Highly Available ACID Memory
Project Report for CS258

Vijayshankar Raman

May 17, 1999

Abstract

Recovery logic is one of the most complex parts of applications such as Database Management Systems (DBMSs), and is very hard to debug or maintain. We present a simple and light-weight ACID memory library for use by such applications, which want to access data in a persistent, atomic manner. It supports concurrency control and recovery in a unified framework, and applications need to only acquire appropriate locks on virtual memory regions in order to become recoverable. We use replication and non-volatile memory for providing recovery, and have extended our approach to provide high availability by replicating across multiple nodes in a cluster. Micro-benchmark evaluations show that performance of ACID memory is comparable to that of more sophisticated logging-based approaches, even without careful application programming. We also find that replicating across multiple machines provides very fast recovery, while still supporting fast transactions.

1 Introduction

Most non-trivial applications have some critical state that they would like to access and manipulate in a persistent, atomic manner in the failure of hardware or software failures. ACID properties [GR93] are widely used as a correctness metric for manipulating such state information: updates must be atomic and persistent at transactional boundaries, and concurrent operations by multiple threads must be synchronized by acquiring read and write locks.

With the advent of the Internet, a second property that applications desire while accessing data is availability. Since most Internet services want to run 24x7, they want to be able to access and manipulate data continuously even in the face of failures. The goal now is not just eventual recovery and correctness, but rather almost application-transparent failure handling. Database Management Systems (DBMSs) are widely used to store and manipulate such information, because they provide ACID transactional behaviour, and often come with Transaction Processing Monitors [GR93] that provide availability.

Unfortunately, concurrency control and recovery are the most complex parts of a DBMS. To quote Stonebraker and Rowe, *"Most DBMSs have a large amount of crash recovery code that is tricky to write, full of special cases, and very difficult to test and debug"* [SR86]. More importantly, the complex logic used for recovery runs through most of the DBMS code, including the storage system, the buffer manager, the index structures and access methods, and the query processing system. At the same time, recovery is one of the critical functions of a DBMS. Applications and organizations typically place important data in DBMSs for reliability, and loss of such data due to bugs in the recovery handling code is disastrous.

1.1 Project goals

In this project we seek to design and develop a user-level ACID virtual memory library for DBMSs, and for other applications that want transactional recovery. Our goal is to take the recovery logic out of the

application[1]. This has two advantages

1. It simplifies the application code, and application writers need not worry about recovery. By moving all the recovery logic to one place, we make it easier to debug, test, and maintain.

2. Since recovery makes use of the persistence properties of storage devices such as disks, the optimal recovery logic changes with changes in hardware technology. The trade-offs in the memory hierarchy between volatility, access speed, and cost dictate the mechanism to be used for recovery [CL99]. Unfortunately, DBMSs have typically found it difficult to make use of hardware improvements because changing the recovery logic would involve rewriting much of the DBMS. For example, although researchers [SR86, NC97] have demonstrated orders of magnitude performance improvements by using battery-backed memories instead of disk for storing logs, commercial vendors have not implemented these. Separating the recovery logic makes it easy to change as hardware technology improves.

A second goal of our project is to handle availability within the framework of our ACID memory. Traditionally, availability has been handled in a separate layer on top of the DBMS. A special Transaction Processing (TP) monitor provides availability and protection against disk crashes by replicating data across multiple machines. This approach often requires expensive specialized hardware and/or software such as *Process pairs* (for e.g. Tandem NonStop Servers [Tan]). Moreover this approach is typically quite wasteful, because updates are logged by the DBMS for recovery, and are replicated by the TP monitor for availability.

In our project, we want to study whether we can easily extend our recovery mechanism to support availability using a cluster of commodity workstations. Handling availability along with recovery in this manner a) reduces the system cost considerably, b) removes a layer of software from the system and c) can potentially improve performance if we use replication directly for recovery, instead of duplicating our efforts in single-node recovery and multi-node replication.

1.2 Design decisions

We use replication and non-volatile memory for recovery, instead of writing logs. Non-volatile memory is fairly inexpensive nowadays [NC97], and memory can be made safe against OS crashes using virtual memory protection [C+96]. Using replication instead of logging, and assuming memory non-volatility, makes the ACID memory much simpler to implement.

