On Transactional Memory, Spinlocks, and Database Transactions

Khai Q. Tran
University of Wisconsin
Madison
1210 W. Dayton
Madison, WI, 53706
khaitran@cs.wisc.edu

Spyros Blanas
University of Wisconsin
Madison
1210 W. Dayton
Madison, WI, 53706
sblanas@cs.wisc.edu

Jeffrey F. Naughton
University of Wisconsin
Madison
1210 W. Dayton
Madison, WI, 53706
naughton@cs.wisc.edu

ABSTRACT

Currently, hardware trends include a move toward multi-core processors, cheap and persistent variants of memory, and even sophisticated hardware support for mutual exclusion in the form of transactional memory. These trends, coupled with a growing desire for extremely high performance on short database transactions, raise the question of whether the hardware primitives developed for mutual exclusion can be exploited to run database transactions. In this paper, we present a preliminary exploration of this question. We conduct a set of experiments on both a hardware prototype and a simulator of a multi-core processor with Transactional Memory (TM.) Our results show that TM is attractive under low contention workloads, while spinlocks can tolerate high contention workloads well, and that in some cases these approaches can beat a simple implementation of a traditional database lock manager by an order of magnitude.

1. INTRODUCTION

To the best of our knowledge, there has been no published consideration of using hardware synchronization primitives such as transactional memory to directly support concurrency control for database transactions. One reason for this is perhaps that these hardware primitives have little or nothing to offer in the context of arbitrarily complex, heavyweight database transactions running over disk-resident data. However, the convergence of a number of important trends lead us to believe that this issue should be revisited.

The first such trend is the demand for special-purpose systems targeted at very fast processing of very simple transactions. The DBMS industry has already begun reacting to this demand — there are startups targeting this market (including VoltDB and other, stealth-mode companies.) An important characteristic of these workloads is that they consist of very short transactions that contain no I/O or think time. For such workloads, the overhead due to traditional concurrency control methods will be by far the major part of the transaction execution path [5]. On a uniprocessor, there is no reason to incur this overhead [4], since there is no wait or think time in one transaction to be overlapped with the execution of another. In other words, on a uniprocessor, the best concurrency control approach for such workloads is likely to be to truly serialize the transactions.

However, true serialization collides with the second trend, that is, the growing dominance of multicore processors. Clearly, on a multicore processor, true serialization of transactions is not an attractive option, since true serialization will waste most of the power of the multiprocessor. If we give up on true serialized execution and run the transactions concurrently, we once again face the need to implement some kind of concurrency control. The traditional locking approach may be inefficient for short-lived transactions due to the high overhead of acquiring and releasing locks, and the cost of resolving contention, including thread scheduling and synchronization.

These trends are converging to place extreme pressure on techniques enforcing the isolation requirements of database transactions. Accordingly, in this paper we investigate using hardware primitives directly as a concurrency control mechanism for workloads consisting of very short transactions. We evaluate and compare the performance of three Concurrency Control (CC) mechanisms: Hardware Transactional Memory (HTM), test-and-set spinlocks, and a simple implementation of a database lock manager. We then use both an early hardware prototype implementation of transactional memory and a simulator supporting TM to conduct a set of experiments on each CC approach under different workload conditions.

We emphasize what we are *not* studying in this paper. We are not studying the question "Given traditional DBMS architectures and workloads, can we improve performance by replacing the lock manager with concurrency control based upon hardware primitives?" The answer to that question is almost certainly "no," since current architectures present substantial overhead due to their generality and the legacy assumptions in their design. Rather, we are studying the question "what will happen with respect to the performance of concurrency control if we build a stripped-down system optimized for the execution of short transactions over main-memory data?"

Building such systems will require advances including rethinking storage structures (slotted pages of variable-length records are probably not optimal) and transaction logging. While it is too early to be certain, it is possible that the emergence of very fast persistent memory such as phase-change memory [2], perhaps in addition to new software techniques like "k-safety" [15], will eventually lead to extremely efficient mechanisms for supporting transaction persistence. We do not pretend to address all of these issues in this paper; rather, our intent is to shed light on what will be a crucial component in any such system, namely, very light-weight concurrency control for short-lived transactions on multi-core processors.

