves as failures of their corresponding node. We do not deal with permanent network failures (in cables, switches). Transient error failures are resolved by VMMC as mentioned above.

Failure detection: We exploit the VMMC semantics to provide a reliable failure detection mechanism. As mentioned, VMMC operations that transfer data return an error when the destination node is unreachable. Since we assume the absence of permanent network failures, this is equivalent to a remote node failure. Thus, a node failure is detected when a VMMC read/write operation to that node fails which is indicated by a specific negative value returned by the respective VMMC call.

When nodes do not communicate and need to wait for a remote response, they send heart–beats (plain VMMC read operations) to detect possible failures. Heart–beats are separated by a *timeout period* during which a process that is waiting for the response spins before attempting the next heart–beat. This timeout mechanism ensures that failure detection happens sufficiently soon to prevent processes from long delays but also, from suspecting nodes too early which could incur unnecessary communication overhead.

Our scheme performs on demand failure detection and recovery and does not require global communication or synchronization for failure detection and recovery. Nodes detect failures independently as a result of failing to establish communication with the failed node.

Discussion: The most advanced fault-tolerant techniques based on rollback recovery in home-based DSM systems are log-based. To ensure deterministic execution replay after a failure, they maintain logs of protocol data exchanged (preferably in the volatile memory of peer processes), as well as checkpoints (preferably independent) of shared pages and protocol data to stable storage. The main drawbacks of these methods are that their efficiency is highly dependent on the checkpointing policy used and that they require additional techniques for controlling the size of the volatile logs and for garbage collecting the checkpoints kept in stable storage.

In our scheme, the memory consistency guarantees make possible the recovery of a failed process *without* protocol data logging, checkpointing of shared data, global coordination or stable storage support. Essentially our design replaces logs with independent short checkpoints at each release operation.

The basic idea in our scheme works as follows: Suppose that node F fails. For a process in F to recover, two things are required: first, the state of shared memory and second, the state of the process' execution, both at the point when execution is resumed after the failure. As far as the shared memory state is concerned, our extended protocol guarantees that the memory remains consistent after the failure and that no shared memory write executed in F since its last release has been performed at any node other than F. This memory consistency guarantee makes clear that the most suitable point to resume execution after a failure is the point after the last release performed by node F. The fact that at that point the memory is already consistent eliminates the need to restore a coherent memory state after a failure and frees processes from checkpointing shared pages during failure-free execution.

In our system, although after recovery the non-failed nodes are able to continue execution, the failed-node is not present. As already mentioned, recovery of the failed process by saving local thread stacks is part of our future work. However, we briefly discuss next how this can be achieved.

In order to restart the failed process from the point of last release, processes need to take checkpoints of their local execution state before a release. However, these checkpoints are completely independent and have minimal memory requirements (the stack is usually a few KBytes). They can be saved to the volatile memory of remote nodes, using a primary-backup data replication scheme. Alternatively, since their space requirements are minimal, process execution state may even be saved to persistent memory.

Under these circumstances, recovery is straightforward. First, the shared memory is restored: remote shared pages can be fetched from their home nodes, while the home pages of the failed node F itself can be retrieved from a backup home. Next, execution is resumed. For lock acquires during execution replay, locks can be re-acquired from their last releasers. As for the timestamp of each such lock, it can be retrieved from its last releaser: at the releaser's side, the timestamp of the lock is saved and has not been changed in the meanwhile. Moreover, all read accesses and writes until the point of failure, can be re-executed safely given that the execution replay starts after the last release performed locally and there has been no other release until the point of failure.

3 Original SVM Protocol

The original shared virtual memory protocol, GeNIMA [4] is based on home–based lazy release consistency (HLRC) [26] and is designed to take advantage of a number of architectural features in modern clusters and system area networks.

In order to comply with the partial order requirements of LRC for shared memory accesses [10], the application execution of each processor on each node is partitioned into *time intervals* that are delimited by consecutive release operations executed by threads on the same SMP. During a time interval, all local page updates are recorded into a common *update-list*.