We provide support for concurrency control along with recovery; the unit of recovery and atomicity is the set of all regions that a transaction has write-locked (we explain this further in Section 2). This is a crucial distinction between ACID memory, and several efforts at transactional or recoverable memories [S+94, HM93, C+96]. As we argue in Section 2, leaving concurrency control to the application results in considerable semantic confusion and potential for errors for the application writer. This decision seriously impacts our recovery system – since DBMSs often want multiple granularity (record level, page level, etc.) locking, we need to provide recovery at a level smaller than a page.

1.3 Questions to answer

Starting with these assumptions, we seek to answer two questions by prototyping a highly available ACID memory library.

First, where is the tradeoff between application programming ease and performance, as far as recovery is concerned? Traditionally, DBMSs use complex logging based schemes for recovery. How simple will it be to build applications using the ACID memory library, and how will they perform relative to applications that explicitly log updates?

[1] Hereafter, we will use application to refer to both DBMSs, and applications that directly manipulate critical data in a recoverable manner. However, the design of our system is optimized for DBMSs

Second, on a cluster, can we use replication directly for high availability, without a separate availability layer on top of the recovery system?

1.4 Outline

We present the application's interface to ACID memory in Section 2. We describe the design and implementation of ACID memory on a single node in Section 3, and evaluate the performance and programmability relative to approaches that require explicit logging in Section 4. We extend our design to provide availability using multiple nodes in Section 5. We briefly look at related work in Section 6 and conclude with directions for further work in Section 7.

2 Acid Memory API

An application using ACID memory has two components. A *transaction manager* to manage transactions, and multiple *transactions* to manipulate data. In our current implementation all transactions are required to be threads in the same address-space as the transaction manager, although this is not critical to the design.

2.1 Transaction Manager Interface

The transaction manager interface is captured in a single function
new TransactionManager(database name, acid memory region)
The application uses a database name to correspond to a particular database that it wants to manipulate in ACID fashion. The database name provides a consistent way of accessing the database, across system/application crashes. The acid memory region is a range of virtual memory that the application wants to be mapped to the data in the database. An application can access multiple databases with different transaction managers.

2.2 Transaction Interface

The transaction interface consists of four functions:
beginTransaction(transaction manager) to start a transaction,
getLock(memory region, lockType) to get a READ or WRITE lock on a virtual memory region, and
commit() or abort() to end the transaction

A typical transaction will begin, acquire one or more locks on various regions as they are needed, manipulate data, and then commit or abort. All locks need not be acquired at the start of the transaction; an application can acquire locks just before accessing the data. All locks are released at the end of the transaction only (this ensures that other transactions don't see its changes before it commits and is needed for correct recovery [Ram98]). The virtual memory region to lock is specified as a virtual address prefix: the region encompasses all addresses with that prefix. We make this restriction, instead of allowing locks over arbitrary ranges, in order to simplify lock handling. All transactions are assumed to have aborted when the system crashes and comes back up.

2.3 Discussion

There are two important decisions hidden in this API. First, we allow applications to directly modify data. This is in contrast to logging based approaches that require applications to manipulate data through special "update" functions which write logs before modifying the value. Any application can now become recoverable by simply inserting locks before modifying any critical data (or even this could be done by a compiler) and does not need significant rewriting. We discuss the performance tradeoffs of this in Section 4.

Second, we combine concurrency control with recovery in the API – recovery is done on write-locked regions. This avoids confusion to application writer, because he only needs to keep track of the regions he updates. If concurrency control were left to the application writer, he needs to acquire a lock on a region for synchronization, and then separately inform the transaction manager that he is going to update the region. Doing only one of these steps, or specifying intent to update on a different granularity than the granularity of locking, will lead to incorrect behaviour. Moreover, the application writer *must* release all locks at the very end of the transaction, to avoid the recovery problems explained before. All these issues are automatically taken care of in our API.

3 Single node ACID memory design and implementation

We assume the existence of a persistent OS file cache (see Rio [C+96] for a description of how to modify the OS reboot process to make the file cache persistent given non-volatile memory), and use it as a basis to build ACID memory.

Given a persistent file cache, we mmap the acid memory area from a file to get persistent access to it. All updates are performed directly on the acid memory region, and are therefore immediately persistent. In order to undo the changes in the event of a system crash or transaction abort, we make an undo copy of the data in a *mirror area* when a transaction acquires a write lock. The mirror area is also mmap'd onto a disk file to keep it persistent. Thus, the only overhead of ACID memory is a memory copy on each write-lock. Commits, and read-locks are free.

