Dynamic Data Replication: an Approach to Providing Fault-Tolerant Shared Memory Clusters

Rosalia Christodoulopoulou[1], Reza Azimi[2], and Angelos Bilas[2]

[1]*Department of Computer Science,*
University of Toronto,
Toronto, Ontario M5S 3G4, Canada
roza@cs.toronto.edu

[2]*Department of Electrical and Computer Engineering,*
University of Toronto,
Toronto, Ontario M5S 3G4, Canada
{azimi, bilas}@eecg.toronto.edu

Abstract

A challenging issue in today's server systems is to transparently deal with failures and application-imposed requirements for continuous operation. In this paper we address this problem in shared virtual memory (SVM) clusters at the programming abstraction layer. We design extensions to an existing SVM protocol that has been tuned for low-latency, high-bandwidth interconnects and SMP nodes and we achieve reliability through dynamic replication of application shared data and protocol information. Our extensions allow us to tolerate single (or multiple, but not simultaneous) node failures. We implement our extensions on a state-of-the-art cluster and we evaluate the common, failure-free case. We find that, although the complexity of our protocol is substantially higher than its failure-free counterpart, by taking advantage of architectural features of modern systems our approach imposes low overhead and can be employed for transparently dealing with system failures.

1. Introduction

Clusters of general purpose, inexpensive machines interconnected by high-speed communication networks are currently widely used for parallel computation and as backend processing servers for a growing number of commercial applications. Until recently, the focus of research efforts has mainly been on two critical issues, namely, programmability [19, 8, 16] and performance [4, 30]. However, as clusters become prevalent and larger in size, the focus of attention moves toward other system characteristics and especially, toward improving the reliability and availability of such systems. The reason for this renewed interest is that although system performance has reached levels that are adequate to serve a number of existing and new applications, system reliability and availability are still not able to satisfy application requirements in these areas [13, 12].

Overall, the current trend toward using commodity components to build larger systems results in systems that are more prone to faults. Although the cost of the basic equipment in such systems tends to be low and to follow modern cost curves for commodity components, the total cost of ownership is still high due to the extensive monitoring, maintenance, and recovery costs required to provide continuous system operation. Higher latencies and the lack of single system image support at the operating system level impose challenges in system monitoring and reconfiguration. Furthermore, the market-imposed requirements for heterogeneity and building open systems seem to be instrumental in increasing the number of faults. The fact that applications need to be written in custom ways for these architectures, requires large development and testing cycles for building robust systems. Moreover, recovery times usually tend to be longer in such systems due to the extensive human intervention required to discover the nature of the fault, to remedy the situation, and to restore the system in a consistent state. Finally, currently, in most clusters used for parallel computation, since failures are not independent, when even a single processor fails, the entire computation is either halted and the system reboots, or the results produced may be incorrect.

In this work we address these issues by providing a reliable, shared memory programming abstraction. We believe that providing such functionality (i.e. dealing with failures) is best done outside the operating system layer due to complexity reasons and below the application layer due to transparency and cost reasons. Today's commodity operating systems have been highly tuned for managing resources within single nodes and it has taken a large effort to reach the current levels of reliability and robustness. Adding multi-node functionality in the same layer jeopardizes these achievements. Instead, the shared memory system we use provides us with transparent access to application memory and the ability to manipulate all shared data.

More specifically, our work targets all-software distributed shared-memory systems and extends a state-of-the-art shared virtual memory (SVM) protocol to tolerate efficiently single, fail-stop node failures or multiple, successive, faults. Our approach tries to achieve reliability through dynamic data replication, thus exploiting the redundancy that is inherent in the system due to the presence of multiple processing elements. At the same time, it finely integrates the extensions for supporting fault-tolerance with the existing SVM protocol in order to incur as little overhead as possible and thus, retain high performance.

Our approach differs from previous efforts in the following three important ways: First, we employ *consistent dynamic replication* of global state to tolerate system failures. We replicate application shared data and critical protocol information in the volatile memories of multiple nodes and we maintain consistency of global state throughout the execution at synchronization points. Unlike our work, most re-

search efforts that address the problem of fault-tolerance in this context, use logging to stable storage, shared disks, or non-volatile and persistent memory and require extensive recovery actions. With our approach, the system can continue execution in the presence of failures by simple reconfiguration operations. Thus, not only can our approach tolerate system failures, but also provides the foundations for continuous system operation, eliminating recovery time.

Second, we take advantage of low-latency remote write and read operations provided by modern system area networks (SANs). Previous work [5, 7, 32] has concluded that such operations are vital for improving overall system performance in scalable servers for various applications. Our work leverages such operations to reduce overheads incurred by protocol extensions for supporting reliability. Ignoring such operations would result in introducing data copies and/or other operations that have been removed from software layers with extensive research efforts.

Third, we provide support for SMP nodes and multiple writers, which has non-trivial implications on protocol design. Our work is the first (to the best of our knowledge) to address these issues. The main problem is that the operation of all threads within each node have to appear atomic with respect to all other nodes. We achieve this by ordering appropriately all protocol operations at synchronization and data transfer points.

Our high-level conclusions are: a) The design and implementation of protocols that tolerate even simple failure scenarios is substantially more complex than their failure-free counterparts. b) Supporting SMP nodes and multiple writers has non-trivial implications on protocol design and performance. c) Our protocol imposes overheads in memory requirements and system performance. Our extensions require that the memory for shared data is roughly doubled (slightly more than this) to accommodate the double copies of all application data. On the performance side, our preliminary results show that in the common, failure-free case, the overhead varies between 20% and 100% across all applications and configurations we use.

The rest of the paper is organized as follows. In section 2 we present an overview of related research efforts. Section 3 describes our platform. Section 4 presents our protocol extensions and modifications. Section 5 provides an analysis of our experimental results and Section 6 identifies the main limitations of our work and discusses directions for future research. Section 7 summarizes the major results of this paper.

2. Related Work

Based on the taxonomy presented in [24], our scheme is a *backward error recovery* scheme that uses replication to distinct volatile memories for storage protection, performs uncoordinated checkpoints across nodes and coordinated inside each node, and achieves separation of checkpoint and working data with full duplication but by partial propagation of modifications, by means of SVM protocol diffs. In this section we summarize representative examples of recent research on fault-tolerance that pertains to our work.

The currently renewed interest in improving server reliability is manifested by the emergence of a large number of commercial cluster management systems, such as Microsoft

MSCS, NCR Lifekeeper and Veritas Firstwatch, that provide fault-tolerance functionalities. Such products differ from our approach in that they employ transactional semantics (for computation and I/O) and failover to backup servers to provide continuous service operation at the application level. Our approach is orthogonal and aims to improve the availability of a single, software shared memory server, by means of checkpointing and rollback recovery.