Shared pages in GeNIMA are assigned a *home* node, according to HLRC [26], to which writers of the page send their updates for that page eagerly, upon a release. Nodes propagate page updates in the form of *diffs*, which consist of the updates performed by the releasing node to the version of the page before its first write (also called the *twin*). Diffs successfully address the problem of false sharing and allow multiple writers to write to different parts of the same page without intervening synchronization.

Lock synchronization in GeNIMA, as in many shared virtual memory systems, is based on a queuing lock algorithm. In order to manage locks among distinct SMP nodes, each shared lock is assigned a *home* node which handles requests for that lock by maintaining a virtual queue of the lock's requesters. In practice, the home node needs only record the *tail* of the queue: when a lock request is sent to the lock's home, the home forwards the request to the last requester (i.e. the node recorded in the tail of the queue), and updates the tail accordingly. The current owner of a lock does not release it unless it receives a remote request and provided that there is no pending local request for the same lock. Exchange of locks *within* an SMP requires no external or internal message exchange.

In SVM systems, processors cycle through *acquire–compute–release* cycles. During an acquire operation, the acquiring processor must ensure that all shared accesses that *precede* the acquire according to the partial order defined by LRC, have also been performed locally. For this reason, the

processor *fetches* from each remote node the list of updates which are needed for this synchronization step and *invalidates* the corresponding pages. The way the acquirer determines the pending updates is through comparison of its own and the releaser's *lock timestamp*, a per-node vector indicating the portion of every node's updates that have been performed locally.

A subsequent access to an invalidated page triggers a page fault that results in remote fetching the latest version of the page from its home node. In this context, we consider that a page *version* is a vector timestamp indicating which modifications applied by each node on that page are included in the currently available copy of the page. Also, pages are fetched in their entirety.

When a thread performs a lock release, it ends the current time interval by placing (*committing*) all updated pages in a protocol data structure, indexed by the interval number. After committing its update intervals, the releasing process computes and sends the diffs of the updated pages to their home nodes. This scheme enables multiple releases to be performed in parallel by different threads on the same SMP node.

4 Maintaining Shared Memory Consistency

Data structure extensions: In consideration of a single-failure tolerant system, we extend the original home-based SVM protocol to assign to each shared page *two* distinct homes, namely the *primary* and *secondary* home. Duplicate home pages should be up-to-date and consistent at the end of each release operation, so that if one home fails, subsequent home page requests will be serviced by the second home.

Control flow extensions: Before a processor can continue execution past a lock acquire, all preceding updates (according to the partial order defined by LRC) must be performed locally, or equivalently, all modified pages must be invalidated. Which are the specific update intervals required to be invalidated is determined by the lock timestamp sent to the acquirer of the lock by the releasing process. Next, these intervals must be acquired. In GeNIMA, two alternative mechanisms are provided for transferring update intervals among nodes. With the first mechanism, upon a lock acquire, a node *fetches* from remote nodes the update intervals it requires. Alternatively, the releasing node can *broadcast* its update intervals before a release. In the extended SVM protocol, update intervals are broadcast before the diff propagation stage.

As mentioned in section 3, GeNIMA also permits concurrent releases by multiple threads on the same SMP node. This makes possible the eager propagation of page updates that do not belong in the view of the releasing thread because they were performed *after* the thread committed its updates. In case of failure, it is clear that this eager update propagation scheme can violate the third consistency guarantee.

In the extended SVM protocol we prevent this scenario by: a) *locking* (i.e.disabling access to) pages while invalidating their update intervals and b) by *blocking* any write page fault handling for locked pages until they are unlocked. Pages are unlocked after their diffs are sent. It is thus guaranteed that new writes to pages that belong to the intervals committed by an outstanding release will not be executed and thus, will not be recorded in the update list until that release operation has been complete. With page locking, concurrent releases are still possible provided that the intervals diffed by distinct threads are disjoint.