We also wish to acknowledge that at this point it is not clear when, if ever, transactional memories will become widely available. Furthermore, due to the extremely primitive nature of the hardware transactional memory prototype to which we had access, the workloads we explore in this paper are greatly oversimplified even when compared to the short transactions that are the target of extreme transaction processing. Hence, again, we are not making claims about the practicality of our work in any near-term sense. Rather, we hope to raise the point that from the database transaction processing perspective, a move by the hardware community away from the development of hardware transactional memory would be unfortunate, as a sufficiently powerful future implementation of HTM could be of great benefit to database transactional processing. To express this differently, one sometimes hears computer architects wondering "what are we going to do with all the transistors we will have on future chips?" In this paper we would like to at least suggest "consider hardware support for transaction isolation."

The main contributions of the paper are as follows:

- We develop timing models for estimating the execution time of a transaction using TM and spinlocks as CC mechanisms. The accuracy of these models is verified through a set of experiments.

- Through both the analytical model and the experimental results, we show that TM and spinlocks are promising CC approaches for workloads consisting of a high volume of very short transactions. TM delivers high performance under low contention. Perhaps surprisingly, although spinlocks do not differentiate between read and write access, and employ busy-waiting instead of sleeping on a queue, spinlocks appear to be more attractive than TM or database locks under higher contention. We show that these hardware primitives can beat the performance of our simple lock manager by an order of magnitude.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the analytical model for each CC approach. Section 4 presents experiments and results. Finally, Section 5 concludes the paper.

2. RELATED WORK

The idea of running short main memory transactions serially was first proposed by Garcia-Molina and Salem in [4]. In that paper, the authors briefly mentioned that even if all transactions are short, some form of concurrency control is needed for multiprocessor systems.

Recently, that idea of serial execution for short-lived transactions has been revived in the H-store system [17]. Stonebraker et al. argued that it was the end of "one size fits all" period for DBMS architecture [16], and that for each specific type of application, we need a specialized database system. Workloads for the H-Store system need to be highly partitioned so that each transaction usually touches only one partition of the database. For that kind of workload, most transactions can be executed serially without any concurrency control at all. In case that transactions need to access multiple partitions, they employ a simple type of speculative concurrency control [10]. However, the performance of this approach degrades in the presence of non-partitioned and abort-frequent workloads because an abort of a multi-partition transaction causes cascading aborts of following speculatively executed transactions.

In recent work, Pandis et al. showed that a conventional lock manger severely limits the performance and scalability of database systems on multicores [14]. By using *thread-to-data* instead of *thread-to-transaction* assignments, they can reduce contention among threads. However, since this approach is still based on database locks, it does not avoid the problem of high overhead in acquiring and releasing locks.

In other recent work, Johnson et al. studied the trade-off between spinning and blocking synchronization [7]. They pointed out that while the blocking approach is widely used in database systems, it negatively affects system performance in multicore environments due to the overhead of coordination with the operating system for scheduling threads. The spinning approach, on the other hand, does not incur that cost and offers a much lower overhead, but system performance degrades under high load since the spinning threads do not release the CPU even if they do not make any progress. This work did not consider the option of using spinning as an alternative approach for transaction concurrency control.

There has been also a great deal of work exploring methods of combining hardware and software facilities to ensure mutual exclusion and to coordinate concurrent access to shared data structures [8]. However, to the best of our knowledge there has been no published work considering using HTM and spinlocks in the context of database transactions.

3. BACKGROUND

In this section, we explore HTM and spinlocks as CC approaches for database transactions. For each approach, we give a simple timing model to provide insight into their expected performance. The model will be used to help explain the results of our experiments on a hardware prototype and a software simulator in Section 4. We also describe the implementation of our simple lock manager.

3.1 Workloads and timing model

The workloads we study in this paper have following characteristics:

- Transactions are very short and do not contain I/Os or think time.

- The active portion of the database is stored in main memory.