Similarly to our work, the authors in [25, 24, 2] address fault tolerance in the context of distributed shared memory (DSM) machines but their approach requires somewhat extensive hardware support. Software-based approaches to building fault tolerant systems of commodity, off-the-self components include active replication [3], development of fault-tolerant management software libraries [14, 29] and application-transparent techniques, such as our own. Due to space limitations, we shall next focus on the latter category.

Sultan et al. [27] 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 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. On the other hand, in our scheme, the memory consistency guarantees make possible the recovery of a failed process *without* protocol data logging or stable storage support. Essentially our design replaces logs with independent short checkpoints at each release.

In [9], Costa et al. have extended Treadmarks [19], a lazy release consistent, 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.

Zhou et al. [33] 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 case of failure. Although this work targets application domains different than ours, the

experimental results suggest that volatile logging using the VMMC model can be used on clusters to achieve reliability efficiently, as opposed to traditional techniques based on stable storage support.

Plank et al. [23] have demonstrated that the elimination of stable storage support offers significant improvements in checkpointing latency and recovery time. Their technique, called *diskless checkpointing*, avoids disk logging by using memory and processor redundancy and a distributed parity protection mechanism to tolerate single processor failures. In contrast to our work, this scheme uses dedicated backup processors and is based on coordinated checkpointing.

Kermarrec et al. [20] have extended a standard sequential consistency protocol used by a DSM system with a recovery scheme that avoids stable storage support by maintaining for each shared page recovery (checkpoint) data and by replicating recovery pages on two distinct node memories. Upon taking a new checkpoint, atomic updates of recovery data are performed using a *globally coordinated*, two-phase commit protocol. Their performance results have demonstrated that replication in a DSM system is a promising approach for providing reliability in an efficient and scalable way.

The authors in [1, 21, 26] investigate 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 the 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 [22].

3. Base system

The system architecture we use for our design and implementation is a SVM cluster of Intel–based, dual–processor systems, interconnected with a Myrinet SAN [6]. This section describes in more detail the communication layer, and the base SVM protocol.

3.1. Communication layer

Myrinet is a high–speed system area network with low bit error rates, very low latency for small messages (less than 10 μs) and high bandwidth in the order of 100's of MBytes/s, limited by PCI bus implementations. Cross-node SVM communication is based on a user–level communication library, namely Virtual Memory Mapped Communication (VMMC) [10]. The most important feature of VMMC is the remote deposit and remote fetch operations, which allow for data that is explicitly transferred between two nodes via a send or receive message, to be deposited in specified virtual addresses in the destination host's main memory without interrupting the remote host processor. VMMC also tolerates transient network errors by using packet retransmission, and guarantees FIFO message delivery.

3.2. Original SVM protocol

The original SVM protocol, GeNIMA [5] is based on home–based lazy release consistency (HLRC) [34] 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 [18], 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 each 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 [34], to which writers send their page updates eagerly, upon a release. Nodes propagate page updates in the form of *diffs*, which consist of the modifications applied to the version of the page before its first write (also called the *twin*). Diffs address the problem of false sharing as they allow multiple writers to modify different parts of the same page without intervening synchronization.

Lock synchronization in GeNIMA, as in many SVM 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 latest requester and updates the tail accordingly. Exchange of locks *within* an SMP requires no external or internal message exchange.

In SVM systems, processors cycle through *acquire–compute–release* cycles (Fig. 1). When a thread performs a lock release, it ends the current time interval by *committing* all pages updated by any local thread during the past interval in a protocol data structure, indexed by the interval number. After committing all local updates, the releasing process releases the corresponding lock to the next requesting node and then, 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.
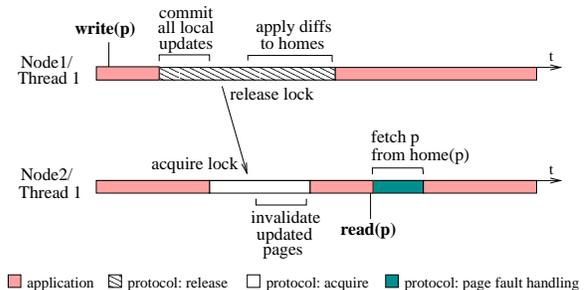

Figure 1. Basic SVM protocol operations.

During an acquire, 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. Pages are fetched in their entirety.

4. Protocol Design

We regard fault tolerance as the twofold problem of a) maintaining shared memory consistency at synchronization points in the presence of failures, and b) recovering the failed processes so that the computation continues and completes successfully. Replicating protocol state (usually small tables) is a simpler problem and we comment on it only briefly.

The key idea behind our design is to guarantee that, at all events, shared memory remains consistent at synchronization points (both locks and barriers). In the context of our shared memory protocol this is equivalent to guaranteeing that in the event of failure of a node F, no shared memory write executed by F since its last successful synchronization point will be performed at any node other than F. As a result, any failure between synchronization points will result in restarting the threads of the failed node from the last synchronization point in 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.

To achieve memory consistency at synchronization points, our extended protocol dynamically replicates all global application state and local thread state in the distributed memory of the cluster. Global application state is replicated at the points of update propagation in the SVM protocol, that is, at lock releases and at barriers. Local thread state is also replicated using uncoordinated (across nodes) checkpointing. In case of failure of node F, a process running on F is able to recover as follows.

As soon as a node failure is detected by an application thread, a global synchronization point is performed among all application threads to exclude the failed node and to perform the recovery actions. These actions include reconfiguration of specific protocol data structures, restoration of shared memory consistency (if the failure occurred during a synchronization operation), and recovery of the failed threads on the backup node where their state has been saved. After that, execution can continue immediately.

In summary, our system provides the following consistency guarantees. In a failure-free execution, the extended SVM protocol guarantees memory consistency at synchronization points. In 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 synchronization point of F, have been performed at the corresponding home nodes in the system. 3. No shared write executed by a thread in F *after* its last synchronization point has been performed at a node other than F.

4.1. Failure Detection

We assume that nodes are subject to *fail-stop* failures. In this work 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 network error failures are resolved by VMMC as mentioned above.

We exploit the semantics of the underlying communication layer as explained later to provide a reliable failure detection mechanism. First, we assume that basic communication operations that exchange data return an error when the destination node is unreachable, that the communication layer deals with transient and permanent *network* failures [28], and that the network cannot be partitioned. Thus, any error returned from communication operations signifies a remote node failure. These are all realistic assumptions for most real-life setups. Second, when nodes do not communicate and need to wait for a remote response, they send heart–beats 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.

When a failure is detected, that is when a communication operation to a remote node returns an error, the communication layer guarantees that any subsequent communication with this node will not complete and will return an error. For previous operations to this node there is no guarantee of success, unless a response has been received by the remote host.