Given that diffs may be performed concurrently by multiple threads, before releasing a lock, a thread must verify that *all* of the diffs for its updates since the last release have been propagated, especially these diffs that were being performed concurrently by other threads on the same SMP. This verification works as a safeguard against releasing a lock before the diff propagation is actually complete.

Home page replication implies that upon a release, page diffs must be propagated to both homes nodes. Thus, in contrast to the original SVM protocol where home nodes do not create diffs for their own pages, diffs for a modified page must now be computed and propagated even when the writer is a home for that page.

To tolerate single failures during diff propagation, we employ the following two-phase diff propagation scheme. In the first phase, for all updated pages, diffs are computed and sent to their primary homes, while in the second phase the same diffs are sent to the secondary homes. At the sender's side, diffs are saved locally temporarily so that they need not be recomputed in the second phase, and they are discarded at the end of the second phase.

Before the diffs of a page are propagated to either of its homes, a special flag is written to the remote page's timestamp, indicating that the page is *invalid* because it is currently being updated. After the corresponding phase is complete, the remote page's timestamp is updated with the value of the writer's timestamp. As a special case, for pages whose primary (secondary) home is the diffing node itself, diffs still need to be written twice, but both times at the same node, the secondary (primary) home of each page.

For each process to take appropriate actions in case of failure, it is necessary to be able to determine at which point in execution, and if applicable, in which phase of the diff propagation, the failure occurs. For this reason, each time a thread on a node P completes both phases of diff propagation, it records its last diffed interval in a variable, namely $lastRelease$, at some remote node. Also, each time a thread on P completes the *first* phase of diff propagation, it records its last diffed interval in a variable, namely $nextRelease$, at the same backup node. Since releases performed by any thread on the same SMP are recognized as *node* releases, there is no need to backup separate information for each different thread.

Recovery actions: At any point in time, $lastRelease$ indicates which page diffs have been applied by P to both the primary and secondary copies, while $nextRelease$ indicates which page diffs have been applied to the primary copies only. If $lastRelease < nextRelease$ then the diffs for pages between intervals $lastRelease$ and $nextRelease$ have been sent to the primary homes, while the update of the secondary homes is still pending.

If a failure occurs within the first phase of diff propagation and in particular, after diffs have been propagated to all primary copies but before all timestamps are updated, invalid timestamps can be restored. Given that the update intervals are *broadcast* by the lock releasers before diff propagation, each node can determine the timestamp of each updated page based on the order number of the update interval it belongs to.

Correctness: *The three memory consistency guarantees are valid at all times.* In a failure-free execution this is achieved by the semantics of the LRC protocol which are also preserved in the extended protocol. In the presence of failures, we distinguish two cases: either the failure occurs during a release operation, that is, at some point during the diff propagation stage and the final lock release, or at any other point in execution. In the latter case, it is clear that all shared writes executed by some thread on the failed node before its last lock release, have already been performed at the corresponding home nodes according to the semantics of LRC. In the former case, consistency is maintained by ensuring that, either all updates executed in the current time interval (i.e. the interval ended by the ongoing release operation) are performed, or none is. In other words, either all corresponding diffs are propagated, or none is.

This requirement is satisfied by the two-phase diff propagation scheme. There are three different cases when the diffing node may fail: 1) when application of diffs to the primary home copies of the modified pages is still in progress, 2) when diffs have been sent to all of the primary copies, but their timestamps' update is still in progress, 3) after the first phase is complete. If the failure occurs in step 1, then any updates sent by the failed node so far are ignored, and the previous version of the shared pages is used, as if the failed node had not acquired the lock. On the other hand, if failure occurs in either step 2 or 3, the release is considered valid since all updates have been propagated.

In particular, let A be a node performing its diffs and R be a remote node attempting to fetch a home page. First, R checks the timestamp of the home page, ts. If the value of ts is valid and the version of the page is at least equal to the requested one, then the page is fetched. Otherwise, if ts is invalid, the requesting node checks whether A is still alive. If this is so, then R waits until the correct page version is sent to the home, else A has failed in either step 1 or 2. This is decidable, based on the values of $lastRelease$ and $nextRelease$, as explained above. If the failure occurred in step 1, R fetches the page from its secondary home that still keeps the previous version of the page. The same holds if the acquirer is the primary home node of the page. If the failure occurred in step 2, R may safely fetch the primary copy since diffs have been sent to all of the primary homes and hence, the release is considered complete. R also restores the home page's timestamp.

