Checkpointing of Control Structures in Main Memory Database Systems

L. Wang, Z. Kalbarczyk, R. K. Iyer

Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 W. Main Street, Urbana, IL 61801
Email: {longwang, kalbar, iyer}@crhc.uiuc.edu

H. Vora, T. Chahande

Mascon IT Ltd.
1699 E. Woodfield Road,
Schaumburg, IL 60173
Email: {hvora, takshak}@masconit.com

Abstract. *This paper proposes an application-transparent, low-overhead checkpointing strategy for maintaining consistency of control structures in a commercial main memory database (MMDB) system, based on the ARMOR (Adaptive Reconfigurable Mobile Object of Reliability) infrastructure. Performance measurements and availability estimates show that the proposed checkpointing scheme significantly enhances database availability (an extra nine in improvement compared with major-recovery-based solutions) while incurring only a small performance overhead (less than 2% in a typical workload of real applications).*

1 Introduction

Main memory database (MMDB) systems store data permanently in main memory, and applications can access the data directly [5]. This offers high-speed access to shared data for applications such as real-time billing, high-performance web servers, etc. However, it also makes database systems highly vulnerable to application errors/failures, as the database is directly mapped into the application's address space.

In addition to user data, the database maintains the control structures (e.g., lock/mutex tables and file tables) necessary for data operation. A database management system (DBMS) maintains data integrity, including recovery in the case of an error/failure of either applications or database services. However, the integrity of control structures is often not well maintained due to the less uniform interfaces to control structures (compared with those to user data). As a result, errors in control structures can become a major cause of system downtime and, hence, an availability bottleneck.

In this paper, we propose and evaluate an application-transparent, low-overhead checkpointing strategy for maintaining the consistency of control structures in a commercial MMDB. The proposed solution is based on the ARMOR architecture and an ARMOR runtime infrastructure [8], [18]. It eliminates (or significantly reduces) cases requiring *database major recovery*, a lengthy process that can take tens of seconds and adversely impact availability. Importantly, the approach can be adapted relatively easily to other applications, and the ARMOR runtime support creates a foundation for providing system-wide error detection and recovery. This work makes the following contributions:

- Introduction of a framework to provide support for checkpointing of MMDB control structures.
- Design and implementation of two checkpointing algorithms. (i) *Incremental checkpointing:* (a) At runtime, a post-transaction (upon transaction completion) state of the control structure(s) accessed by each write transaction (an update of the control structures) is collected and merged with the current checkpoint. (b) At recovery time, the checkpoint is used directly to restore the correct state. (ii) *Delta checkpointing:* (a) At runtime, a pre-transaction (before any updates occur) state of the control structure(s) accessed by a given transaction (both write or read-only) is preserved as a current checkpoint (delta). (b) At recovery time, the current state of control structures in the shared memory is merged with the delta checkpoint to restore the state.
- Performance evaluation of the proposed checkpointing algorithms. The data show that for a rather harsh workload of 60% write transactions, the performance overhead varies in the range of 1% to 10%, depending on the frequency of transactions.
- Database availability estimation under different frequencies of crashes that require major recovery. The data show that under the error rate of one crash per week, a checkpointing-based solution provides about five nines of availability, one nine more than the baseline system.

2 Target System Overview

Target system. The target system in this study is a commercial relational MMDB intended to support development of high-performance, fault-resilient applications requiring concurrent access to shared data [2], [3]. The process accessing the shared data can be either a client or a database service. A *service* is a process that performs functions to assist the proper processing of transactions. For example, the *cleanup* service detects failures of connected clients/services and performs recovery (including launching major recoveries).

In addition to supporting user data, the database supports control structures (*SysDB*) necessary for correct operation. Figure 1 depicts the example architecture of *SysDB* containing three tables: (i) the *process table*, which maintains process *ids* and *mutex lists* for each process as well as information on database mapping into the process address space, (ii) the *transaction table*, which maintains logs and locks for active transactions, and (iii) the *file table*, which keeps user database files. Each client/service process maps *SysDB* into its own address space before accessing the database.