4.2. Home page replication

In order to guarantee shared memory consistency in the event of single node failures, we employ duplication of the application shared data in the distributed volatile memory of the cluster. Each shared page is replicated at two distinct nodes and if one copy of the page becomes inaccessible or corrupted due to a failure, then the other one is used in three ways: first, to allow the computation to continue by providing alternative access to critical data residing at the failed node; second, if needed, to create a new replica of the page and restore consistency among the two replicas so that any subsequent failure can be tolerated in a similar fashion; and third, to support the execution replay of the failed processes.

In this respect, we extend GeNIMA and assign to each shared page two homes, namely the *primary* and the *secondary* home. Similarly, a page is called a primary (*secondary*) home page of node N, if N is the primary (*secondary*) home of that page. The assignment of primary homes to pages is performed by the application in a way that maximizes parallelism and optimizes performance under the HLRC memory model. The secondary homes of our scheme are initially set to be the nodes immediately following the corresponding primary homes in node order.

In a HLRC protocol, the role of home pages in maintaining memory consistency is crucial. The home copy of a page contains all of the modifications that have been performed to the page, up to the last release and therefore, it is the one fetched upon an acquire. Similarly, in our case, both home copies of a page contain the latest version of the page and are kept consistent. During a failure-free execution, it is always the primary copies that are fetched while the secondary copies serve to store tentative modifications and are used as backup in case of failure.

Each home node maintains for every primary home page p an additional page, called the *committed* copy, and for every secondary page, an additional page called the *tentative* copy. During application execution, *local* modifications of p

are performed in the original copy of the page, *p*, also called the *working* copy of p. As part of the SVM protocol, during a release operation, any *remote* modifications of p performed by the releasing node are propagated to the tentative and committed copies. This is described in more detail next.

Update propagation. Page updates are propagated in the form of diffs. Home page replication implies that upon a release, page diffs must be propagated to both home nodes. Thus, in contrast to the original SVM protocol where home nodes do not send diffs for their own pages, twins are now created and diffs for a modified page must be computed and propagated even for the home pages.

To tolerate single failures and guarantee shared memory release consistency during diff propagation, we employ the following *two-phase* diff propagation scheme (Fig. 2). In the first phase, the releasing node computes the diffs for each and every page updated locally, independent of whether this is a home or non-home page. For each page, the diffs are applied remotely to the tentative copy of the page at its secondary home. In the second phase, the same page diffs are applied to the committed copies of the pages at their primary homes.
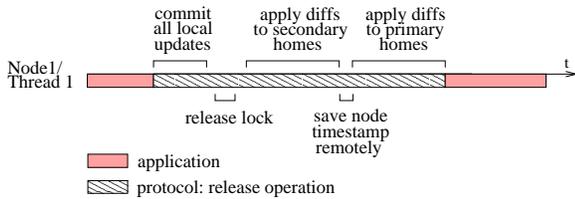

Figure 2. Diff propagation in the extended protocol.

The order of propagating updates to the two homes is important in that it guarantees that home updates are serialized. Given that nodes always fetch pages from their primary homes which are updated last, a page is fetched after both home copies have been updated in the same serial order.

The two-phase scheme also guarantees that during a release operation updates are propagated atomically. That is, in the event of a node failure during a release, either all updates performed by the releasing node will be propagated or none. In particular, if a failure occurs during the first phase of diff propagation, the secondary home pages that have been modified during the release, can be restored to their state before the release, using the primary copies. If a failure occurs during the second phase of diff propagation, then the execution can roll forward since the primary copies of the modified pages can be updated using the secondary copies and the release will complete successfully.

Based on this idea, in case of failure and during recovery, shared memory consistency can be restored by copying one home copy to the other. Which copy among the two is the valid one is determined with the help of the timestamp of the failed node which was saved remotely by the failed node at the end of the first phase of diff propagation upon its last release (Fig. 2).

The atomicity of update propagation is also preserved by using the additional (*tentative* and *committed*) pages to aggregate global page modifications at the home nodes, instead of using the working copies of the pages as is the case in the base SVM protocol. Figure 3 illustrates a scenario in which

the use of one single home page receiving both local and remote modifications could lead to violation of atomicity.
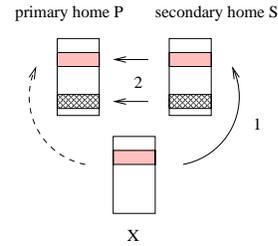

Figure 3. Atomicity violation during update propagation.

Given that our SVM protocol is multi-writer, the same page p can be modified concurrently by some node X and the secondary home S of the page (false sharing). When node X performs a release, it will propagate its diffs to S first and then to the primary home of the page, P. Similarly, when S performs a release, it will propagate its own diffs to P. However, when computing its diffs by comparing the latest version of the page with its twin, S has no way to distinguish between the updates that have been performed locally and the updates that have been propagated by some remote node. As a result, if X's updates have been propagated to S before S's release, then these updates will be propagated to P by S along with any other local updates when S performs its own release. Thus, if X fails *before completing* the first phase of diff propagation, its updates will be partially committed and atomicity will be violated.

We bypass this problem with the use of the tentative and committed home copies. Now, all remote updates are applied to these copies, while local updates are applied to the local working copies of the pages. Thus, when a home node computes the diffs for a home page, it only computes and propagates the diffs that are due to its own modifications.

The new home page layout results in two further protocol changes related to the way home pages are fetched. First, the nodes must now fetch the committed page copies and not the working copies from the pages' primary homes. The committed page copies consist the latest version of the pages that contain all remote and local modifications that are permanent and remain unaffected in case of failure. Second, the primary homes no longer view directly the remote modifications performed at their home pages. Upon a page fault, they now have to fetch the version needed from the local, committed copy of the home page. The secondary homes fetch pages from their primary homes, as in the original protocol.