In our actual implementation, since we do not have non-volatile memory or a persistent file cache, we can only provide recovery from software crashes. Every time a new transaction manager is created, we spawn a *backup process* that also mmaps the acid memory region from the same file (with the MAP_SHARED option, so that it can see changes). The backup process periodically polls (via kill messages) to see if the process that started the transaction manager has crashed. If it has, then the backup process does an msync on the mapped region so that any changes in the acid memory region are made permanent.

Recovery: When a transaction aborts, we undo changes on all write-locked regions using the copy in the mirror area. When the system crashes and comes back up, we need to undo the changes of all aborted transactions. Therefore we need to maintain a list of all write-locked regions. Again, we mmap this list to a file to ensure that it is persistent.

Lazy aborts: Having such a list of write-locked regions allows us to perform transaction aborts in a lazy fashion. When a transaction aborts, we return immediately, and we need to actually perform the undo only when we run out of memory to store the list of write-locked regions. In fact, if we can postpone the abort until another transaction request a write-lock on the same region, we can avoid a memory copy for the new transaction. In effect, we have performed the abort for free.

4 Evaluation of Acid Memory

We evaluate our acid memory scheme along three dimensions: simplicity of implementation, ease of programming applications, and transaction performance, with respect to more sophisticated methods that explicitly log update.

4.1 Simplicity of ACID memory implementation

As the previous section shows, our ACID memory scheme is fairly simple. Our current implementation comprises only 644 lines of commented code, about 50% of which is spent on lock management. The simplicity arises mainly because a) persistent memory implies that we never need to redo updates and so we need not have a redo log, and b) we use replication for undo recovery instead of managing an undo log. In

Using ACID Memory

```
/* a points to A's account */
/* b points to B's account */

trans = new Transaction(transactionManager);
trans→getLock(a, WRITE);
trans→getLock(b, WRITE);
a = a − 1000000;
b = b + 1000000;
trans→commit();
```

Using Logging

```
/*Update() creates the needed logs*/
BeginTransaction();
getLock(A's account, WRITE);
getLock(B's account, WRITE);
read(A's account, a);
read(B's account, b);
a = a − 1000000;
b = b + 1000000;
Update(A's account, a);
Update(B's account, b);
commit();
```

Figure 1: A sample application programmed with ACID memory and with logging

contrast a typical DBMS using logging needs a) complex buffer management code (because when a page is flushed onto disk its changes are immediately persistent), and b) a sophisticated log manager that handles log compaction and spools log records onto disk every time a transaction commits or a page is flushed from the buffer [HR83].

4.2 Complexity of Application programming using ACID memory

As mentioned before, the only changes that an application has to make in order to use ACID memory are a) it must call beginTransaction and commit at transactional boundaries, and b) it must acquire suitable locks before accessing data in the ACID memory region.

Figure 1 shows a sample application programmed using ACID memory and using a conventional logging based mechanism. The logging based mechanism that we use is similar to the interface provided by a DBMSs' storage manager to its query processing system. The application involves transferring some money from Account A to Account B in an atomic manner. The principal difference between the two approaches is that with ACID memory, the application can access the data directly, whereas with logging the application needs to explicitly update the data by a call to an Update function.

4.3 Performance evaluation

We have seen that ACID memory is simple to implement and relatively easy to program with. We next use micro-benchmarks to check if we lose in terms of performance by using replication instead of logging. We ran all experiments on lightly loaded Sun Ultra-1's with 256 MB main memory (NOW machines). For all experiments, we ran 100000 transactions and took average times.

Latency for a simple transaction: We first consider a simple transaction that write-locks a single memory region, updates it, and commits. Figure 2 shows the difference between the time taken for this transaction and the time taken for doing the update on normal (non-ACID) memory, as a function of the size of write-locked region. The baseline overhead of about 35 microseconds is the lock acquisition overhead (this is the only overhead for a read-lock). Since our aim is study the efficacy of acid memory for recovery we have not optimized the performance of our lock manager. For large region sizes, the overhead per write-lock scales roughly linearly with the region size at about 5.47 microseconds/KB. This corresponds to the memory copy overhead.

