Lightweight Logging and Recovery for Distributed Shared Memory over Virtual Interface Architecture

Soyeon Park, Youngjae Kim, Seung Ryoul Maeng

Department of Electrical Engineering and Computer Science
Korea Advanced Institute of Science and Technology (KAIST)
{sypark, yjkim, maeng}@camars.kaist.ac.kr

Abstract

As software Distributed Shared Memory(DSM) systems become attractive on larger clusters, the focus of attention moves toward improving the reliability of systems. In this paper, we propose a lightweight logging scheme, called remote logging, and a recovery protocol for home-based DSM. Remote logging stores coherence-related data to the volatile memory of a remote node. The logging overhead can be moderated with high-speed system area network and user-level DMA operations supported by modern communication protocols. Remote logging tolerates multiple failures if the backup nodes of failed nodes are alive. It makes the reliability of DSM grow much higher. Experimental results show that our fault-tolerant DSM has low overhead compared to conventional stable logging and it can be effectively recovered from some concurrent failures.

1. Introduction

Clusters of workstations and PCs have been adopted as cost-effective platform for parallel computing. Software Distributed Shared Memory (DSM) simplifies parallel programming tasks by providing a shared memory abstraction on clusters. DSM continues to improve its performance and scalability with a relaxed memory consistency and various optimization techniques [1,2,3,4]. Recently, high availability and reliability of DSM also become critical as DSM becomes attractive for long-running applications on larger clusters.

A common approach for fault-tolerant systems is to take a checkpoint periodically so that the system can roll back to one of the checkpoints after a failure. In DSM systems, some dependency relations are established between processes accessing shared memory. Thus, even live nodes should roll back together to keep shared memory consistent after recovery. Conventional solution to cope with the rollback propagation (i.e., *domino effect* [5]) is message logging : the messages affecting memory contents and states are logged during failure-free execution [6,7,8,9]. Only a failed node rolls back to the last checkpoint, then reproduces the same sequence of computations by applying logs.

Logging guarantees bounded rollback after a failure, but it has a significant impact on the performance of fault-tolerant DSM during failure-free execution. Especially, earlier logging schemes flush logs to disk before writes to shared data are performed to other processes [6,8,10]. Their frequent disk accesses degrade the performance of DSM. In sender-based volatile logging schemes [9,11], the message sender stores logs in its local memory and flushes them to the stable storage on a checkpoint. While the schemes have low overhead, they cannot be tolerant to the concurrent failures of multiple nodes since logs indispensable for the failed node are scattered in the others. As a new approach, a data replication scheme has been recently proposed [12]. It eliminates logging and memory checkpointing by replicating shared data on two distinct nodes. However, it requires an additional overhead for atomic updates between shared data and its replica.

In this paper, we proposed a FT-KDSM (Fault-Tolerant KAIST-DSM), which aims at lightweight logging and recovery from some concurrent failures. Most previous approaches focus on lightweight logging schemes, but under the assumption of only single node failures. As cluster size grows, it has been important to deal with concurrent failures while preserving the system performance.

In our *remote logging,* the logs for a failed node are saved to the memory of a single remote node, called a log home. Remote logging has some attractive properties as follows. First, like other volatile loggings, it does not require any disk accesses at every synchronization point. Secondly, the logging overhead due to additional message transfers is moderate on a high-speed System Area Network (SAN) and modern communication protocols. Simple logging by DMA operations, unlike the data

[1] This research is supported by KISTEP under the National Research Laboratory program.

replication scheme, does not make significant interference with the host process of a backup node. Finally, it increases the reliability of a system by tolerating some concurrent failures in multiple nodes under the situation where their log homes are alive.

With our recovery protocol, the synchronization operations are independently replayed with logs of failed processes, whereas the page fetching operations are affected by the recovery progress of other failed processes. Our work is the first to measure the overhead of recovery from multiple failures.

We implemented FT-KDSM over VIA (Virtual Interface Architecture) [13], one of the user-level communication protocols, and evaluated it on a cluster interconnected with Myrinet SAN [14]. Experimental results show that the remote logging leads to much lower failure-free overhead than traditional stable logging scheme. In ocean, remote logging imposes the logging overhead of 21% on a base DSM while stable logging degrades the performance of base DSM by 206%. In our experiments with other applications, remote logging incurs little overhead ranging from 1% to 11%. Additionally, our recovery protocol effectively recovers a system from concurrent failures in multiple nodes.