SMP-specific extensions. Supporting SMP nodes induces additional modifications in the release operation of the extended SVM protocol. During diff propagation, GeNIMA permits the eager propagation of page updates that do not belong in the view of the currently releasing thread. Such updates are performed to pages that happen to be commonly modified by the releasing thread and other threads on the same SMP. Fig. 4 depicts an example of this scenario. Threads 1 and 2 on the same SMP node modify page p (false sharing). Thread 1 performs a release, commits the updated pages of the last interval, including p, and propagates the diffs of all updated pages to their home nodes. For page p in

particular, these diffs include the modifications performed by thread 2. Although in a failure-free execution this eager diff propagation does not violate correctness, in case of failure it may lead to violation of the atomicity of diff propagation and result in incorrect execution replay after recovery. Specifically, if a failure occurs after thread 1 has completed its first phase of diff propagation, but before thread 2 has performed its own release (fig. 4), then page p will contain updates that will have to be replayed by thread 2 during its rollback.
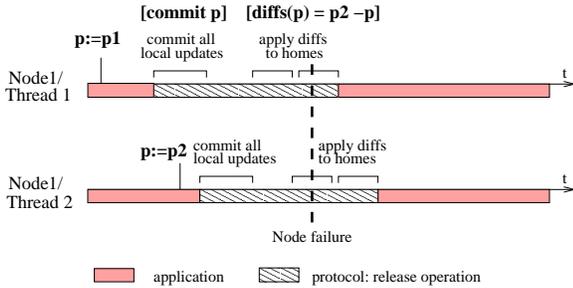

Figure 4. Atomicity violation due to eager diff propagation.

In the extended SVM protocol we prevent this scenario by: a) *locking* the updated pages when they are committed and b) by *blocking* any page fault handling for locked pages until they are unlocked. Pages are unlocked after the diff propagation is complete. Page locking in combination with blocking the page fault handling for the locked pages guarantee that new writes to pages that belong to the intervals committed by an outstanding release will be stalled and thus, will not be recorded in the new list of updates until that release operation has completed. With page locking, different lock releases can still be performed concurrently provided that the set of pages diffed by distinct threads are disjoint.

4.3. Lock synchronization

An important consideration when nodes are subject to failures is how, in case of failure, the locks being held or managed by the failed node are managed after the failure. We have considered two alternative ways of modifying the lock synchronization scheme of GeNIMA in order to tolerate single–node failures.

The first option that we designed, implemented, and evaluated is based on a primary-secondary 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. A queue-based lock algorithm like this has many 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 multithreaded environments, and it incurs low storage overhead.

Our experience with developing the above scheme has shown that its benefits are negated by its main disadvantage: its excessive complexity to implement in the failure-free case and the complexity of recovery actions required in case of failure. We have identified two main reasons for this: first, the requirement to maintain state information (e.g. the id's of

last and second to last node in the queue of requesters) and second, the FIFO processing of lock requests on the primary home. In the extended version of the algorithm, these requirements increase complexity significantly because of the need for consistently serializing all lock-related events on the two homes. The multithreaded and asynchronous nature of the system introduces further synchronization problems that in the presence of failures in particular, are hard to resolve.

To simplify the operations needed to resume lock synchronization after a failure, we replace the basic lock synchronization scheme of GeNIMA with an alternative locking algorithm that replaces interrupts with protocol-level polling and does not require any synchronization support from the network interface. Each lock is represented as a vector with one element for each node in the system and assigned to a *home* node. When a node needs to acquire a lock, it performs a remote write of a non-zero value to its element in the lock vector and then reads the whole vector. If only its element is non-zero then it has acquired the lock, otherwise it resets its element in the lock vector with a remote write operation and retries. Similarly to the old algorithm, the exchange of locks *within* an SMP requires no external or internal message exchange and is equivalent to a few assembly instructions. The new scheme is also extended with the use of secondary homes for the locks in order to tolerate single failures.

The main advantage of the new centralized lock algorithm is that by being stateless, it greatly simplifies the recovery actions related to locks. Our evaluation indicates that with the new locking scheme, lock contention is increased but not prohibitive, and that livelock can be avoided with the use of appropriate backoff times. Our results also demonstrate that the centralized algorithm performs at least as well as the distributed queuing lock algorithm. All things considered, in favor of simplicity over network traffic reduction, the centralized algorithm is our scheme of choice.

4.4. Support for Thread Migration

Our approach in resuming the execution of the failed threads on a backup node while at the same time preserving correctness, is based on the observation that between the completion of the failed node's last release and the point of failure, none of its local updates has been performed remotely. This implies that the execution of the failed threads can be safely resumed at the point of their node's last successful release.

To enable this type of migration, threads checkpoint their local execution state (thread context and stack) at designated points in the SVM protocol execution and in particular, during each release operation. To illustrate this better, let us consider thread $T1$ in Fig. 5 which is performing a release. Clearly, $T1$ must take a checkpoint of its own state as soon as it completes the first phase of diff propagation (point *B*). This is the point where, conceptually, the release operation completes since all local updates have been propagated to the first set of homes. Should a failure occur at any time between this point *B* and the corresponding point *B'* of the succeeding release, the releasing thread $T1$ can safely resume from the most recently saved checkpoint of its state. All updates that had been performed and propagated remotely after that point will be cancelled during recovery and the process of restoring

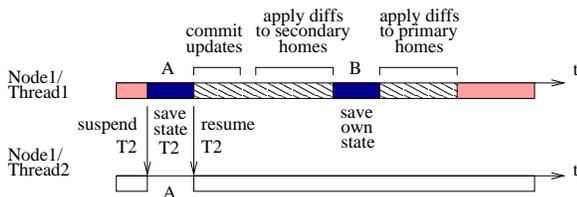shared memory consistency, as described in section 4.5.2.

Figure 5. Thread state checkpointing.

The checkpointing scheme as described above applies directly in the case of a single-threaded environment. The fact that the nodes are SMPs and that releases are performed at the node and not at the thread level, introduces additional overhead. Specifically, suppose that the node of the example above is a 2-way SMP and there is a second thread $T2$ executing on the same node. It is now important to note that before $T1$ enters the diff propagation stage, it commits *all* local updates including the ones performed so far by $T2$. Because the updates of all threads within a node have to be performed atomically with respect to all other nodes, the state of $T2$ must be saved upon termination of the current interval by $T1$ (signified by the commitment of updates), that is, at point A. Similarly, in the case of more than two threads per node, at point A, the state of all threads other than the releasing one must be checkpointed. At both points A and B, the thread state is saved remotely at a designated backup node, where the failed threads will be resumed in case of failure.

One clear implication of this type of checkpointing is that the executions of all local threads become interdependent and some type of internal synchronization is required to ensure correctness during checkpointing. In our approach, at point A of each release, the releasing thread suspends the rest of the local threads, saves their state remotely, and then resumes the suspended threads and continues with the release operations.

Another implication due to SMP nodes is that checkpointing performed by different threads cannot be overlapping. Unavoidably, this imposes the constraint that simultaneous lock releases inside each node must now be serialized. Alternative base protocol designs could alleviate such implications. For instance, using processes in each node as opposed to threads that use the same operating system page table, would eliminate this type of intra-node synchronization. However, such protocols have other implications and the tradeoffs are not clear.