Reliability problem. The error model we address in this paper is the inconsistency of control structures due to the abnormal termination (crash) of one of the clients or services[1]. Upon such a crash, the target system denies services to all other user processes and restarts the entire database system; this is a *major recovery*. It may take tens of seconds, depending on

[1] In the current implementation, we do not detect silent corruption of data, i.e., incorrect data being written to the database.

the size of the data files, and can significantly degrade system availability (not acceptable for services provided to critical applications). A major reason for these problems is the way the database handles access to control structures in *SysDB*. The system employs multiple mutexes to guarantee the mutual exclusion semantic in accessing control structures by user processes. When a client or service crashes while still holding a mutex, the database may remain in an inconsistent state. Since there is no way for the system to identify which updates the crashed client has made, the cleanup process restarts the database to bring the system back into a consistent state. Because major recovery imposes significant system downtime, an approach is needed to eliminate or reduce cases in which it is needed[2].

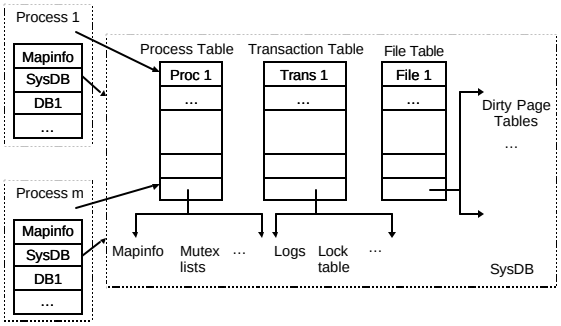

Figure 1: Example Control Structures (*SysDB)*

3 ARMOR High-Availability Infrastructure

The ARMOR infrastructure is designed to manage redundant resources across interconnected nodes, detect errors in both the user applications and the infrastructure components, and recover quickly from failures when they occur. ARMORs (Adaptive Reconfigurable Mobile Objects of Reliability) are multithreaded processes internally structured around objects called *elements* that contain their own private data and provide elementary functions or services. All ARMORs contain a basic set of elements that provide a core functionality, including the abilities to (i) implement reliable point-to-point message communication between ARMORs, (ii) respond to "Are-you-alive?" messages from the local daemon, and (iii) capture ARMOR state.

ARMORs communicate solely through message-passing. The ARMOR microkernel (present in each ARMOR process) is in charge of distributing messages between elements within the ARMOR and between the ARMORs present in the system. A message consists of sequential operations that trigger element actions. This modular, event-driven architecture permits the ARMOR's functionality and fault tolerance services to be customized by choosing the particular set of elements that make up the ARMOR. Several ARMOR processes constitute the runtime environment, and each ARMOR plays a specific role in the detection and recovery hierarchy offered to the system and the application. The *Fault Tolerance Manager (FTM), Heart Beat ARMOR (HB),* and *Daemons* are fundamental components of an ARMOR-based infrastructure. For more details on ARMOR architecture the reader is referred to [8], [17], [18].

[2] In this discussion, we consider only preserving the consistent state of control structures. Any inconsistency brought to the user data is handled/recovered by the default database services, such as two-phase commit, checkpointing, and logging. As long as *SysDB* consistency is preserved, the system can operate correctly and recover user data.

4 ARMOR-based Checkpointing

Embedded ARMORs. In most of cases, an ARMOR launches the application and monitors its behavior. The application is treated as a black box, and only limited services can be provided by the ARMOR infrastructure, e.g., restart of the application process. In the embedded ARMOR solution, an application links the core structure of the ARMOR architecture (the ARMOR microkernel) and uses the ARMOR API to invoke/interface with the underlying element structure of an embedded ARMOR. The embedded ARMOR process appears (i) as a full-fledged ARMOR to other ARMORs and (ii) as a native application process to non-ARMOR processes (e.g., database clients). As a result, the application can take advantage of all services provided by ARMORs (e.g., adding or removing elements to customize ARMOR functionality) without having to change the original application's organization. In this way, the application does not need to be rewritten, and only lightweight instrumentation with a few ARMOR APIs is needed to embed the ARMOR-stub into the application.