The remainder of this paper is organized as follows. Section 2 presents related work. Section 3 describes system model and memory consistency model used by FT-KDSM. Our remote logging and recovery schemes are presented in Section 4 and Section 5, respectively. The performance of FT-KDSM is discussed in Section 6. Finally, Section 7 concludes the paper.

2. Related work

An exhaustive research of rollback recovery schemes has been conducted for fault-tolerant DSMs. It is a popular approach that processes independently take a checkpoint and log the exchanged messages. A failed node can reproduce the globally consistent state of a system by replaying the messages.

The first logging scheme [10] for sequentially consistent DSMs flushes logs to disk at every page transfer. In [6], based on Lazy Release Consistency (LRC) model [3], a receiver logs the messages in memory and flushes them to disk at every synchronization point. In these stable logging schemes, failed nodes can be independently recovered from multiple failures. However, they suffer from the long latency for disk accesses on the critical path. Coherence-Centric Logging (CCL) [8] has proposed for Home-based LRC (HLRC) model [4] which has been widely used due to its scalability. To hide disk latency for log flushing, CCL overlaps disk accesses with coherence-induced communications already present in HLRC. However, a process should still confirm that logs are flushed before it continues the next computations.

Costa et al. have extended Treadmarks [15], LRC-based DSM, to implement volatile logging and two-level recovery schemes [16]. The data dependencies are logged in the volatile memory of sender and receiver nodes, thus it tolerates only single node failures. For dealing with multiple failures, consistent checkpointing is required during garbage collection operations of LRC. Sultan et al. also used volatile logging with independent checkpointing for single node failures on HLRC-based DSMs [9]. They focused on log trimming and checkpoint garbage collection rather than supporting the high level of reliability.

Recently, a data replication scheme has been proposed in [12]. As a new approach, each shared page has a primary and a secondary copy at two distinct nodes. The copies are atomically updated by a two-phase commit protocol. It does not require shared memory checkpointing. A failed node can get the consistent backup pages from the remote memory. It is similar with our remote logging in that the information for recovering a failed node is centralized in a specific remote memory. However, the data replication scheme requires some complicated protocols for maintaining consistency between two replicas. Moreover, it does not deal with concurrent failures.

3. Overview of FT-KDSM

In this section, we describe our FT-KDSM which is fault-tolerant home-based DSM. We briefly present the system model including failure models and system configuration, and memory consistency protocol.

3.1. System model

We assume fail-stop model, that is, nodes fail only by stopping and do not exhibit any incorrect behavior. FT-KDSM deals with only transient failures. A failed node can be recovered from a failure and then resume normal computations with other nodes. We consider tradeoffs between the levels of reliability and overhead for it, and then, suggest a lightweight remote logging and a new recovery scheme for tolerating some concurrent failures in multiple nodes. A checkpointing policy is beyond the scope of this paper.

FT-KDSM was implemented on a PC cluster of eight nodes running Linux and interconnected with high-performance Myrinet SAN. It uses VIA [13], which is the industry standard of user-level communication. It provides user-level DMA operations which directly transfer data to/from the virtual address space of a remote node. It does not interfere with the host-processes of the receiver. It makes our remote logging more efficient. VIA also supports packet retransmission mechanism and FIFO message delivery, thus transient network errors can be covered on the reliable communication channel. Since the communication operations return an error when a receiver is unreachable, live nodes can detect the failure of a remote

node. In this paper, we do not deal with permanent network failures.

3.2. Home-based Lazy Release Consistency

Among several relaxed consistency models, HLRC is a popular model for efficient DSMs. It is a page-based multiple-writer protocol. Program executions are divided with intervals by synchronization operations (i.e., locks and barriers) and shared memory become consistent only at the end of an interval. Happened-before ordering [17] between intervals across nodes is maintained by *vector timestamp* in each processor. It keeps track of the recent intervals of all processes.

In HLRC, every shared page has its designated home node which keeps an up-to-date page. Writes to a page by non-home nodes are propagated to its home by *diffs,* a summary of modifications. For detecting the modified parts in a page, a process makes a copy of a page, called a *twin*, before writing it. At the end of an interval, the writer makes diffs by comparing the modified page to its twin and sends them to the page home.

The cached pages are invalidated at a lock acquire and a barrier. Each process records the list of dirty pages, called *write-notices*, at the end of an interval. A process acquiring a lock receives write-notices from the previous lock releaser and then invalidates the corresponding pages. At barrier time, the write-notices of all processes are also exchanged with each other. When a page fault occurs by an access to an invalid page, a process fetches an up-to-date page from its page home.