The two-phase diff propagation ensures that no partial but only complete diffs are propagated during a release. The additional restriction imposed concerning page locking during diff propagation, ensures that upon a release operation, no updates are propagated other than those that were performed and committed within the time interval of that release. Thus, the third guarantee is also satisfied.

As a result, the shared memory remains consistent and available at all times, and active processes can continue executing past a failure.

Performance implications: To implement the above extensions we use mostly asynchronous remote deposit operations in VMMC. These operations incur very low host-processor overhead (in the order of 2μs) and do not require interrupts or other remote host-processor intervention. Thus, the additional overhead in most cases is the host overhead in initiating the direct remote operations.

A more important issue is the computation of diffs for home pages, which was not present in the original SVM protocol. Evaluating the exact impact of this operation on performance is part of the next step in our work. Finally, it is interesting to see to what extent the restriction imposed on eager diff propagation limits concurrency in practice.

5 Lock synchronization

Another consideration when nodes are subject to failures is how, in case of failure, the locks being held at the failed node are managed after the failure. We decided to use a primary-backup home scheme where the primary home handles lock requests similarly to the base queuing lock algorithm, while the secondary home is used as a backup node that takes over the primary in case of failure.

The queue-based lock algorithm remains the algorithm of choice because of its advantages: it is potentially scalable to large numbers of processors, it minimizes the memory contention and network load due to lock requests, it provides low latency and a reasonable degree of fairness, it prevents starvation, it fits well in multithread environments, and it incurs low storage overhead.

In this work we extend the queue-based lock algorithm to tolerate single–node failures as follows.

Data structure extensions: Each lock is assigned *two* distinct homes, namely the *primary* and the *secondary* home. Except for the tail of the queue of requesters, the primary home now also records the *second to last* node in the queue. This information is replicated at the secondary home.

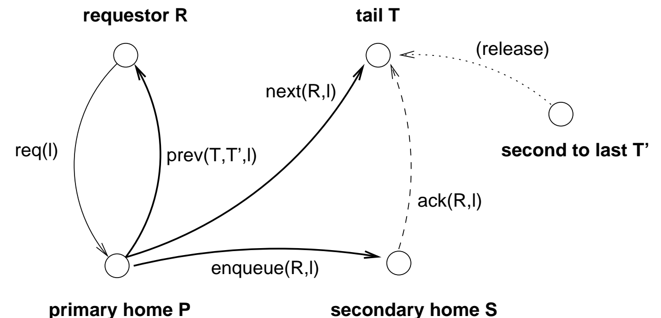

Figure 1: Extended queuing lock algorithm

Control flow extensions: In order to acquire a lock l, a requester node R sends a request message to the lock's primary home, P (Figure 1). When the primary home receives the lock request, it atomically updates its record of the last and second to last node in the queue and subsequently sends three messages: a) a message to the requester R, containing the node id's of the *two* nodes that precede R in the queue, b) a message to the (former) tail of the queue containing the id of the new requester (to which the lock should be forwarded), and c) a message to the lock's secondary home containing the id of the new requester. As a result of this third message, the queue information at the backup node is updated and an *acknowledgement* message is sent by the secondary home to the (former) tail of the queue. The knowledge of the id's of the two preceding nodes in the queue is necessary for resolving failures.

In order to guarantee memory consistency, a node is free to forward a lock to the next node only if it has received both the forward message from the primary node *and* the ack message from the secondary home. This restriction ensures that the queue information at the two homes is consistent whenever a lock is transferred between nodes. Figure 1 depicts the exchange of messages under a failure-free execution.