ARMOR-based checkpointing. In order to expose ARMOR services, the database is instrumented in two ways: (i) ARMOR stubs are embedded in the database processes, facilitating communication channel(s) between the database server and the ARMOR infrastructure. (ii) Functionalities are embedded for checkpointing *SysDB* data structures; this modifies selected library functions of the database but preserves function interfaces and, hence, is transparent to clients.

Figure 2 illustrates the basic architecture of the ARMOR infrastructure integrated with the target database. The FTM, FTM daemon, HB, and Daemon constitute the skeleton of the ARMOR infrastructure. The solid lines are the ARMOR communication channels. An ARMOR element called the *image keeper* is embedded into the Daemon ARMOR to maintain the image (checkpoint) of the data structures in *SysDB*. When a client or service process opens the database, the database kernel library creates an *Embedded ARMOR (EA)* stub within the process and establishes the communication channel between the EA and the Daemon. From then on, the checkpoint data can be transmitted directly from the source process (with its EA stub) to the destination ARMOR, which maintains the image in memory. The image is then stored on disk by ARMOR's checkpoint mechanism.

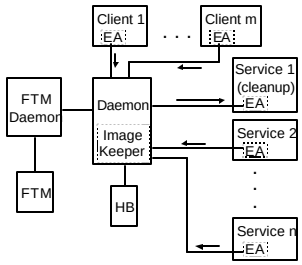

Figure 2: Basic Checkpointing Architecture

Arrows in Figure 2 depict the data flow during system operation. Each client or service, when it acquires a mutex (or re-

leases a mutex, depending on the checkpointing strategy applied), sends the related checkpoint through the ARMOR communication channel to the image keeper. The image keeper processes the message according to the checkpointing strategy. If there is no error, the checkpoint reflects the latest consistent state of the *SysDB*. When a client/service crashes while holding a mutex, the cleanup service requests from the image keeper the saved correct copy of the relevant data structure(s). On the successful restoration of the data, the cleanup service allows the system to continue normal execution without invoking a major recovery.

5 Checkpointing Algorithms

This section discusses two algorithms for checkpointing control structures of the target database system: *incremental checkpointing* and *delta checkpointing*. We begin with brief description, summarized in see Table 1, of similarities and differences between the two proposed alternatives.

5.1 Incremental Checkpointing

In the incremental checkpointing scheme, only updates (incremental changes) to data are sent to the image keeper. The basic algorithm is as follows:

- *After system initialization, the database server sends the image of all the control structures to the image keeper.*
- *In the following processing, a client/service acquires a mutex and then performs operations on the control structures. On each write operation, any changes to the data are stored in the local buffer.*
- *After all updates are successfully finished, the mutex is released, and the client/service delivers the buffered increments to the image keeper for maintaining the up-to-date checkpoint of the control structures.*
- *Upon a crash while the mutex is held, the cleanup service requests from the image keeper the latest checkpoint data and restores the corrupted control structures.*

Handling mutex overlaps. In some cases, a single section of control structures is protected by multiple mutexes. To properly handle this scenario, the image keeper maintains, in addition to the checkpoint, the mapping between a mutex and the data section(s) protected by this mutex. To assist in the mapping, the checkpoint increments sent to the image keeper piggyback the mutex id and the information necessary to identify the correct sections in the control structures.

Figure 3 depicts an example configuration of control structure images kept in the image keeper and illustrates the mapping of mutexes to control structures. With this mapping, overlapped data sections can be protected and their consistency with the corresponding copies in *SysDB* can be preserved, even when multiple mutexes are acquired at the same time.