We chose the HLRC as a base model of our FT-KDSM because it has some advantages in comparison with other memory consistency models. Network overhead due to coherence-related communications is lower in HLRC than home-less protocols. It takes only one round-trip message to fetch an up-to-date copy of a shared page from home node. Especially, HLRC can be efficiently implemented with modern communication protocols supporting user-level DMA operations [1,2].

4. Remote logging

4.1. Overview

FT-KDSM considers remote logging with independent checkpointing. In our remote logging, each process logs the write-notices and diffs received from other nodes in volatile memory of a specific remote node, called a *log home*. The messages may modify protocol states or shared data, thus they should be logged for a correct replay after a failure. Each node is used for both application processing and the log repository for another node. In remote logging, if a process concurrently fails with its log home, the system cannot be recovered. Otherwise, remote logging can tolerate concurrent failures in multiple nodes.

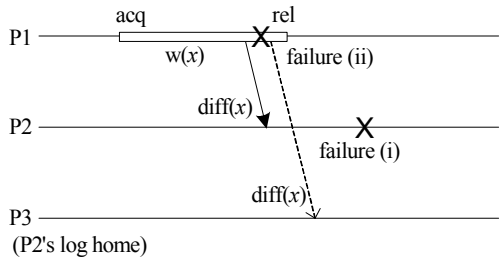

Figure 1. Diff logging

Remote logging is lightweight and efficient by taking advantages of high-performance SAN and modern user-level communication protocols. The user-level DMA operation allows the logs to be deposited in specified virtual addresses in the memory of a log home with low cost. It removes kernel intervention from the critical communication path and supports zero-copy data transfer. Especially, the host process of a log home can be excluded from logging process.

After a failure, writes already performed to other nodes should be replayed for a correct recovery. The page updates are propagated at a synchronization point in the relaxed consistency. Therefore, in conventional schemes, a process should confirm at each synchronization point that its logs will be accessible even after a failure. In remote logging, however, a process can regard a successful sending log messages as the completion of logging without requiring acknowledgements from a log home. It makes sense with a modern communication protocol guaranteeing reliable message delivery.

The logs in a log home, i.e., diffs and write-notices, can be simply discarded when its peer node takes a checkpoint. It is because the logs are only for the recovery of the peer node. On the other hand, a page home also saves received diffs in local memory for servicing old pages while the recovery of other nodes. Thus, a page home should flush such diff logs to a local disk on a checkpoint.

4.2. Diffs

If a failure occurs in a page home, the received diffs should be applied to home pages in the same order for a correct recovery. Thus, a page writer also sends the same diffs to the log home of its page home. The page writer can complete an interval only after the diffs are successfully sent to the page home and then to its log home. It guarantees that the pages performed to other nodes can be recovered with the diff logs.

When a page home receives diff messages, it should also log the diffs in its local memory. The diff logs in a page home are used for regenerating a page. A failed process requires the old copy of a remote page while it replays the computations. When a page home receives such a request from the failed process, it can service the same page as before the failure by replaying the diff logs. The page home should save such diff logs to a local disk on its checkpoint.

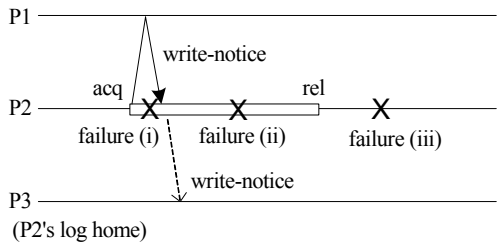

Figure 2. Write-notice logging

We show the correctness of diff logging in Figure 1. A process P1 writes a page *x* and, at release time, sends the diffs to a page home of P2 and the P2's log home, P3. (i) If P2 fails after receiving diffs, it can be recovered with diff logs in P3. However, (ii) P1 can concurrently fail before sending the diffs to the log home P3. In this case, it is certain that other processes did not access the modified data from P2 because P1's release was not performed before the failure. P1 can newly execute the interval and send the new diffs of a page *x* to P2 and P3 without consistency problems.

4.3. Write-notices

In remote logging, when a lock acquirer receives write-notices, it forwards them to its log home. The acquirer sends them through a reliable communication channel and thus needs not wait explicit acknowledgements. Alternatively, a lock releaser can send write-notices to the log home. In this case, the acquirer should check whether the write-notices are successfully logged in its log home before it releases the lock.