A final issue related to checkpointing is the memory overhead that it introduces. In our case, checkpoints are completely independent across nodes and have minimal memory requirements (the stack is usually a few KBytes). Thus, they can be saved to the volatile memory of remote nodes or even to persistent memory. Somewhat more important, but still, not particularly significant, is the memory overhead for reserving thread stack space in each node. To deal with the typical migration problem of stack data pointer resolution [15, 30], during system initialization, on each and every node, we create as many threads as the overall number of compute threads in the system. For the additional (shadow) threads we reserve thread stack space in a way that the thread stacks of each compute and its corresponding shadow thread share exactly the same virtual address space.

4.5. Recovery actions

When a node detects a node failure, it broadcasts a failure notification message which triggers a **global** recovery phase. This consists of the following operations:

4.5.1. Reconfiguration of locks and pages

The failure of a node F has a direct impact on system execution because of the role of F as primary/secondary home for a number of pages and locks. Not only are the live nodes unable to fetch any of F's primary home pages, but they are also unable to commit their updates to F and continue with their execution. Similarly, the locks whose primary or secondary home was F cannot be acquired, while the locks owned by F at the point of failure, are not released.

Thus, the first step taken when a failure of a node F is detected, is to designate a new home for all pages and for all locks for which F was either a primary or a secondary home. The assignment of new homes is straightforward and in both cases, employs a simple mapping scheme that guarantees that under any failure scenario, the two replicas of each shared page and each lock data structure will always be resident at distinct nodes. Henceforth, the affected home pages will be fetched from their new primary home and their tentative updates will be sent to the new secondary home. Lock synchronization can also resume directly by using the two new lock homes.

4.5.2. Restoring shared memory consistency

In order to determine the recovery actions required to restore memory consistency, we distinguish two cases of a node failure. The first case of failure occurs while the node executes application code or protocol operations, *except for a release*. In this case, the node has not yet propagated any local updates performed after its last release. As a result, rolling back and replaying its execution from the point of last release is straightforward: all shared data accessed from that point until the point of failure is accessible at some remote node (since data is replicated twice) and still valid (since no modifications have been propagated). The second and most interesting case of failure, happens while the failed node is performing a release and has two important implications. First, the affected home page replicas must be brought to a consistent state so that the system can tolerate any subsequent failures. Second, it should be guaranteed that after the data recovery actions are complete and the failed processes resume, their execution satisfies the memory consistency model requirements and produces correct results.

The two replicas of a page might be left inconsistent due to a failure that occurred while the failed node F was propagating diffs for that page. Clearly, the two replicas become consistent again by copying one home copy to the other. The valid replica is identified with the help of the timestamp of F which had been saved remotely by F at the end of the first phase of diff propagation upon its last release. In essence, this timestamp designates the set of updates that have been propagated to at least their secondary home node as part of a release that is considered complete. Hence, if a page belongs to this set, then it is the tentative copy of the page that is copied to the committed copy. This is equivalent to

rolling the failed node's execution forward up to the completion of its interrupted release. Otherwise, the committed copy is copied to the tentative copy, thus cancelling any updates sent partially by F to the secondary home nodes. This is equivalent to rolling the failed node's execution backward to the point of its last successful release.

To ensure correctness during the page reconstruction described above, it must be guaranteed that there are no outstanding updates being propagated by any node in the system other than the failed one. This precondition is imposed by the possibility of false sharing and guards against the risk of cancelling or prematurely committing updates performed concurrently by other nodes. For this reason, we stipulate that the recovery phase is initiated only after all nodes have synchronized and that when a node reaches this global barrier, there are no pending releases performed at this node. This requirement implies that during recovery, the two home copies of any page will be identical, except perhaps of the updates applied incompletely by the failed node.

4.5.3. Recovery of failed threads

We now discuss how, in case of failure, the failed threads are resumed by using their remotely saved state. Again, we distinguish two cases of a node failure. The first case of failure occurs while the node executes application code or protocol operations, *except for a release*. More specifically, consider thread $T1$ in Fig. 6. If a failure occurs *between* the two successive releases, thread $T1$ that released last will resume from point B and will return immediately to the application (since the corresponding release is complete). For all other threads, execution will resume from point A, that is the state that was captured by $T1$ and includes all –and only those– updates that have been performed remotely.
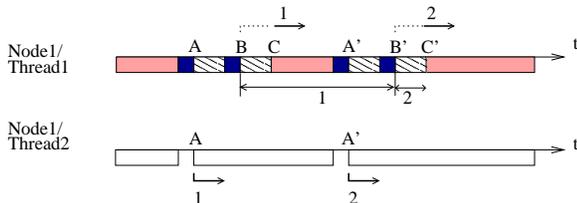

Figure 6. Recovery of failed threads.

The second case of failure, happens while the failed node is performing a release and more specifically, while it is executing in the diff propagation stage or while checkpointing. In particular, if the failure occurs during the first phase of diff propagation or at any time during checkpointing, the execution is resumed based on the state of the threads saved during the *previous* release, similarly to the aforementioned case. Fig. 6 depicts the points from which the two threads are resumed when a failure occurs at any point within the time interval 1. If the failure occurs during the second phase of diff propagation (i.e. between points B' and the end of release C' in Fig. 6), the execution is resumed based on the state of the threads saved during the current release. Fig. 6 depicts the points from which the two threads are resumed when a failure occurs at any point within the time interval 2. Thus, in any case, the execution is resumed from the point of last successful release at the failed node.

From the previous analysis, it becomes clear that the starting point of execution replay varies according to the point of failure and to tolerate failures even during checkpointing, two copies of each thread state need be maintained on the backup node: one copy being updated during the current release, and the other containing the state saved during the last successful release. The two copies are updated alternately, ensuring that a consistent copy of the state that was saved last is always available in case a failure occurs while checkpointing is still in progress.

5. Performance evaluation & Discussion

5.1. Experimental Testbed

The system we use in our evaluation is a shared virtual memory cluster of eight 400MHz, 2-way Pentium-II SMP nodes running Windows NT and interconnected with a low–latency, high–bandwidth Myrinet SAN [6]. All eight nodes are connected directly to an 8-way switch. The communication library that we use on top of the Myrinet network is VMMC. In our cluster, VMMC provides a one way, end–to–end latency of around $8\mu s$, which is among the best performing systems using a Myrinet interconnect.

For our performance evaluation, we use the SPLASH-2 [31, 17] application suite. The specific applications and problem sizes that we use are: FFT (1M points), LU-contiguous (1024×1024 matrix), WaterNsquared (4096 molecules), WaterSpatialFL (4096 molecules), RadixLocal (4M keys), and Volrend (head).

5.2. Discussion of execution time breakdown

To better evaluate our performance results, we break down application execution time into the following six components: compute time, data wait time, synchronization, diffs, checkpointing and protocol processing.