Handling data access without mutex protection. The proposed algorithm works correctly as long as all updates to control structures are performed within the mutex blocks. There are, however, cases in which control structures are updated directly, without mutex protection, e.g. during database initialization (when it is assumed that no processes try to access the database). While this example is a rather benign case, practice shows that application developers often make somewhat arbitrary decisions and allow the accessing of control structures without mutex protection[3]. Handling such scenarios would require (i) locating, in the application code, all the places of potential updates outside mutex blocks and (ii) augmenting the implementation to ensure the checkpoint in the image keeper is up-to-date. This can be difficult given the size and complexity of real-world applications, such as our target database system. Delta checkpointing, discussed next, is an attempt to alleviate this problem.

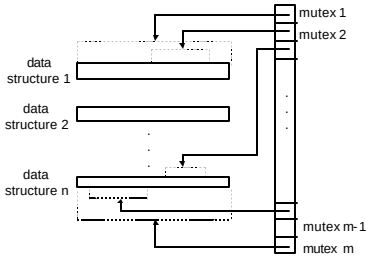

Figure 3: Control Structure Images

5.2 Delta Checkpointing

The delta checkpointing algorithm is based on the following assumption: *In a correctly implemented system, any access to control structures outside the mutex blocks, after system initialization, does not violate data consistency*. Consequently, crashes outside the mutex blocks do not cause data inconsistency, and sections of control structures not updated by any currently executing mutex block are always consistent[4]. As a result, the image keeper does not need to maintain a copy of all control structures (as in incremental checkpointing, discussed in the previous section). It is sufficient to preserve data sections modified (plus information on the type and parameters of the update operation) while executing a given mutex block. In other words, the algorithm only needs to recognize, collect, and send to the image keeper the modified data section, *delta* (*delta* is a before-image). Upon a failure of a client/service while executing a mutex block, the primary copy of the control structures still exists in the shared memory. The entire image of the related structures (*base*) can then be delivered to the image keeper. Using delta and base, the image keeper computes the original (at the time of entering the mutex block) control structures image (*orig* = *base*+*delta*) and sends it back to the cleanup process for recovering the data. The possible updates include data insertion, deletion, and replacement. In summary, the algorithm includes two basic steps:

(1) When a client/service acquires a mutex, it sends the delta *to the image keeper. The delta's content depends on the update to be performed in the mutex block.*

(2) Upon a crash while the mutex is held, the cleanup service sends the base *to the image keeper. The image keeper merges the base with the saved delta and generates the valid image of control structures. The cleanup service requests from the image keeper the regenerated image and restores the corrupted control structures.*

[3] Identification of all cases of updates outside mutex blocks would require reviewing/profiling the entire code base. We could not do this due to limited access to the database.

[4] Under this assumption, incremental checkpointing (Section 5.1) still needs to determine all locations in the code where *SysDB* is updated.

Table 1: Comparison of Incremental and Delta Checkpointing Algorithms

Algorithm	Incremental Checkpointing	Delta Checkpointing
Similarities	Close checkpointing architecture; small overhead; no checkpoint taken for read-only access; same way of handling mutex overlap and access to external devices while holding a mutex.	
Differences	1. At start time – an image of all control structures is stored as an initial checkpoint; 2. At runtime – a post-transaction (upon transaction completion) state of the control structure(s) accessed by each write transaction is collected and merged with the current checkpoint; 3. At recovery time – the checkpointed image of control structures is directly loaded to the shared memory from the image keeper; 4. Must checkpoint data updates due to operations within and outside mutex blocks.	1. At start time – an initial checkpoint is an empty data set, i.e., no need to store any control structures; 2. At runtime – a pre-transaction (before data updates occur) state of control structure(s) accessed by a given transaction (write or read-only) is preserved as a current (delta) checkpoint; 3. At recovery time – the image of control structures from the shared memory is merged with the latest delta stored in the image keeper; 4. Must checkpoint only updates due to operations within mutex blocks.

