Survey: Transactional Memory and Distributed Databases

Sean Maloney
University of California, Santa Barbara
cczn@cs.ucsb.edu

June 14th, 2012

Abstract

Transactional memory is a hot topic in the research community but it has yet to be fully embraced by industry. If Transactional memory can be proven to show high performance or simplify parallel programming, perhaps it will be adopted faster. In this paper we focus on whether Transactional memory fulfills those criteria specifically for database servers. We discuss previous research that implies hardware and software Transactional Memory will benefit database query times despite the fact that databases and memory are accessed at different granularities. However, only in some situations – results show that when traffic or query complexity is high for a database the chances of data contention increase. This is an issue because data contention incurs large overhead costs in transactional memory.

1 Introduction

If you think about it, database management systems and parallel programming have a couple of similarities when it comes to how they need to handle transactions. In a database management system, multiple clients access the data at the same time and must be allowed to store and read without overwriting or corrupting data. Likewise, in a parallel application, multiple cores accessing the same memory must be able to ensure this same correctness. There are differences though, database transactions usually access a large amount of data in a query while parallel programs are fighting to access only one or two memory locations at a time. This raises the question of whether transactional memory can support the data access patterns of a database query efficiently. Transactional memory is the new kid on the block as a research topic when compared to database transactional correctness which has been studied ad nauseum. Though there has been a good amount of research into the subject, transactional memory has yet to be adopted widely in industry. However, if transactional memory can be shown as an asset to improving the type of workloads that distributed databases handle, the industry may profit from it and help increase adoption. This paper plans to survey where transactional memory research is currently with regards to its use in distributed databases and main memory databases. Why focus on these two types of database systems? Because current trends in industry are toward scalable cloud computing which often requires the use of these two types of database management systems.

In the first section of the paper, we provide the basic concepts behind hardware-based transactional memory in order to provide some background. We specifically focus on hardware transactional memory implementations as most database systems have already implemented some form of software transactional correctness. Then the paper will discuss the architectures of Main Memory Databases and why today’s scalable systems would want to use them. We then discuss the differences between transactions in memory and transactions on a database. This paper then presents some research into how transactional memory can be used to increase the performance of these databases given certain as-
2 Transactional Memory

The need for transactional memory was not driven by a desire to make databases work better, but by the fact that we cannot speed-up processor clock frequencies much further without overheating. In order to continue the processor speed-up that industry has come to expect, we have been adding more cores. To fully utilize multi-core processors, developers must learn to write effective multi-thread applications to take full advantage each core. Thus, if you have many cores writing to the same memory, those reads and writes must be done in a correct way. There are two types of transactional memory, Software-based Transactional Memory (STM) and Hardware Transactional Memory (HTM). Both types intend to relieve some of the parallel programming burden from the developer and both types are designed for fast conflict resolution of one or two memory locations at a time.

![Diagram](image)

Figure 1: One implementation of HTM: notice the transaction control bits to mark what cache line is locked and the write buffer which contains the write history of what gets committed.[1]

Hardware Transactional Memory was an architecture proposed in 1993 [2] that enables transaction atomicity and isolation at the hardware. There are several important functions that Hardware Transactional Memory must be able to perform in order to support transactional correctness: it must identify locations for memory accesses, manage read/write sets of transactions, detect data conflicts, manage register state, and be able to commit/abort transactions [3]. To identify transactional memory accesses, commands such as begin_txn, end_txn and load/store_txn are added to the instruction set and all memory operations contained within must be processed transactionally. Extra e bits can be added to the cache line to indicate it is locked and currently being operated on. Read and write sets can be stored in hardware data caches since all memory reads involve cache lookups and caches are optimized for low-latency memory access [2]. It is important to note that these hardware caches have a finite size which limits the size of each transaction to a small granularity. Detecting conflicts in transactions is done by looking for overlapping memory locations in the read/write sets. This can be done by working with the cache coherence mechanisms which track the write state of memory locations as Shared, Exclusive, or Modified. Conflicting transactions are aborted when a transaction requests access a cache line and that data has already been marked as shared or modified and the aborted transaction is resolved by cancelling it, rescheduling it, or allowing the developer to specify their own conflict resolution. Since registers must also be restored to a previous value if a transaction aborts, the mapping between physical registers and logical registers can be adjusted to point the logical registers from the current physical register to a register with the previous value.