Figure 2 shows the write-notice logging during a lock acquire operation. A process P2 receives write-notices from the previous lock owner P1 and then immediately forwards them to its log home of P3. We consider the cases of failures at following points. (i) P2 may fail before sending write-notices to its log home and thus P2 cannot recover the lock acquire. It does not violate memory consistency because both the acquire and the following operations were not performed before the failure. P2 can acquire the lock again after recovery. (ii) P2 may fail after logging write-notices to its log home but before releasing the lock. Such a case does not enforce the recovery of the lock acquire since any writes in the interval were not performed to other processes by HLRC protocol. If the logs are not available, P2 can finish recovery process and start the normal execution from the acquire operation. (iii) If P2 fails after releasing the lock, the interval must be recovered for memory consistency. Before the failure, P2 completed a release operation, which means that it successfully forwarded write-notices to its log home. Thus, P2 can recover the interval with write-notice logs in P3.

During a barrier operation, write-notices are logged in a similar way. The barrier operation is divided to two phases. All processes send write-notices to a barrier manager and its log home in the first phase. After a barrier manager confirms that the write-notices are completely logged in its log home, it sends back up-to-date write-notices to each process and a receiver's log home in the second phase. The number of messages increases during a barrier, but logging overhead can be minimized by sending log messages for the barrier wait time.

5. Recovery

5.1. Single node failures

When a process fails, it rolls back to the last checkpoint and regenerates the same sequence of computational states with logs. Other processes detect the failure and service the requests from the failed process until recovery is finished. We first consider the recovery actions in single node failures.

- **At a lock acquire and a barrier :** A failed process fetches the diff logs recorded before the synchronization point and write-notice logs from its log home. A pointer to the last diff log is recorded with write-notices at each lock acquire, thus the log home can easily send those logs. In case of a barrier, the log home sends diffs created in the intervals corresponding with an up-to-date vector timestamp of the failed process. The failed process applies the received diff logs to its home pages and invalidates the remote pages according to the write-notices. The received diffs are also logged in its memory to support recovery from subsequent failures of other processes. The process in a recovery updates its local vector timestamp as during normal execution.

- **At page faults :** A failed process can reproduce the same sequence of computations only if its all read accesses to remote pages can return the expected values. For correct recovery, a page home services a page request by regenerating an old copy of the page. A page home gets an initial copy from its checkpoint, which does not contain more advanced writes than the faulting access, and then evolves it by applying partially ordered diff logs. It may not service the exactly same pages as before a failure because HLRC is multiple-writer protocol. However, it is sufficient to service the minimal version of the page, which contains only writes happened before the faulting access. Specifically, a page home keeps the first copy of a page requested during recovery. When receiving a subsequent request, it can apply diff logs between the previously serviced version and the expected version to the copy and then send it. Alternatively, a page home can send partially ordered diff logs instead of the page itself and then a failed process can apply diff logs to its local copy. It reduces network traffic when the size of diff logs is smaller than

page size. In our implementation, the latter scheme is applied.

After the recovery is finished, the process whose log home is the failed process takes a checkpoint since its logs are discarded from volatile memory due to the failure. Other processes send again some pending requests to the failed process and resume normal execution.

5.2. Concurrent failures in multiple nodes

In FT-KDSM using remote logging, a process can be recovered even if other processes, except for its log home, are concurrently crashed. At each synchronization point, failed processes can separately execute the recovery process with logs as mentioned above. However, a failed process should interact with other failed processes for page fault handling. During recovery, when a failed process receives a page request from another process, it can immediately regenerate the expected page if required diff logs have been already flushed to disk before a failure. Otherwise, the page fault handling is stalled until the page home fetches the diffs from a log home for its own recovery. It does not make any deadlock conditions because the faulting accesses of different processes are partially ordered according to the happened-before relation.

Figure 3 shows an example of remote logging and our recovery process. In this example, page homes of page *x* and page *y* are processes P2 and P4, respectively. We assume that P2's log home is P3 and P4's log home is P5. In Figure 3(a), P1 acquires a lock for writing page *x* and page *y* during failure-free execution. At the time of lock release, P1 sends the diffs of *x* to the page home of P2 and its log home of P3. Similarly, the diffs of *y* is also sent P4 and P5. The next acquirer P2 receives write-notices from P1 and logs them to P3. At the lock release time, P2 creates the diffs of *x* and logs it in local memory though page *x* is homed page. It is for regenerating page *x* when subsequent failures occur in other processes. P2 also sends the diffs of *y* to P4 and P5 and releases the lock.