Observe that the image keeper merges the *base* with the latest *delta* it receives. To avoid using the wrong delta in the case of recovery, it is important to send out a delta at each mutex acquisition, even if the delta is empty, i.e., no changes to the control structures were performed during the current mutex block.

5.3 Image Keeper

The *image keeper* is a separate element within the ARMOR process that collects and maintains checkpoint data representing the correct state of control structures. It is a passive component, which means that it is only invoked by the incoming messages (checkpoint updates) and that it performs proper actions according to the received messages. Figure 4 illustrates the basic structure of the image keeper, which consists of (i) a set of memory blocks for preserving images of control data structures (*control structure images*) and (ii) management support (*manager*) for updates of the checkpoint and recovery actions in response to client failures. The image keeper communicates with the database processes by means of the ARMOR communication channel.

The *control structure images* in Figure 4 represent a memory pool that stores the images of control structures in *SysDB*. Different mutexes map their corresponding data sections into the copy of control structures in the image keeper (Figure 3).

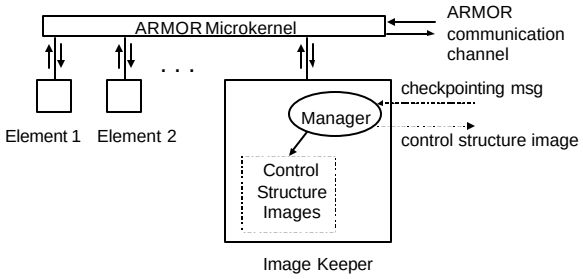

Figure 4: Structure of the Image Keeper

6 Performance Evaluation

This section presents performance measurements of the prototype implementation of the ARMOR-based incremental checkpointing scheme applied to the target database system[5]. The testbed consists of a Sun Blade 100 workstation running the Solaris 8 operating system on top of a 500MHz UltraSPARC-II CPU with 128MB of memory. The measurements are conducted in (i) error-free scenarios, in which normal operation of the database under a synthetic workload mimics actual database activity, and (ii) error-recovery scenarios, in which the database recovers from the checkpoint after a failure while executing transactions issued by synthetic clients. While checkpointing is applied in the context of the *file table mutex*, the proposed solution applies to other mutexes as well.

6.1 Performance of ARMOR-based Incremental Checkpointing in Error-Free Scenarios

Workload. Each workload invocation involves execution of a sequence of transactions, which arrive with a predefined frequency, and each transaction is represented as a set of operations of variable execution time. Some of the operations need to acquire the file table mutex (*mutex*) to preserve mutual exclusion in accessing shared data by multiple clients.

Operations associated with mutex acquisition can be either data write (*write*) or data read (*read*). The operation pattern within a transaction is a sequence of alternate reads and writes, e.g., *read – write – read – write – ….* The *sleep* function is used to emulate the execution time of operations that (i) do not require mutex acquisition (mutex-free operations), (ii) occur when the mutex is held (mutex operations), or (iii) represent idle time, i.e., the period after completion of a current transaction and before arrival of the next.

Parameters. The workload is flexible and can be configured to mimic actual execution scenarios. The tunable workload parameters (and other experiment settings) are as follows:

- *Transaction frequency (freq)* – number of transactions arriving within one second.
- *Number of mutex acquisitions per transaction* (*num_acq*).
- *Percentage of read-only operations (read_per)* – fraction of mutex acquisitions for read-only operations.
- *Mutex operation time (mutex_op)* – processing time while holding the mutex, i.e., the interval between the time the mutex is granted and the time it is released.

[5] Due to the limited time for accessing the target system, we provide measurements only for incremental checkpointing. Because checkpoint data transmission time dominates the performance of both schemes in error-free scenarios and both algorithms transmit a similar amount of data, it is expected that the performance of the two schemes is similar.