Recovery actions: The update of the secondary home's queue ensures that, in case of failure of the primary home, lock requests can be handled directly by the lock's secondary home without any violation of the semantics of the lock algorithm, and with no additional complexity.

A node that timeouts waiting for a lock, may verify whether the delay is due to its preceding node's failure since it is aware of its id. If this is indeed the case, and since the waiting node cannot tell whether the failed node was still holding the lock when the failure occurred, the same node can send a notification message to the *second to last* node in the queue. The latter will then forward the lock to the new destination, either immediately (if it has released the lock already), or upon releasing the lock (if it is still waiting for or using the lock). If the failed node was the first one to acquire the lock, the next requester grabs the lock immediately.

Correctness: We show now that the extended lock algorithm tolerates any case of failure. The failure of a lock's primary home node is detected by any node that requests that lock and timeouts while waiting for the home node to reply. In that case, the lock request is simply resent to and serviced by the secondary home. Even if the primary home fails after forwarding the lock request to the last node in the queue, but *before* updating the secondary home, the extended algorithm avoids inconsistencies between the actual distribution of locks and the queue information kept at the secondary home: since no acknowledgement is sent in regard to the new request, the lock will be blocked at the last owner of the lock. Finally, a node failure does not affect the lock synchronization as far as its use as a backup node is concerned. After a secondary home's failure, lock requests are serviced by the primary home exclusively, and lock holders release the locks without waiting for an acknowledgement.

As already mentioned, the failure of a node that fails while waiting for, or holding a lock, can be detected by the next node in the queue of requesters, which as a result, sends a notification to the second to last node in the queue if that one exists, otherwise it simply acquires the lock. In any case, eventually, all locks held by the failed process will be released. For the release consistency constraints to be satisfied in case the current *holder* of a lock fails, it is important for the next acquirer to use the lock time vector that was recorded by the last releaser. To address this, before a release, the releasing node saves its lock time vector at a remote node from which its lock time vector can be retrieved in the event of failure. The remote node used to backup this information is the primary home of the lock provided that this one is different from the releasing node; otherwise, it is the lock's secondary home.

Performance implications: In our scheme, as shown in Figure 1, the critical path to acquire a lock in the failure free case is still 2 hops as in the original protocol. However, the home of the lock needs to forward two messages (to the tail and the secondary home) and thus generate two interrupts, one on each node. Finally, the requester needs to wait for two replies as opposed to one in the original scheme. The overhead at the home (to send the second message) consists simply of a send initiation. The overhead at the secondary home is also low in terms of communication. However, the extra interrupt may have a more significant impact, which we are currently evaluating.

6 Related Work

There is a large body of work in fault tolerance in a number of research areas. In the next few paragraphs we summarize the most recent research that is most relevant to our work.

Sultan et al. [22], [21] follow a *log-based* approach to tolerate single node failures in a home-based, lazy release consistent SVM cluster of PCs. In particular, they use *volatile logging* of protocol data combined with *independent checkpointing* to stable storage for replaying execution in case of failure. Because their proposed scheme is log-based, their work focuses on how to dynamically optimize log trimming and checkpoint garbage collection in order to control efficiently the size of the logs and the number of checkpoints kept. In contrast to our approach, their proposed scheme does not address the problem of storage support, while its effectiveness is dependent on the application running on the SVM system, and specifically on the checkpointing behavior of the individual processes. This introduces the additional problems of balancing the amount of recovery state held across the system, and of implementing specific, application-driven checkpoint policies.

In [6], Costa et al. have extended Treadmarks [11], a lazy release consistent, distributed shared memory (DSM) system, to introduce single fault-tolerance support in a cluster of uniprocessors. Their algorithm is based on logging the data dependencies (due to remote synchronization operations) in the volatile memory of peer processes and uses independent checkpointing to stable storage to reduce recovery time. Similarly to the previous scheme, this algorithm also faces the problem of bounding the size of checkpoints and logs. This is handled by exploiting the global garbage collection operation already performed by Treadmarks, however at the cost of efficiency, since this operation requires *global* synchronization. Our approach is clearly different from the log-based ones in that we eliminate extensive logs by essentially using short independent checkpoints at release operations, thus avoiding the problems of checkpoint garbage collection and log trimming, and enhancing system scalability. Also, our system supports SMP nodes and requires no global synchronization and no stable storage support.