Compute time is the time spent for application execution. This includes stall time to local memory accesses. During our experimentation, we observe that with our extended protocol and particular applications, the compute time increases compared to the compute time observed when running GeN-IMA, especially as the problem size and the number of processors per node increase. We attribute this problem to the increasing number of data transfers in the extended protocol that causes higher contention on the SMP memory among the processors within each SMP node.

Data wait time refers to the time spent in handling page faults. When a remote page is needed, the local processor fetches the committed copy of the page from its primary home node, using a remote fetch operation. In the extended protocol, there are four extensions that have an impact on the page fault time. First, the propagation of updates to two home nodes in combination with the requirement that the committed copy of the page be updated last, increase the mean time to update the pages at their primary homes. Second, the page fault handling is stalled if the faulting page is *locked* until it is unlocked (section 4.2). Third, in contrast to the original protocol where home nodes do not create twins for their own pages, twins must now be created even for home pages. Finally, if the faulting page is a home page,

then the local processor must perform a *local* fetch of the required version of the page by copying the committed copy of the page to the working copy. Therefore the average data wait time in the extended protocol is expected to increase.

Synchronization time consists of the wait time at synchronization points and more specifically of a) the inter-node and intra-node synchronization time at the barriers and b) the wait time at lock acquires. We focus on the second factor which is directly affected by our protocol extensions.

Lock wait time is the time between the issue of a lock request and the actual acquire of that lock. Lock wait time is clearly affected by the lock synchronization algorithm used. However simple, the polling-based algorithm used in the extended protocol increases the load at the network interface on each node and the network traffic as compared to the original protocol where the critical path to acquire a lock is only 2 hops. The actual impact of the new lock synchronization scheme on wait time depends on the number of locks used, the number of processors in the system, and the lock contention induced by the application.

Lock wait time in the extended protocol is further increased due to replication. Lock-related data structures (namely, the buffers used for polling and the lock timestamps) are now replicated to two homes and all lock synchronization data must be updated consistently to both homes upon each lock acquire and each lock release. For fairness in our performance comparison between GeNIMA and our extended protocol, we employ the same lock algorithm based on polling in both protocols. However, the former uses the basic version of the algorithm, while the latter uses the fault-tolerant version, that supports duplication of lock homes and replication of lock-related protocol data.

Finally, the lock wait time is indirectly affected by the serialization of releases. Unlike GeNIMA that enables multiple releases to be performed in parallel by different threads on the same SMP node, our initial design of the extended protocol does not support parallel releases, thus limiting concurrency and introducing delays in the exchange of locks.

Diffs have a significant impact on the performance of the extended SVM protocol. First, the replication of shared data updates and protocol information to two distinct homes (instead of one) results, in the general case, in a twofold increase of messages sent during the diff propagation stage. Second, the fact that diffs must now be computed and propagated for home pages introduces computational and communication overhead that was not present in the original protocol. This suggests that the extended protocol's overhead due to diff operations is expected to be pronounced in the case of applications like FFT, where each processor processes data that belong, exclusively or primarily, to its home pages.

This increase in the amount of data exchanged during diff propagation is critical for the system performance. On each node, as the number of messages posted in the queue between the processors grows, the network interface becomes more loaded. If the number of send messages grows beyond the size of the post queue then the network interface becomes full, and, although messages are sent asynchronously, the sending processor is blocked waiting for the queue to be drained before new requests can be posted. Noticeably, our SVM protocol favors the occurrence of this scenario because the diff computation is performed eagerly at release points

and hence diff messages are clustered at releases.

Another implication of the new, two-phase diff propagation scheme is that diffs must be saved locally. In GeNIMA, when a processor computes a diff, it sends the diff directly to the home, using a remote deposit operation. In contrast, in the extended protocol, the computed diffs must be stored in a local data structure so that they need not be recomputed in the second phase of diff propagation. Hence, while we still use the asynchronous send mechanism with remote deposit to update the shared application data pages at the home nodes, our scheme requires some local processing of the diffs.

Checkpointing introduces a new component in the execution. In our system, a thread's context and stack are sufficient to migrate that thread between nodes. There are no open files or other resources allocated from the operating system that need to be handled, and all global data resides in shared memory. The impact of checkpointing varies across applications and mainly depends on the following parameters: the number of checkpoints taken that is proportional to the total number of releases, the size of the thread stack, and the number of threads executing concurrently on each SMP node.

Protocol processing refers to the aggregate execution time excluding all of the components explained above. The two dominating factors of protocol processing time are the cost for page invalidations and communication. Protocol processing time can be reduced by applying protocol level optimizations or by using faster communication primitives. Protocol level optimizations have been the focus of past research efforts [11]. Our actual communication costs are relatively low and are not an important bottleneck in our work.

5.3. Preliminary Results

To evaluate our system we run our application suite using both the original and the extended version of the SVM protocol. The latter contains all of our protocol modifications and extensions. We also use two different configurations: 8 nodes with 1 compute thread per node and 8 nodes with 2 compute threads per node.

In order to better analyze our performance results we break down the application execution in two different formats. The first divides the execution time into four components (shown from bottom to top in Figure 7): *compute time*, *data wait time*, *lock time*, and *barrier time*. The second format divides the execution time into six components (shown from bottom to top in Figure 8): *compute time*, *data wait time*, *synchronization time*, *diffs*, *protocol processing* and *checkpointing*. For each application we show two bars: the program execution using the original SVM protocol (0) and the program execution using the extended SVM protocol (1). All times are given in milliseconds.

5.3.1. Uniprocessor nodes

Figure 7 shows the execution times of our application suite on 8 nodes with 1 compute thread per node. We observe that the overall execution overhead ranges across applications between 20% (Radix) and 67% (WaterSpFL).

Let us first examine the effect of computing the diffs of home pages on performance. As already mentioned, while originally the diff processing of home pages was completely absent, in the extended protocol, not only are diffs sent for
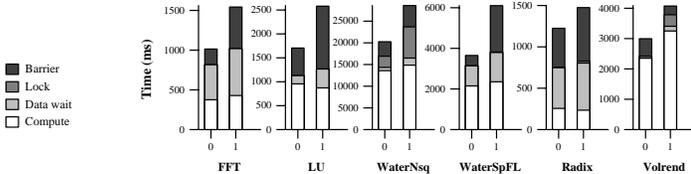
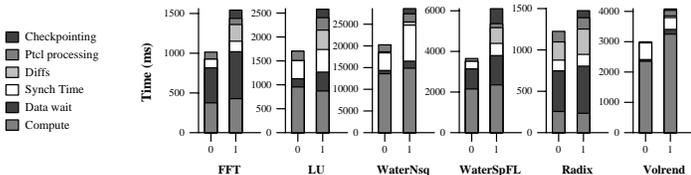**Figure 7. Execution breakdown: 8 nodes, 1 thread/node.**