- *Mutex-free operation time* (*other_op*) – processing time for mutex-free operations within the transaction.
- *Delivered data* – the amount of checkpointed data. In the case of the file table mutex, the typical data to be checkpointed is a single file entry in the file table (approx. 3700 bytes). In all measurements, the size of the checkpointed data is assumed to be 4000 bytes.
- *Experiment duration* – time duration of the experiment.

The transaction frequency should satisfy the following requirement for experiments to run correctly:

*1/freq >= mutex_op*num_acq + other_op + time{get/release mutexes+checkpointing}*

Results. The workload configuration parameters for performance measurements are as follows: *num_acq=5, mutex_op=0.002s, other_op=0.03s, delivered data=4000 bytes, experiment duration=20s.*

Table 2 shows the time per transaction (with 95% confidence intervals) for four transaction frequencies and with read-only percentages *(read_per)* ranging from 0% to 100%. The transaction time includes mutex-free operations, mutex operations, mutex acquisition/release, and checkpointing time. Table 3 depicts the performance overhead of checkpointing per transaction. The case of *read_per=100%* is the one without checkpointing, and hence, it is the baseline against which the overheads in other scenarios are computed and compared.

Table 2:Transaction Time [*s*]

freq (1/s)	*read_per* (%)					
	0	20	40	60	80	100
0.75	0.109 ±.0001	0.102 ±.0001	0.102 ±.0002	0.102 ±.0002	0.101 ±.0002	0.101 ±.0007
1.25	0.115 ±.0095	0.104 ±.0027	0.103 ±.0008	0.104 ±.0017	0.102 ±.0007	0.102 ±.0009
1.75	0.124 ±.0122	0.115 ±.0070	0.107 ±.0039	0.104 ±.0026	0.105 ±.0018	0.104 ±.0015
2.25	0.130 ±.0116	0.121 ±.0068	0.113 ±.0067	0.103 ±.0015	0.102 ±.0002	0.102 ±.0001

Table 3: Performance Overhead of Checkpointing per Transaction [*s*]

freq (1/s)	*read_per (%)*					
	0	20	40	60	80	100
0.75	0.008 (7.9%)	0.001 (1.0%)	0.001 (1.0%)	0.001 (1.0%)	0 (0%)	N/A
1.25	0.013 (12.7%)	0.002 (2.0%)	0.001 (1.0%)	0.002 (2.0%)	0 (0%)	N/A
1.75	0.020 (19.2%)	0.011 (10.6%)	0.003 (2.9%)	0 (0%)	0.001 (1.0%)	N/A
2.25	0.028 (27.4%)	0.019 (18.6%)	0.011 (10.8%)	0.001 (1.0%)	0 (0%)	N/A

From the results, one can see that as transaction arrival rate increases, the performance overhead and, hence, the transaction time increases. This is because when there are more requests, ARMORs take more time to process each individual request. However, the frequency increase does not significantly degrade the performance; the overhead ranges from 1ms to 28ms for 80% and 0% (all transactions are write) read operations, respectively. If more than half of the mutex acquisitions are read-only, the performance overhead is very small. Note that the variation of the run time without checkpointing (last column in Table 2) can dominate the performance overhead (columns 5 and 6 in Table 2). In real applications, more than 50% of mutex acquisitions are for read-only operations; the measurement data indicate that under such workloads the overhead due to checkpointing is negligible.