Software Transactional Memory on the other hand is generally implemented as an API with locking mechanisms in a low level programming language such as C/C++. On example is dynamic Software Transactional Memory (DSTM) proposed in [4]. In DSTM, they create a TMO class which can encapsulate all other object classes and protect them so that transactions across multiple processes occur atomically. The TMO class ensures this atomicity and isolation by implements a locking mechanism. Below is an example of what the code would look like
for a simple counter.

```java
Counter counter = new Counter(0);
TMObject tmObject = new TMObject(counter);
Counter counter = (Counter)tmObject
.open(WRITE);
counter.inc(); // increment the counter
```

3 Conventional Relational Databases

Conventional databases such as Oracle or SQL Server already have a form of concurrency control, but are not designed for storing data to memory. They are optimized for transactions within disk storage and sacrifice query response time for increased data capacity. An illustration of this is that these databases primary storage structure is a B-tree. B-trees have a very high fan-out which reduces the I/O operations needed to find data in a block oriented file system. Never less, disk I/O requests can still take a significantly larger number of cycles as compared to memory accesses. Therefore I believe that transactional memory will not increase the performance of conventional relational databases significantly as most of the time is lost in disk seek times anyway. The benefit that transactional memory would have on conventional relational databases is mainly in simplifying the code that it takes to write such a database management system. There is very little research on this topic so these are just assumptions and the paper will not spend any more time looking at them.

4 Main-Memory Databases

Certain activities such as web crawling or information retrieval require accessing and processing large amounts of data in a very short time span. One example of this is a search engine that needs to parse through a complex graph in order to retrieve websites relevant to a query terms before the user gets impatient and browses elsewhere. Main-memory databases can serve this corner of the market well. Conventional relational databases are hard to scale to such a size and speed requirement. And since memory has become cheap, it is easy to scale out many commodity machines with large amounts of memory. MMDBs take advantage of memory's tiny access times and random distribution of data to read and write much faster than disk based storage. Some servers, such as Oracle's Exadata Database Machine have up to 2TB of main memory [5].

Since memory acts so differently from disk storage, Main Memory Databases must manage memory-resident data differently. For example, in concurrency control, locking mechanisms are used to prevent data from being overwritten. Traditional databases store a hash table with entries for the objects that are locked but the objects on disk contain no locks. In a MMDB, locking bits can be allocated within the object stored in memory and the locking additional hash table and look up time can be eliminated [6]. Additionally, since memory accesses are so fast, contention in main memory is considerably lower than traditional databases, so if one query locks a dataset, it is less likely that another query will have to wait for it to finish.

5 Memory and Database Transactions

5.1 Comparison of TM and DB Transactions

There has not been a lot of research done on the workloads of databases and transactional memory however researchers are starting to dip their toes into studying it. But before we look at existing research, we must discuss some of the differences between the design intentions for transactional memory and database transactions.

Databases often execute queries across large sets of data. For example when a user executes the statement:

```sql
update customers
set coupon='$10'
where state = 'CA'
```
Database query schedulers will lock at either the table level or every row affected, then write all of the updated rows to a memory buffer and to a commit log. Let’s assume this statement updates one million rows; these query schedulers are optimized to work with large sets of data like this. The commit log acts to store the history of the entire transaction, no matter how large it is.

Transactional memory, on the other hand, is designed to focus on a few memory accesses per critical section. If you compare the reads and writes in a block of code to the reads and writes in a database query, the former is much generally much smaller than the latter. Many hardware transactional memory implementations plan around this fact, and optimize for access speeds [2] instead of access quantity. The read/write cache is usually small in order to reduce look up times and this imposes a limitation on transaction size and length [7].

5.2 Current Research

The authors of [8] develop a stripped down version of a main memory database and study a simple environment in which Transactions are very short, and come frequently enough to keep the CPU busy. They study the transaction time, overhead time, and cost of conflict resolution for hardware transactional memory and compare it to spinlocks and traditional database locking mechanisms.

Figure 2 shows the results of such experiments. Their studies show that in a low contention environment, transactional memory significantly outperforms the other locking mechanisms in terms of overhead. Transactional Memory overhead times are constant when more objects are accessed in comparison to the other methods which have linear overhead times. After that, they ran a test to see the breakdown of where time was going.