Figure 8. Overhead breakdown: 8 nodes, 1 thread/node.

home pages but also all diffs are propagated twice. For FFT and LU, diff processing is the factor that contributes the most to the observed overhead. This is an expected outcome since data sets in FFT and LU are partitioned so that data are only updated by their home nodes. Specifically, in the extended case, the diffing of home pages introduces 20% and 24% overhead respectively, which constitutes approximately half of the total execution overhead in both applications. The diff processing overhead is also significant in the case of WaterSpatialFL where the pages diffed throughout the execution are dominated by home pages (more than 99% out of the total number of pages diffed). Hence, the associated overhead is again 20%. In the case of WaterNsquared around one fourth of the pages diffed are home pages and hence the overhead is less pronounced. In RadixLocal, the numbers of pages diffed in the original and extended case are comparable, with the percentage of pages diffed in the latter case being only around 12% of the total number of pages diffed. As a result, the contribution of the additional diff processing to the overall overhead is the least significant among all applications.

Synchronization time in the extended case also increases although, in general, to a smaller degree than diff processing time. For FFT and LU, lock synchronization is absent and hence, synchronization time is dominated by the all-to-all internode synchronization at the barriers. The synchronization overhead is especially pronounced in WaterNsquared, increasing the execution time by 20%. WaterNsquared uses 4105 locks (one lock per molecule plus 9 additional locks used for synchronization variables). Despite the large number of locks used, the lock wait time presents more than a two-fold increase which is mainly attributed to the inherent lock synchronization in WaterNsquared. Also, WaterNsquared, as opposed to WaterSpatialFL and RadixLocal, is characterized by a high frequency of releases resulting in higher communication traffic which has a negative impact on the polling-based lock acquires. WaterSpatialFL and RadixLocal use a much smaller number of locks (518 and 66 respectively). However, for these applications, the average lock wait times in the original and extended case are practically the same. As a result, the overall synchronization overhead is not significant (6% and 1% respectively) and the overall synchronization overhead is primarily due to the internode synchronization overhead.

Data wait time in the extended case presents an expected

increase due to the fact that a modified page must be updated to both of its homes before it can be fetched. Although the number of page faults when using the extended protocol changes only slightly in some cases and remains unchanged in the rest, the average wait time per page increases and the data wait time overhead across all applications ranges between 3% and 15%. The largest overhead is observed in the case of FFT and LU, for which originally, there were no home page updates propagated and no associated stalls.

Protocol processing introduces minor overhead (less than 5% across all applications). In the case of one compute thread per node we attribute this increase of protocol overhead to the new page layout that distinguishes between the working and the home copies of the pages and thus requires processing of additional coherence information.

Checkpointing, in all applications but WaterSpatialFL, constitutes less that 10% of the original execution time. The checkpointing overhead is proportional to the average size of the thread stack and to the number of checkpoints throughout the execution. For all applications the average stack size varies between 2 and 2.8 KBytes, while in all cases except for WaterNsquared the number of checkpoints is relatively small $(4 - 311)$. In WaterNsquared, the total number of checkpoints is $10,277$ and as a result, the associated overhead is more significant, introducing 20% overhead.

Finally, compute time presents slight changes among the original and extended executions of all applications. The change ranges between $-5%$ and 6% of the original execution time. Such fluctuations are acceptable, and are mainly due to changes in the memory bus contention resulting from the variations in message traffic and the associated demand for DMA transfers.

5.3.2. SMP nodes

Figure 9 depicts the execution times of our application set on 8 nodes with 2 compute threads per node. The overall execution overhead due to the SVM protocol extensions ranges between 24% (Radix) and 100% (LU, WaterSpFL). For all applications (except for volrend), the overall overhead increases relatively to the single-threaded case. The largest overhead increase compared with the single-threaded case is observed for LU 1024 and the smallest for RadixLocal 4M.

More specifically, for all applications except for WaterNsquared, barrier time is the component that is mostly affected by the use of multiple threads per node. Lock releases and in particular diff propagation consist again the major performance bottleneck, which is now more pronounced than in the single-threaded setup. The largest barrier overhead is observed for LU 1024 since the updates of all home pages are now propagated due to data replication and is as high as 86%.

In general, using more than one compute threads per node causes the number of messages sent during diff propagation (including both messages containing shared data updates and messages with protocol data) to grow significantly. In the extended protocol in particular, the increase in system traffic is even greater due to replication and most important, it is concentrated at synchronization points rather than spread over large time intervals, thus developing high contention at the network interface and the network. This effect is manifested in the high latencies observed despite the use of asynchronous communication operations and implicitly affects

the rest of the execution components. In the case of multiple compute threads per node, additional overhead is also expected due to the serialization of releases. The particularly high synchronization overhead in WaterNsquared is indicative of the effect of release serialization.
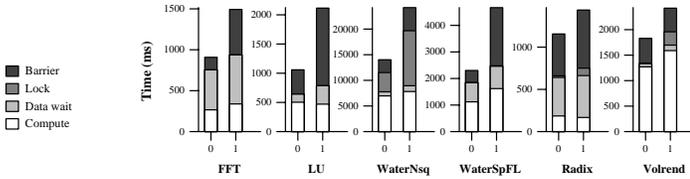

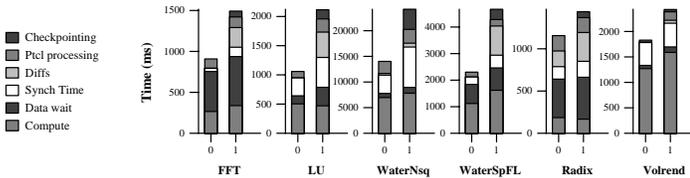**Figure 9. Execution breakdown: 8 nodes, 2 threads/node.**

Figure 10. Overhead breakdown: 8 nodes, 2 threads/ node.

The overhead in data wait time as compared to the single-threaded case decreases. Due to multiprocessing within each SMP node the average number of page faults per thread is now reduced by as much as 50% and hence, data wait time does not consist a major bottleneck despite the fact that the average data wait time per page fault is slightly increased.

Our results also show that the synchronization time overhead generally increases with multithreading. Because lock synchronization in the extended protocol is based on polling, its communication cost is particularly high under conditions of high lock contention and the lock-related messages are stalled in the already overloaded queues at the network interface. As a result, the average lock wait time is now dependent on the overall communication traffic and is expected to grow as the latter increases. Finally, the checkpointing overhead remains reasonable, being for most applications less than 15%. WaterNsquared consists an exception. The cost of checkpointing in WaterNsquared is much larger (almost 30% overhead) and is related to the excessively larger number of checkpoints taken (18,362 over 216 in the worst case for the rest of the applications).