6.2 Performance of ARMOR-based Incremental Checkpointing in Error-Recovery Scenarios

The database used in the test consists of five *db files*. Each file contains 100 tables, and each table contains two thousand 200-byte records. So the total size of the user database is 200MB, a typical size for the database the target system processes in practice. Three clients are used in the error recovery scenario: (i) *testsc*, which updates the table records one after another without acquiring a file table mutex, (ii) *testsc_ftmutex*, which repeatedly acquires a file table mutex, updates the table records, and releases the mutex, and (iii) *ftmutextest*, which gets a file table mutex and emulates a crash while still holding the mutex. Table 4 presents measurements comparing the performances of both major recovery and ARMOR-based incremental checkpoint recovery. The time listed in Table 4 represents the recovery time, i.e., the time from the crash of the failed client (*ftmutextest*) to the first successful acquisition of a file table mutex by the waiting client (*testsc_ftmutex*).

Table 4: Performance of Major Recovery and ARMOR-based Incremental Checkpointing

Trial	Major Recovery [s]		ARMOR-based Checkpointing [s]	
	Expr 1 (testsc_ftmutex + tmutextest)	Expr 2 (testsc + testsc_ftmutex + ftmutextest)	Expr 1 (testsc_ftmutex + ftmutextest)	Expr 2 (testsc + testsc_ftmutex + ftmutextest)
1	13	31	2	3
2	12	25	3	6
3	11	29	4	5

Four experiments (each is performed for three trials) are conducted with different clients and recovery policies. The results reported in Table 4 indicate:

- Major recovery can cause significant system downtime (11 to 31 seconds in our experiments). The downtime depends on how much data is loaded into memory when major recovery occurs. (*Testsc_ftmutex* updates a small fraction of table data, so recovery time is small. When *testsc+testsc_ftmutex* is used, since *testsc* updates a whole table, the loaded data is much larger, and recovery time is greater.)
- ARMOR-based incremental checkpointing eliminates or significantly reduces the downtime due to the client crashes: (i) the crash of a client does not impact other processes as long as they do not acquire the same mutex as the terminated client, (ii) an overhead of 2 to 6 seconds (in our measurements) is encountered by any process that attempts to acquire the same mutex as the terminated client, and (iii) recovery using checkpointing does not depend on the amount of loaded data, as there is no need to reload data.

Availability. Availability of the database is estimated based on the data on recovery time, assuming different frequencies of crashes that require major recovery. (We consider database availability using Experiment 2 as an example.) The measured average recovery time for major recovery and checkpointing-based recovery are 28.3s and 4.7s, respectively. Table 5 shows the system's availability for various error frequencies. One can see that under an error rate of one per week, checkpointing-based recovery provides about five nines of availability, which is one nine of improvement compared with the major-recovery-based solution.

Table 5: Availability (Expr. 2)

Solutions	Error Frequency				
	1/hour	1/day	1/week	1/month	1/year
Major recovery	99.21	99.97	99.995	99.9989	99.99991
Checkpointing-based recovery	99.87	99.995	99.9992	99.9998	99.99999

7 Related Work

A number of checkpoint techniques have been proposed to ensure the durability of MMDBs. In Hagmann's *fuzzy checkpointing* [6], the checkpoint is taken while the transaction is in progress. An improved variant of fuzzy checkpointing is proposed in [11] and [12]. Non-fuzzy checkpointing algorithms are introduced in and [7], [9], and [14].

Levy and Silberschatz [10] design an incremental checkpointing scheme that decouples transaction processing and checkpointing. The *propagator* component observes the log at all times and propagates the updates of the primary copy in memory to the backup copy on disk. While these traditional techniques rely on control structures to checkpoint user data, we address checkpointing of control structures themselves.

Sullivan and Stonebraker [16] investigate the use of hardware memory protection to prevent erroneous (due to addressing errors) writes to the data structures. In [4], Bohannon, et al., achieve such protection by computing a codeword over a region in the data structures. Upon a write, the data region and the associated codeword are updated. A wild write results in an incorrect codeword, which triggers recovery of the corrupted data region. These schemes protect the critical control structures against erroneous writes. Our checkpointing algorithms defend against client crashes and data inconsistency, which is a different failure model. Another technique that addresses this type of failure is *process duplication*. For example, Tandem's process-pair mechanism [1] provides a spare process for the primary one. The primary executes transactions and sends checkpoint messages to the spare. If the primary fails, the spare reconstructs the consistent state from the checkpoint messages. The idea of lightweight, recoverable virtual memory in the context of providing transactional guarantees to applications is explored in [15]. A Rio Vista system for building high-performance recoverable memory for transactional systems is proposed in [13].