Zhou et al. [25] have investigated how virtual memory-mapped communication can be used effectively to reduce the failover time of single nodes on clusters used for running time–critical applications, like transaction-based applications. They have implemented two failover protocols based on a primary-backup node approach: using VMMC, the primary process transfers directly to the backup process' volatile memory the modifications of its application data, as well as periodic checkpoints of its execution environment in order to enable rollback recovery in the case of failure. Although this work targets different application domains to ours, the experimental results provide evidence that volatile logging using the VMMC model can be used on clusters to achieve reliability while maintaining efficiency, as opposed

to traditional techniques based on stable storage support.

Kermarrec et al. [12] have demonstrated that the natural replication that comes with a DSM system can be exploited to provide efficiency and scalability with the number of processors in the system. In particular, they have used data replication to extend a standard *sequential consistency* protocol, used by DSM on an Intel Paragon, with a recovery scheme that avoids stable storage support. For each shared page, their scheme maintains at least two *recovery* copies that contain the version of the page since the last global checkpoint and are kept on distinct node memories. Each shared page is assigned a *manager node* that is responsible for maintaining coherence information and copy-set data about that particular page. The manager node forwards page requests to the current owner of the page and performs all invalidations necessary to maintain coherence of the page's copies. Also, global checkpoints are taken to establish recovery points using a *globally coordinated*, two-phase commit protocol. In case of failure and after the failure is broadcast, all nodes in the system rollback and take actions to restore the latest recovery point and to reconfigure the system. This involves heavy communication and use of broadcasting especially when a manager node fails, since there is no use of a *backup* manager node during failure-free execution. The basic extended coherence protocol described in [12] has also been implemented on a SVM cluster of PCs interconnected with an ATM local area network [14]. In our system, in terms of data replication overhead, the home-based variation of lazy release consistency, alleviates the communication cost due to the management of replicated shared pages by reducing both the number of messages and the amount of data exchanged. Also, our scheme avoids global checkpointing overall. Upon a failure, active processes do not rollback and continue with application execution. Shared pages hosted at the failed node are retrieved *on demand* from the backup home with no need of further recovery actions or coordination since it is ensured that both homes are kept up-to-date during failure-free execution.

Plank et al. [17] have investigated the benefits of eliminating stable storage support in the context of fault-tolerance techniques by introducing *diskless checkpointing*, a checkpointing technique used for rollback recovery on networks of workstations that communicate by message passing. Diskless checkpointing avoids stable storage support, and exploits memory and processor redundancy to tolerate single processor failures. Although this scheme makes use of dedicated backup processors that do not participate actively in the computation and is based on coordinated checkpointing, diskless checkpointing algorithms when compared with their disk-based counterparts, achieve comparable checkpoint overhead and significant improvement in checkpointing latency and recovery time. Hence, the experimental results verify the fact that stable storage is indeed a major cause for performance degradation, and thus its elimination is a promising optimization technique for improving performance.

[2, 13, 20] have investigated various aspects of fault tolerance in contexts that differ from our work either in the underlying technology (such as older generation interconnection networks), the goals (such as to compute amount of replication needed to achieve fault tolerance), or their methodology (theoretical analysis and simulation without actual system implementation). Finally, a survey of recoverable distributed shared virtual memory systems is presented in [15].

7 Conclusions

This paper presents how a lazy release consistent SVM protocol on a commodity cluster may be extended efficiently to guarantee memory consistency in the event of single failures. In contrast to past techniques, the proposed scheme targets efficient execution in the absence of failures by eliminating the traditional bottleneck of stable storage support and by using the efficient virtual memory-mapped communication model. Reliability is achieved through shared memory data replication in the volatile memory of at least two nodes across the system. In case of failure, no memory copying or reloading data from disks is needed to recover the memory state of the failed node, and as a result, the failover delay is potentially reduced. Also, the system overhead relative to the total execution time of applications is kept minimal by tightly integrating the fault-tolerant extensions with the original SVM protocol.