Comparison with logging: We next compare the performance of ACID memory with an approach based on logging. The benchmark that we use in this case involves a transaction that write-locks a 1 KB

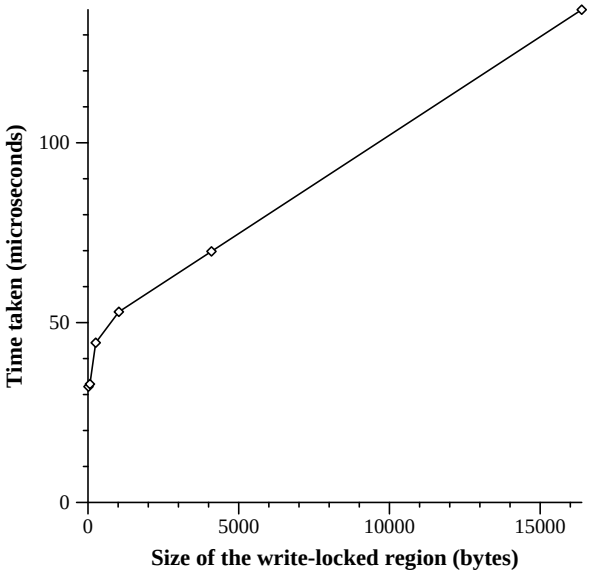

Figure 2: ACID memory overhead for a transaction that write-locks a single region

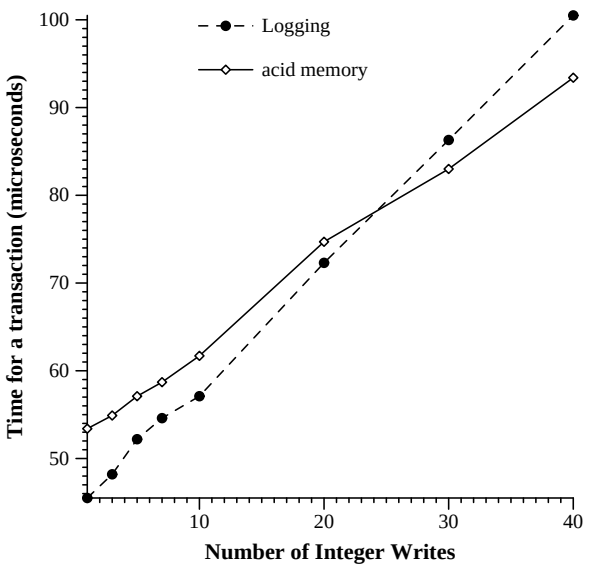

Figure 3: Time taken for a transaction that updates multiple integers

data-structure, updates a number of integers within the region, and then commits. We compare two implementations. The first uses ACID memory and, performs getLock on the entire region, and then manipulates the integers directly. The second does not write lock the entire 1KB region. Instead it updates each of the integers by explicitly calling an Update function which logs the operations[2]. The second approach avoids memcpy'ing the entire region, but requires that the application manipulate data in a specific manner. Figure 3 shows the time taken as a function of the number of integers updated. We see that ACID memory is only about 15 to 20% slower than explicit logging, even when we update only a few integers. Beyond 25 updates the overhead of calling the Update function causes logging to be actually slower than using ACID memory.

5 Multiple-Node Highly available ACID memory

Availability has traditionally been handled in a separate layer on top of the DBMS. A special Transaction Processing (TP) monitor provides availability and protection against disk crashes by replicating data across multiple machines (see Figure 4). This approach is quite wasteful, because updates are logged by the DBMS for recovery, and are replicated by the TP monitor for availability.

We now consider how to make ACID memory available by using multiple processors. We use replication across multiple machines on a cluster for availability and recovery, instead of adding a separate availability layer to a recoverable system. This avoids redundant replication, and allows us to get fast transactions without relying on non-volatile memories. It also simplifies crash-handling, because crashed nodes need not recover their state.

[2]Our Update function does not actually log anything but merely does a memcmp of the new value with the old value. A real implementation will be more expensive because it has to difference the two regions and append a log record to the log.