Figure 3(b) assumes that P2 crashes at a certain time after releasing the lock and P4 also concurrently fails. P2 and P4 independently roll back to the last checkpoint and replay the logged data. At the lock acquire, P2 get the diffs which have been logged before the lock acquire and write-notice logs from P3. It applies the partially ordered diffs to page *x* and invalidates page *y* according to the write-notice logs.

When P2 accesses on page *y*, it should fetch the page from P4. If P4 has the local diff logs of page *y* at receiving the request, it can regenerate the page by applying them to an initial copy obtained from a checkpoint. Otherwise, it can service the page request when fetching the diff logs from P5 for its recovery. In Figure 3(b), P4 fetches the diff logs of page *y*, which was created by P1, at the lock acquire operation. Though there was happened-before relation

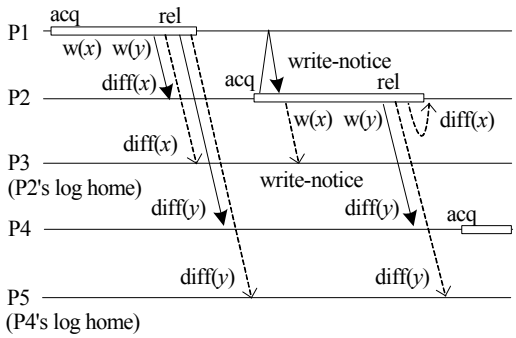

Figure 3. An example of logging and recovery

between the intervals of P2 and P4 during failure-free execution, the intervals can be independently recovered using their logs without any ordering constraints. P4 services the pending request for the page *y* and also logs the diffs in the local memory for subsequent page requests from other failed processes.

6. Performance evaluation

In this section, we describe the hardware and software platform of FT-KDSM and show the evaluation results of remote logging and our recovery protocol. We first compare the performance of remote logging with that of traditional stable logging. Subsequently, we show the recovery speeds of FT-KDSM, which varies according to the number of concurrent failures.

6.1. Experimental setup

Our experiments are performed on a cluster of eight PCs running Linux 2.2.15. Each node contains an 850MHz Pentium III processor. The nodes are interconnected by Myrinet network, which has 1.28Gbps bandwidth and low bit error rates. NIC in each node has a 66MHz Lanai 9.1 processor and 4MB SRAM. We use a GM-VIA [14] provided by Myricom, which is a kind of VIA implementations. FT-KDSM uses DMA write operations of VIA for logging as well as the transfer of pages and coherence-related messages. The memory regions directly accessed by remote nodes should be pinned through VIA

registration. For efficient experiments, we allocated 1GB of physical memory on each node. In FT-KDSM, no checkpoint is taken and all logs remain in memory since we focus on the overhead of only logging and recovery.

In this study, we employ five parallel applications. Table 1 shows the applications and their characteristics. Water, Barnes, FFT and Ocean are from SPLASH-2 benchmark suite [18] and TSP is from Rice University. All of the applications except for Ocean are from CVM distribution [19].

Table 1. Application and their characteristics

Applications	Problem size	Characteristics
Water	1728 mols, 5steps	locks, barriers
Barnes	64K bodies	barriers
TSP	20 cities	locks
FFT	128*128*64 points	barriers
Ocean	258*258 ocean	locks, barriers

6.2. Performance of remote logging

Figure 4 shows the failure-free overhead of our remote logging and traditional stable logging. The execution times are normalized by base DSM performance with no logging. From left to right, the bars for each application are the performance of base DSM, remote logging and stable logging, respectively. Stable logging keeps the messages received during the previous interval in its memory and flushes them to stable storage at a release and a barrier. Similarly to remote logging, it logs the coherence-related data such as diffs and write-notices.