We have implemented our extended protocol and we are currently evaluating its performance by conducting experiments on a local area cluster of 32 Intel-based, dual-processor systems interconnected by a Myrinet SAN. We are mostly interested in measuring the execution time overhead of our protocol extensions during failure-free executions and the failover time in the case of failures.

Based on the performance results we are planning to apply additional protocol optimizations to cover a wider spectrum of failure scenarios, including multiple failures. Finally, we would like to investigate the issue of replicating thread stacks to achieve transparent failure recovery for applications that do not follow the semantics provided by our modified protocol and maybe willing to incur additional overheads.

8 Acknowledgments

We would like to thank Peter Jamieson for assisting with various aspects of using the cluster during the implementation. This work was conducted with the support of Natural Sciences and Engineering Research Council of Canada, Canada Foundation for Innovation, Ontario Innovation Trust, the Nortel Institute of Technology, and Communications and Information Technology Ontario.

References

[1] http://www.emulex.com/newsroom/art/index.html.

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

[3] S. Araki, A. Bilas, C. Dubnicki, J. Edler, K. Konishi, and J. Philbin. User-space communication: A quantitative study. In *Proc. of SC98: High Performance Networking and Computing*, Nov. 1998.

[4] A. Bilas, C. Liao, and J. P. Singh. Accelerating shared virtual memory using commodity ni support to avoid asynchronous message handling. In *Proc. of the 26th International Symposium on Computer Architecture*, May 1999.

[5] 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.

[6] 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.

[7] 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. A short version appears in IEEE Micro, Jan/Feb, 1998.

[8] R. Gillett, M. Collins, and D. Pimm. Overview of network memory channel for PCI. In *Proceedings of the IEEE Spring COMPCON '96*, Feb 1996.

[9] D. Jiang, B. Cokelley, X. Yu, A. Bilas, and J. P. Singh. Application scaling under shared virtual memory on a cluster of smps. In *Proc. of the 13th ACM International Conference on Supercomputing (ICS'99)*, June 1999.

[10] 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.

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

[12] 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.

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

[14] C. Morin, A. M. Kermarrec, M. Banatre, and A. Gefflaut. An efficient and scalable approach for implementing fault tolerant DSM architectures. *IEEE Transactions on Computers*, 49(5):414–430, May 2000.

[15] C. Morin and I. Puaut. A survey of recoverable distributed shared virtual memory systems. In *IEEE Transactions on Parallel and Distributed Systems*, volume 8, pages 959–969, 1997.

[16] S. Pakin, V. Karamcheti, and A. Chien. Fast Messages (FM): efficient, portable communication for workstation clusters and massively-parallel processors. *IEEE Concurrency*, 1997.

[17] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–1001, 1998.

[18] L. Prylli and B. Tourancheau. BIP: a new protocol designed for high performance. In *PC-NOW Workshop, held in parallel with IPPS/SPDP98, Orlando, USA*, March 30 – April 3 1998.

[19] R. Samanta, A. Bilas, L. Iftode, and J. P. Singh. Home-based SVM protocols for SMP clusters: Design and performance. In *Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4)*, 1998.

[20] 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.

[21] F. Sultan, T. Nguyen, and L. Iftode. Limited-size logging for fault-tolerant distributed shared memory with independent checkpointing. Technical report, Dept. of Computer Science Technical Report DCS-TR-409, Rutgers University, Feb. 2000.

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

[23] H. Tezuka, A. Hori, and Y. Ishikawa. PM: a high-performance communication library for multi-user parallel environments. Technical Report TR-96015, Real World Computing Partnership, 1996.

[24] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proc. of the Nineteenth International Symposium on Computer Architecture*, 1992.

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

[26] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared memory virtual memory systems. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation (OSDI'96)*, pages 75–88, 1996.