However, it seems as if transactional memory is not the best at resolving conflicts when there is contention. Figure 3 shows the overall query time and, as we can see the conflict cost grows significantly to the extent where it is now slower than spinlocks. This demonstrates the limits of the transactional memory architecture when we assume that larger database queries will take longer to run and therefore have more contention.

In a separate paper, the authors of [9] combine the operations of memory and database transactions in a unified model. It is essentially an implementation of software transactional memory. They test their implementation on a PostgreSQL database which is more of a traditional database architecture. Their results are consistent with the results of hardware transactional memory tests in [8]. Figure 4 shows for high contention, the software transactional model they
propose outperforms the database alone until the number of concurrent threads reaches about 10. At that point, their unified model underperforms and adds additional overhead to the transactions. For low contention, however, the software-based transactional memory consistently outperforms the database alone. Since the disk access speeds are much longer in a traditional database system, there is more chance for contention so this may not be a viable. This test was run on a server with 8 dual-core processors, so it could have been more revealing to show results for more than 16 threads when the processors have to do more scheduling.

Both of these papers are in agreement, transactional memory incurs high overhead when executing database queries in high contention, whether it is software transactional memory or hardware transactional memory. The read/write caches are just not large enough to store data about many different database-size transactions. There is good news – one solution has been proposed to overcome this issue, and it is called Unbounded Transactional Memory [7]. In previous TM architectures, the cache stores write state and uses the cache coherency protocol to detect conflicting transactions. Unlike previous hardware systems, Unbounded Transactional Memory allows overflow from the cache into main memory, allowing the transaction to grow beyond the size limitation of the cache. The author proposes an “overflow handler” that rests between the hardware write-set cache and the main memory that is essentially a hash map pointing to memory locations with the write set overflow. The proposed architecture is described by Figure 5.

Unfortunately the author only tests his system with the SPECjvm98 benchmarks and not with a real database load, but the speed-up results from the SPECjvm98 tool are promising.
6 Further Research

There is a lot of research still to be done regarding databases and transactional memory. For example, the recent proliferation of solid state drives has increased data read and write speeds. Can a transactional memory model be applied to solid state drives since they use a form of NAND-based flash memory and would this have improved performance implications for traditional disk-based databases. Another area of research is to determine whether hardware transactional memory would be beneficial for traditional database systems if the DBMSs increased their memory buffer size and kept more of the database in memory.

CloudSuite[10] is a testing suite that employs programs that you would normally see used in cloud-based applications to benchmark processor performance. The paper discovered several improvements that can be made to processors regarding that workload. It would be interesting to see if the same cloud based programs could be used to benchmark memory accesses on transactional memory versus regular memory and see if any memory improvements can be discovered.

As previously mentioned, one of the issues with hardware transactional memory is that it is focused on fine grained memory accesses. This results in overhead when there is high contention or large data sets accessed. Further research can be focused on designing a new architecture for hardware transactional memory that has database usage in mind. This new design must be able to handle concurrency control at the memory level for very large sets of data. We have seen one proposal [7] for an architecture that addresses this problem, but it was not tested explicitly with a database management system. So while the results appear promising, additional discoveries can be made in this area.

7 Conclusion

Based on the limited amount of research for existing Transactional Memory implementations, it appears that there are two factors that affect how HTM and STM will benefit some database query times. One is high contention systems where there are a lot of transactions. And the other is the query size, most current transactional memory systems are limited by their transactional speculative state hardware buffer. Further research is being done into architectures that address this limitation. In the mean time, System architects will need to determine how much data contention and how large what query granularity will their application has on a case-by-case basis. If the overhead incurred by Transactional Memory is allowable or new TM architectures are built then adopting of Transactional memory will increase. The author of this survey estimates that, in most cloud-based services, contention will be low which makes Transactional Memory more appealing especially when employing a main memory database. This is based on how data is spread across servers in cloud-based architectures and on the assumption that websites are read more than they are updated. But for some tasks, like nightly data warehousing, Transactional Memory may not be the optimal solution since large, complex transformation queries are executed. Unfortunately, Hardware Transactional Memory has a few kinks to work out so it may still be several years before we see a commodity server with transactional memory in action.

References