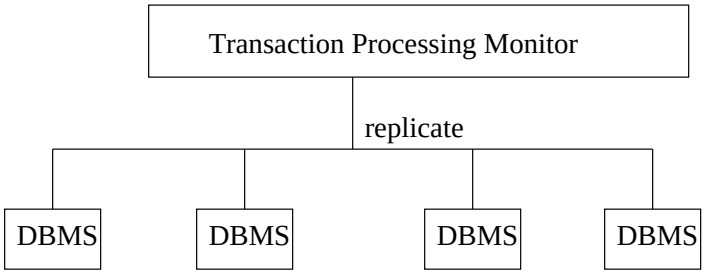

Figure 4: The TP monitor approach to high availability

5.1 Architecture

Figure 5 shows the architecture of our system. For consistency, the acid memory area has a single *lock manager* residing on the *Owner* node. The data is replicated across multiple nodes, and all these copies are synchronized at transaction boundaries. Transactions are run by one or more *transaction handlers*. All lock requests go to the owner, but data requests can go to any of the replicas. Since this architecture places an extra burden on the Owner node, we balance the load by partitioning the acid memory region into multiple partitions, each of which has a different owner.

Failure Model: We assume a fail stop failure model where crashed nodes never send incorrect messages to other nodes. We also assume that the network is never partitioned.

Transaction Processing: All lock requests are sent to the Owner node. After a transaction gets a lock, it asks for a copy of the locked region from a randomly chosen replica. When a transaction commits, it sends copies of all write-locked regions to the Owner, who needs to propagate the changes atomically to all replicas.

Nonblocking Atomic commit: When a transaction commits and sends its updates to the Owner, the Owner needs to propagate these to all replicas in an atomic manner: either all replicas must get the update, or none must. For instance, if there are 5 replicas and Owner sends the updates to 2 of them and then crashes, the system is in an inconsistent state. The traditional database approach to solve this problem is 2-phase commit [MLO86]. The Owner logs a copy of the changes and the list of replicas before propagating the changes so that it can recover after a crash and send the updates to all the replicas. However this approach *does not work if we want high availability.* Since we want applications to continue executing without waiting for crashed nodes to recover, there is no point in logging the updates at the Owner. Instead we use a *Nonblocking 3-phase commit protocol* [Ske81]. The Owner first sends a copy of the updates to all the replicas. The replicas do not apply the updates in place (doing so would mean that the replicas have transient dirty data and therefore cannot be used for fault-tolerance), but instead copy them to the mirror regions. After the replicas have acknowledged receipt of the data, the Owner sends a PREPARE message to all replicas to signal that everyone has received a copy of the data. Finally the Owner sends a COMMIT message to all replicas. When the replicas receive a commit message they can commit the transaction. [Ske81] gives a proof of correctness for this protocol and the following crash recovery mechanism.

Crash Recovery: Crashes by any of the replicas are insignificant – the Owner merely notes the failure[3] to avoid repeatedly sending data to a crashed node. If the Owner were to fail after sending a prepare message to only a subset of the replicas, any in this subset can detect that the Owner has failed and propagate the PREPARE message to all the other replicas. If the Owner were to fail before sending any PREPARE message, the replicas that received the data detect the failure and abort the transaction. We use a predefined static ordering among the nodes to decide who becomes the new Owner when the current Owner crashes.

[3] All crashes are detected via timeouts

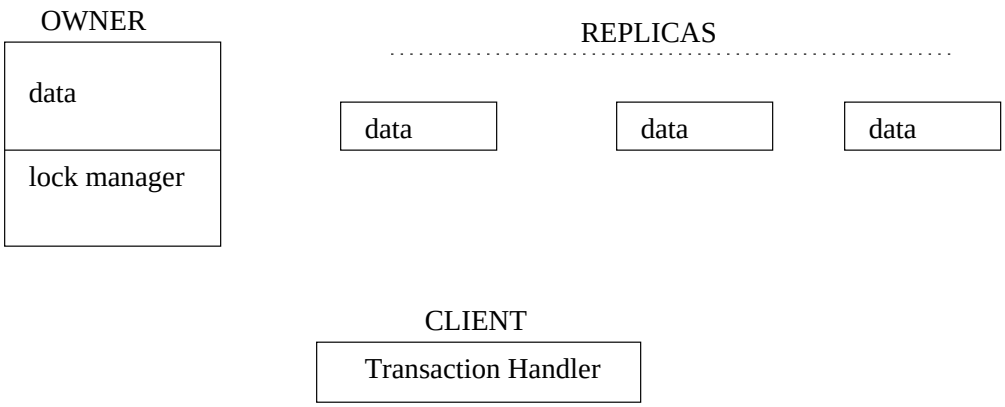

Figure 5: Replication architecture for high availability

5.2 Performance Evaluation

If there are n replicas, the overhead of the above algorithm is 4n control messages in addition to the n data messages that are needed to propagate the updates. While this may appear large, this is still much smaller than the time it would take to log updates to disk.