The results show that the remote logging has lower overhead than stable logging in all applications. Remote logging adds little overhead to the execution time of base DSM, ranging from 1% to 11%, over several applications except for Ocean. In FFT and Ocean, while stable logging degrades the performance by 125% and 206%, our remote logging does by only 11% and 21% respectively. Stable logging requires disk accesses for flushing logs at synchronization points, and hence frequent synchronization operations and extensive logs lead to a significant degradation in performance. Specifically, FFT includes a high disk overhead due to the extensive amount of diff logs. Ocean requires not only a large number of synchronization operations but also an extensive logging overhead of write-notices and diffs. The long latency of disk accesses increases the barrier stall time by incurring the difference in execution time between processes. A process also stalls remote page requests during disk operations, thus increasing page fault time. On the other hand, even with extensive amount of logs, remote logging does not significantly increase total execution time because it uses the remote memory over high-speed network instead of disk.

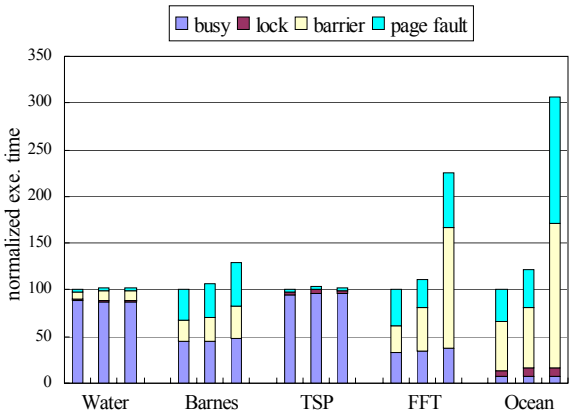

Figure 4. Logging overhead during failure-free execution

The stable logging can be used to tolerate multiple failures since the logs for recovery are always accessible from the local disks of failed processes. However, the system with stable logging should endure a substantial loss of performance during failure-free execution. Our remote logging imposes a little overhead during failure-free execution and tolerates to concurrent failures in some cases.

6.3. Performance of crash recovery

In a system without a crash recovery protocol, all processes have to re-execute the program from the initial state without any logs. Thus, the system consumes the same amount of time for replaying its computation as before a failure. In contrast, with a crash recovery protocol, a failed process can restart the program from only the last checkpoint and replay the execution with logs. Since the starting point of replay varies according to the point of a failure, however, we evaluated recovery time by restarting the failed process from its initial state and only applying logs. Our system does not take a checkpoint.

We evaluated the recovery time increasing the number of failed nodes. In Figure 5, from left to right, the bars for each application represent the recovery times from the failures in a single node, two nodes and three nodes, respectively. They are normalized by re-execution time under no recovery protocol. The results show that the recovery process reduces the cost of synchronization operations compared to the re-execution because the replay of them can be performed with only write-notice and diff logs without real interactions with other processes for synchronization. During recovery, the page fault time includes the overhead of searching and applying diff logs, which is generally small as well. However, it slows down as the number of concurrent failures increases. Page fetch operations from any other recovering nodes are sometimes stalled until the relevant diff logs are applied to the home pages. Due to such interference between concurrently failed processes, the recovery of entire system spends more time than individual recovery process.

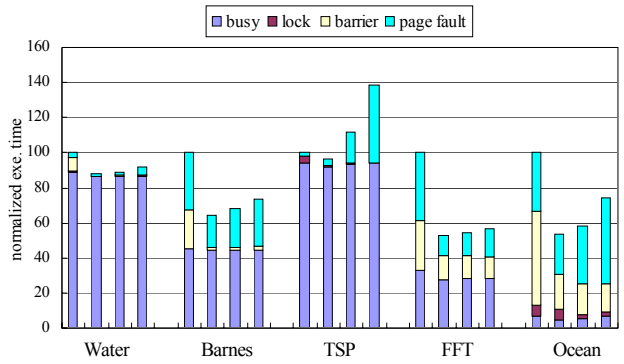

Figure 5. Impacts of the number of failures on recovery time

In TSP, the synchronization time and page fault time occupy only the small portion of application execution. Thus, replaying the logs is unable to save much time in the execution compared to other applications. On the other side, the page fetching time increases considerably as the number of concurrent failures does. Ocean also shows the similar results. In our experiments, the recovery times were measured by assuming the worst case that failed processes roll back to the starting point of the application. In real rollback-recovery systems, they can replay the executions only from the last checkpoint, thus the recovery time may be always smaller than re-execution time even if log replay imposes somewhat high overhead in some cases.

7. Conclusion

We have proposed a lightweight remote logging and a recovery protocol for home-based SDSM in this paper. In contrast with the previous work, our remote logging is lightweight and simple because it fully utilizes the properties of a modern high-speed SAN. Especially, remote logging can tolerate some concurrent failures in multiple nodes since the logs indispensable for recovery of a process are kept in memory of only one remote node.