8 Conclusions

This paper presents ARMOR-based, transparent, and performance-efficient recovery of control structures in a commercial MMDB. The proposed generic solution allows eliminating or significantly reducing cases requiring major recovery and, hence, significantly improves availability. The solution can be easily adapted to provide system-wide detection and recovery. Performance measurements and availability estimates show that the proposed ARMOR-based checkpointing scheme enhances database availability while keeping performance overhead quite small (less than 2% in a typical workload of real applications).

Acknowledgments

This work was supported in part by NSF grant ACI-0121658 ITR/AP. We thank F. Baker for careful reading of this manuscript.

References

[1] J. Bartlett. A nonstop kernel. *Proc. Eighth Symposium on Operating Systems Principles*, 1981.

[2] P. Bohannon, et al. The architecture of the Dali main memory storage manager. *Journal of Multimedia Tools and Applications*, 4(2), 1997.

[3] P. Bohannon, et al. Distributed multi-level recovery in main memory databases. *Proc. 4th Int. Conf. on Parallel and Distributed Information Systems*, 1996.

[4] P. Bohannon, et al. Detection and recovery techniques for database corruption. *IEEE Trans. on Knowledge and Data Engineering*, 15(5): 2003.

[5] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Trans. on Knowledge and Data Engineering*, 4(6), 1992.

[6] R. Hagmann. A crash recovery scheme for a memory-resident database system. *IEEE Trans. on Computers*, 35(9), 1986.

[7] J. Huang and L. Gruenwald. An update-frequency-valid-interval partition checkpoint technique for realtime main memory databases. *Workshop on Real-Time Databases*, 1996.

[8] Z. Kalbarczyk, et al. Chameleon: A software infrastructure for adaptive fault tolerance. *IEEE Trans. on Parallel and Distributed Systems*, 10(6), 1999.

[9] T. Lehman and M. Carey. A recovery algorithm for a high-performance, memory-resident database system. *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, 1987.

[10] E. Levy, and A. Silberschatz. Incremental recovery in main memory database systems. *IEEE Trans. on Knowledge and Data Engineering*, 4 (6), 1992.

[11] X. Li, et al. Checkpointing and recovery in partitioned main memory databases. *Proc. IASTED/ISMM Int. Conf. on Intelligent Information Management Systems*, 1995.

[12] J. Lin and M. Dunham. A performance study of dynamic segmented fuzzy checkpointing in memory resident databases. *TR 96-CSE-14*, Southern Methodist University, Dallas (TX), 1996.

[13] D. Lowell and P. Chen. Free transactions with Rio Vista. *Proc. 16th ACM Symposium on Operating Systems Principles*, 1997.

[14] K. Salem and H. Garcia-Molina. Checkpointing memory-resident databases. *Proc. Int. Conf. on Data Engineering*, 1989.

[15] M. Satyanarayanan, et al. Lightweight recoverable virtual memory. *Proc. 14th ACM Symposium on Operating Systems Principles*, 1993.

[16] M. Sullivan and M. Stonebraker. Using write-protected data structures to improve software fault tolerance in highly available database management systems. *Proc. Int. Conf. on Very Large Databases*, 1991.

[17] K. Whisnant, et al. An experimental evaluation of the REE SIFT environment for spaceborne applications. *Proc. Int. Conf. on Dependable Systems and Networks*, 2002.

[18] K. Whisnant, Z. Kalbarczyk, and R. K. Iyer, A system model for dynamically reconfigurable software. *IBM Systems Journal*, 42(1), 2003.