Figure 6 shows the ACID memory overhead for a transaction that write-locks a region and updates it, as a function of the region size. The updates need to be propagated to 2 replicas[4]. After subtracting the 35 microsecond latency for lock acquisition, we see that the overhead of updating the replicas is about 25 microseconds for small writes (this is the overhead of the control messages), and about 20 microseconds/KB thereafter (this is the overhead of propagating the actual changes). In contrast, commiting logs to disk will take on the order of milliseconds.

Recovery times: By using a non-blocking commit protocol, we have simplified crash recovery from the perspective of the transaction handler. If the transaction handler detects that the Owner has failed, it needs to only switch to the next replica according to the pre-defined order. In our experiments, we found that the recovery time for an Owner failure was *424 microseconds* on the average, once the failure has been detected. In contrast, rebooting the OS and recovering the data by reading a log from disk typically takes several minutes for commercial DBMSs [Ora].

6 Related Work

6.1 Reliable/Transactional Memory

Reliable Memory: RVM [S+94] provides support for reliable memory using a write-ahead-log for recovery. It uses an in-memory undo log for transactions that abort, and a on-disk redo log. This is quite expensive since every transaction involves one memory copy to the undo log, one synchronous disk write to the redo log, and a asynchronous operation to truncate the log later. Since it uses virtual memory mapping for the recovery and does not support concurrency control, the unit of recovery must be a multiple of the page size.

Transactional Memory: [HM93] describes how to extend a snoopy bus protocol to support transactional load and store operations. This approach supports recovery on individual physical memory locations using a separate *transactional cache*. A problem with this method is that applications have to manipulate critical data through special transactional instructions, and hence have to be rewritten significantly. Second, since recovery is on physical memory regions, all transactions must run in the same address space, and the entire

[4] As before we use the difference between the time taken for the transaction with ACID memory and the time taken for the transaction on regular memory to measure the overhead of recovery.

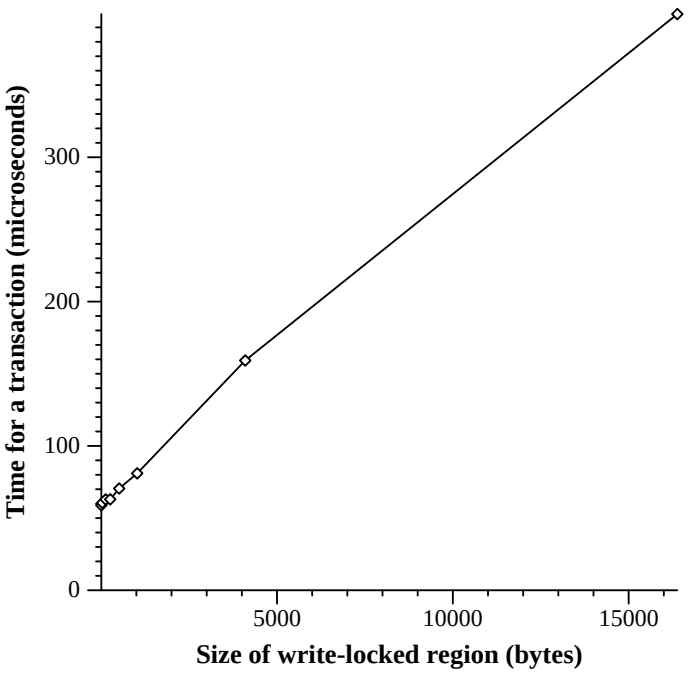

Figure 6: Time taken for a transaction that write-locks a single region. The updates need to be propagated to two replicas

database must fit in memory. Finally, transactions cannot modify more data than the size of the transaction cache.

Shadow Paging: One of the early methods used by DBMSs for recovery was shadow paging [HR83]. This approach is similar to that of RVM, but the DBMS explicitly controls the virtual memory paging to avoid synchronous disk writes. This approach is almost never used today because it does not allow fine-granularity recovery.

Persistent Memory: Recently, the Rio [C+96] project has developed ways making memory non-volatile in the face of system crashes and power failures, by using battery backups and virtual memory protection. They also describe how to use such non-volatile memory to build persistent file caches. This significantly reduces the disk write overhead of recovery. We assume Rio's persistent file cache as a basis for building ACID memory.