Evaluation with several well-known benchmarks reveals the tradeoffs between logging overhead during failure-free execution and the degree of reliability. Systems with traditional stable logging can be completely recovered from multiple failures but its logging overhead can significantly degrade the performance of failure-free execution. The logging overhead can offset a benefit of high reliability in some cases. In FFT and Ocean, stable logging leaded to performance degradation by 125% and 206%, respectively. While remote logging imposed relatively low overhead on the base DSM, e.g., by 11% for FFT and 21% for Ocean, it can tolerate some failures in multiple nodes except for concurrent failures with their log homes.

We implemented our recovery protocol and evaluated recovery speed by varying the number of concurrent failures. No prior work has ever focused on recovery from multiple failures. We observed that the synchronization cost tends to be lower during recovery than normal execution and is not affected by the number of processes concurrently failed. However, the page fault time increased due to waiting for the regeneration of home pages, and hence the recovery of a system spent much time when relatively many processes were concurrently failed.

Currently, we are evaluating some practical schemes using logs for enhancing the performance of FT-KDSM. The diff logs can be used to reduce the page fault time during failure-free execution.

References

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

[2] M.Rangarajan and L.Iftode, “Software distributed shared memory over virtual interface architecture: implementation and performance”, In Proceedings of the 4th Annual Linux Conference, pages 341-352, Oct. 2000.

[3] P.Keleher, A.L.Cox, and W.Zwaenepoel, “Lazy release consistency for software distributed shared memory”, In Proceedings of the 19th Annual Symposium on Computer Architecture, pages 13-21, May 1992.

[4] Y.Zhou, L.Iftode, and K.Li, “Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems”, In Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation, pages 75-88, Oct. 1996.

[5] B.Randell, P.A.Lee, and P.C.Treleaven, “Reliability issues in computing system design”, ACM Computing Surveys, 10(2), pages 123-166, June 1978.

[6] G.Suri, B.Janssens, and W.K.Fuchs, “Reduced overhead logging for rollback recovery in distributed shared memory”, In Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing, June 1995.

[7] T.Park, S.B.Cho, and H.Y.Yeom, “An efficient logging scheme for recoverable distributed shared memory systems”, In Proceedings of the 17th International Conference of Distributed Computing Systems, pages 305-313, May 1997.

[8] A.Kongmunvattana and N.F.Tzeng, “Coherence-centric logging and recovery for home-based software distributed shared memory”, In Proceedings of the International Conference of Parallel Processing, pages 274-281, Sept. 1999.

[9] F.Sultan, T.D.Nguyen, and L.Iftode, “Scalable fault-tolerant distributed shared memory”, In Proceedings of Supercomputing, 2000.

[10] G.G.Richard III and M.Singhal, “Using logging and asynchronous checkpointing to implement recoverable distributed shared memory”, In Proceedings of the 12th Symposium on Reliable Distributed Systems, pages 58-67, Oct. 1993.

[11] D.B.johnson and W.Zwaenepoel, “Sender-based message logging”, In Proceedings of 17th International Symposium on Fault-Tolerant Computing Systems, pages 14-19, July 1987.

[12] R.Christodoulopoulou, R.Azimi, and A.Bilas, “Dynamic data replication : an approach to providing fault-tolerant shared memory clusters”, In Proceedings of the 9th International Symposium on High-Performance Computer Architecture, Feb. 2003.

[13] D. Dunning et al., “The Virtual Interface Architecture”, IEEE Micro, 18(2), pages 66-75, 1998.

[14] http://www.myrinet.com

[15] P.Keleher et al., “Treadmarks: distributed shared memory on standard workstations and operating systems”, In Proceedings of the Winter 1994 USENIX Conference, pages 115-131, 1994.

[16] M.Costa et al., “Lightweight logging for lazy release consistent distributed shared memory”, In Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation, pages 59-73, Oct. 1996.

[17] L.Lamport, “Time, clocks, and the ordering of events in a distributed system”, Communications of the ACM, 21(7), pages 558-565, 1978.

[18] S.C.Woo et al., “The SPLASH-2 programs: characterization and methodological considerations”, In Proceedings of 22nd International Symposium on Computer Architecture, pages 24-36, June 1995.

[19] P.Keleher, “CVM: The Coherent Virtual Machine”, Technical Report, University of Maryland, Nov. 1996.