During this preliminary evaluation we have found that certain communication layer parameters affect the system performance significantly. The most important aspect in our implementation is the fact that at synchronization points the extended protocol induces increased traffic, compared to the base protocol. To alleviate the related overheads we take advantage of asynchronous operations present in the communication layer we use. In our experiments we have found that specific NIC parameters have a critical impact on system performance. These are mainly the size of the post queue for asynchronous messages and the priorities with which the NIC handles different types of events [10]. Although our results presented here include preliminary tuning of these parameters, further investigation of these issues is necessary.

6. Limitations and Future Work

Our system at its current stage has certain limitations, which we aim at addressing in future work. We design and implement *one possible* set of SVM protocol extensions selected out of a broad configuration space. In this respect, our study sheds light on some of the underlying tradeoffs in supporting fault tolerance through data replication, identifies key optimizations and architectural enhancements that are necessary for achieving higher performance, and demonstrates the large variety of protocol design alternatives. All of these issues require further examination.

Our preliminary performance evaluation is restricted to relatively small problem sizes and a small number of applications. Experimenting with larger problem sizes that are representative of realistic situations is necessary for a meaningful evaluation of our system.

Our evaluation has suggested several potential directions for improving performance. More specifically, our protocol prototype can be enhanced with optimizations such as employing a more efficient way of propagating diffs at the barriers, decreasing contention at the network interface by sending fewer and larger messages, stabilizing our lock synchronization algorithm by fine tuning critical parameters or using NIC synchronization operations as opposed to protocol-based locks, and reducing the number of execution points where shared data is maintained consistent.

Finally, the SPLASH-2 applications form our basis for understanding the underlying tradeoffs in employing data replication for supporting fault tolerance and in certain respects, the design of our protocol extensions relies on these applications' characteristics. It is interesting to investigate how well our approach can perform in a broader application domain that includes server and other non-scientific applications.

7. Conclusions

In this work we present the design, implementation and preliminary evaluation of a shared memory protocol that uses dynamic data replication to tolerate single-node failures in modern SVM clusters. Multiple failures can also be tolerated provided they are not simultaneous and the system is able to recover between successive failures.

There are two key ideas behind our protocol design. First, we exploit the inherent redundancy of the system to dynamically replicate application shared data and coherence information in the distributed memory of the cluster. Second, we rely on the low-overhead direct remote memory operations available in modern system area networks to maintain data coherency for all copies of global and local system state at synchronization points and to reduce the protocol overhead in the common, failure-free case. Our approach also employs an efficient uncoordinated (across nodes) checkpointing scheme which, in case of failure, permits the rollback recovery of the failed node alone, leaving the remaining nodes practically unaffected, and minimizes recovery time. In addition, our scheme does not require non-volatile storage (memory or disk) and supports SMP nodes.

We implement our protocol on a state-of-the-art cluster of dual-processor, x86-based systems, interconnected with a low-latency, high-bandwidth Myrinet network. We

evaluate the performance implications of our extensions in the common, failure free case, using six applications from the SPLASH-2 application suite, on both single and multi-threaded SMP configurations. Our performance results show that the impact of our SVM protocol extensions on application performance varies between 20% and 67% across applications in the single-threaded setup and between 24% and 100% in the multi-threaded setup. Our results indicate that in most cases, the cost of replication for supporting fault tolerance is not prohibitively expensive and that our approach is viable in the context of state-of-the-art SVM clusters. However, further investigation is needed to better understand the implications of our protocol extensions.

Based on our experience with our system so far, we find that protocols based on data replication are able to transparently support continuous system operation in the presence of failures, without the need for extensive recovery overheads. Our work also indicates that building robust systems involves complications not obvious when designing or simulating protocol extensions. Overall, our results suggest that despite the underlying complexity, research in this direction is highly promising in building reliable and highly-available computing infrastructures and as such, it should be pursued further.

Acknowledgments

References

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

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

[3] C. Basile, Z. Kalbarczyk, K. Whisnant, and R. Iyer. Active replication of multithreaded applications. Technical Report CRHC-02-01, University of Illinois at Urbana-Champaign, 2002.

[4] A. Bilas, D. Jiang, and J. P. Singh. Accelerating shared virtual memory via general-purpose network interface support. *ACM Transactions on Computer Systems*, 19(1):1–35, 2001.

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

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

[7] E. V. Carrera, S. Rao, L. Iftode, and R. Bianchini. User-level communication in cluster-based servers. In *Proceedings of the 8th IEEE Int'l Symposium on High-Performance Computer Architecture (HPCA 8)*, February 2002.

[8] D. Chen, S. Dwarkadas, S. Parthasarathy, E. Pinheiro, and M. L. Scott. Interweave: A middleware system for distributed shared state. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 207–220, 2000.

[9] 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.

[10] 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.

[11] C. Gibson and A. Bilas. Shared virtual memory clusters with next-generation interconnection networks and wide compute nodes. In *8th Int'l Conference on High Performance Computing (HiPC01)*, 2001.

[12] J. Gray. What next? a dozen remaining it problems (turing lecture). In *The ACM Federated Computer Research Conference in Atlanta, Georgia*, May 1999.

[13] J. Hennessy. Back to the future: Time to return to some long standing problems in computer systems? (keynote talk). In *The ACM Federated Computer Research Conference in Atlanta, Georgia*, May 1999.

[14] Y. Huang, P. Chung, C. Kintala, C. Wang, and D. Liang. Nt-swift: Software implemented fault-tolerance on windows-nt. In *2nd USENIX WindowsNT Symposium*, pages 3–5, August 1998.

[15] A. Itzkovitz, A. Schuster, and L. Shalev. Thread migration and its applications in distributed shared memory systems. *The Journal of Systems and Software*, 42(1):71–87, 1998.

[16] P. Jamieson and A. Bilas. CableS : Thread control and memory system extensions for shared virtual memory clusters. *Lecture Notes In Computer Science*, 2104:170–181, 2001.

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

[18] 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.

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

[20] 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.

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

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

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

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

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

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

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

[28] J. Tang and A. Bilas. Tolerating network failures in system area networks. Aug. 2002.

[29] L. Technologies. Aurora management software. http://www.bell-labs.com/project/aurora, 1999.

[30] K. Thitikamol and P. Keleher. Thread migration and communication minimization in DSM systems. *The Proceedings of the IEEE*, 87:487–497, March 1999.

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

[32] Y.Zhou, A. Bilas, S. Jagannathan, C. Dubnicki, J. Philbin, and K. Li. Experiences with vi communication for database storage. In *Proceedings of the 29th Int'l Symposium on Computer Architecture*, May 2002.

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

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