6.2 Replication based recovery

Due to the problem with fine-grained recovery, replication has not been used much for recovery in DBMSs. However there is quite a bit of work in the operating systems and architecture world.

Fault Tolerant COMA: There has been some recent work on extending the COMA cache coherence protocols to provide Backward Error Recovery for applications [MGBK96]. However a general problem with COMA techniques is that a data item has no fixed memory location. Hence we need to store the locations of all copies of a locked item, along with information about the different copies as to which is dirty etc. This does not scale for a DBMS application because a DBMS uses fine-grained locking and we will need to store this directory information for every record in the database. Moreover, database transaction processing applications do not exhibit very much locality, and so keeping a copy of an item at the node which read it will not be useful.

User Level Shared Memory: The Stache [RLW94] protocol is similar to the COMA technique but implements the shared memory in software; each processors memory is viewed as a cache for remote data.

7 Conclusions and Future Work

We have designed and developed a ACID memory library for use in DBMSs and other applications that want to update data in a persistent, atomic manner. Our library supports concurrency control and recovery in a combined framework. It is relatively easy to use, since applications can manipulate the directly and need to only acquire locks to make the manipulation persistent and atomic. It is quite light-weight compared to changes that log differences, since the only overhead is a memory copy on a write-lock.

We have found that assuming persistent memory and using replication makes the implementation of the ACID memory quite simple. Experiments with simple micro-benchmarks indicate that although applications can get better performance by explicitly logging their updates, the difference is not very much.

We have extended ACID memory to be highly available by replicating data on multiple machines. This gives us fast transactions, without requiring non-volatile memory. Moreover, it results in extremely low recovery times, without using any specialized hardware or OS support.

An interesting direction for future work will be to port a DBMS such as Postgres to ACID memory and see how much we can simplify the DBMS code, and how well it performs.

A second issue that we have not dealt with in this project is the decision of when and how much to replicate. Increased replication improves read performance, reduces write performance, and improves fault tolerance. It will be interesting to study how these factors can be used to dynamically decide the extent of replication.

Acknowledgments

Discussions with Mike Carey, Joe Hellerstein, and Beau Shekita were very useful in understanding the issues involved in designing an ACID memory system that can be used by DBMSs. Paul Aoki, Marcel Kornacker, and Mehul Shah helped me figure out the details of the non-blocking commit protocol.

References

[C+96] P. Chen et al. The Rio file cache: surviving operating system crashes. In *ASPLOS*, 1996.

[CL99] P. Chen and D. Lowell. Reliability hierarchies. In *HotOS*, 1999.

[GR93] J. Gray and A. Reuter. *Transaction processing: concepts and techniques*. Morgan Kauffman, 1993.

[HM93] M. Herlihy and J. Moss. Transactional memory: Architecutral support for lock-free data structures. In *ISCA*, 1993.

[HR83] Haerder and Reuter. Principles of transaction oriented database recovery. In *Computing Surveys*, 1983.

[MGBK96] C. Morin, A. Gefflaut, M. Banatre, and A. Kermarrec. COMA: an opportunity for building fault-tolerant scalable shared memory multiprocessors. In *ISCA*, 1996.

[MLO86] C. Mohan, B. Lindsay, and Obermarck. Transaction management in the r* distributed database management system. *TODS*, 1986.

[NC97] W. Ng and P. Chen. Integrating reliable memory in databases. In *VLDB*, 1997.

[Ora] Oracle 8i high availability features. http://www.oracle.com/database/features/highavail.html.

[Ram98] R. Ramakrishnan. *Database Management Systems*. McGraw-Hill, 1998.

[RLW94] S. Reinhardt, J. Larus, and D. Wood. Typhoon and Tempest: User-level shared memory. In *ISCA*, 1994.

[S+94] M. Satyanarayanan et al. Lightweight recoverable virtual memory. *TOCS*, 1994.

[Ske81] D. Skeen. Non-blocking commit protocols. In *SIGMOD*, 1981.

[SR86] M. Stonebraker and L. Rowe. The design of Postgres. In *SIGMOD*, 1986.

[Tan] Nonstop sql whitepapers. http://www.tandem.com/prod_des/nstmmppd/nstmmppd.html.