- Transactions come to the system at a rate high enough to keep the CPUs busy.

To analyze the performance of a given CC method, we separate the time spent for executing a transaction, called *total time*, into three different parts:

- *Transaction time*: the time required to execute the transaction in isolation with no CC. This time is constant with respect to any CC mechanism.

- *Overhead*: the time spent running the transaction in isolation with the CC method minus the transaction time. This fraction depends on both the transaction and the CC method.

- *Conflict resolution cost* (or *conflict cost* for brevity): the time to run a transaction with the CC method in the presence of other, concurrent transactions, minus the transaction time and overhead. This fraction depends not only on the transaction and the concurrency control method, but also on the workload.

3.2 Transactional Memory

The concept of TM was first introduced by M. Herlihy and J. E. B. Moss [6]. The idea of TM is to let pieces of code run atomically and in isolation on each processor in a multiprocessor system. These pieces of code appear to be executed atomically and serially with respect to the memory state. TM can be implemented either in software or in hardware. In this paper, we only consider hardware Transactional Memory (HTM) because of the low overhead it offers.

HTM guarantees the atomicity and isolation of its transactions by keeping track of their read sets and write sets, recording old values of the write sets, and using cache coherence protocols to detect possible conflicts among transactions. The initial machine state and values of objects that are modified by a transaction are logged so that if the transaction fails, the machine can roll back to the initial state. While the transaction is running, the cache coherence protocol automatically checks if there is any overlap between the write set of the transaction and the write sets or read sets of other concurrent transactions or between the read set of the transaction and the write sets of others. If there is such an overlap, the machine will abort one of two conflicting transactions.

From a DBMS point of view, the idea of HTM is very similar to that of optimistic concurrency control (OCC) [11]. However, they differ in the way of implementing the idea: in HTM, conflicts are detected automatically by hardware, while OCC is implemented in software.

3.2.1 Timing model

Suppose t_0, Δ_{tm}, t_{tm} are the transaction time, the overhead, and the total time of using TM to run a transaction, respectively.

Under no contention, we have:

$$t_{tm} = t_0 + \Delta_{tm}$$

Under contention, suppose p_{tm} is the probability that a transaction conflicts with others. Suppose that each time a transaction gets aborted due to a conflict, the conflict occurs right before the completion point. Thus, if the transaction is successfully executed after k restarts, the total execution time is $(k+1)(t_0+\Delta_{tm})$, and the probability for this to happen is $p_{tm}^k(1-p_{tm})$. On average, the total time is given by:

$$\begin{aligned} t_{tm} &= \sum_{k=0}^{\infty}(k+1)(t_0+\Delta_{tm})p_{tm}^k(1-p_{tm}) \\ &= (1-p_{tm})(t_0+\Delta_{tm})\sum_{k=0}^{\infty}(k+1)p_{tm}^k \end{aligned}$$

by reduction, we have

$$\sum_{k=0}^{\infty}(k+1)p_{tm}^k = \frac{1}{(1-p_{tm})^2}$$

therefore

$$t_{tm} = \frac{t_0+\Delta_{tm}}{1-p_{tm}} \tag{1}$$

or

$$t_{tm} = t_0 + \Delta_{tm} + t_{cf\_tm} \tag{2}$$

where

$$t_{cf\_tm} = \frac{p_{tm}(t_0+\Delta_{tm})}{1-p_{tm}} \tag{3}$$

is the conflict cost for TM.

We see that when $p_{tm} \rightarrow 1, t_{cf\_tm} \rightarrow \infty$, or in other words, TM performs very badly under high contention workloads because it keeps restarting. We estimate the conflict probability as follows.

Suppose n_t, n_r, n_w, and DB_size are the number of threads, the size of the read set, the size of the write set, and the size of the database, respectively. A transaction conflicts with others if at least one object of its read set belongs to the write set of any other transaction, or at least one object of its write set belongs to the read set or the write set of any other transaction.

So at a given time, we need to know how many read objects and write objects are currently in the read set and the write set of a transaction. Suppose that, on average, those numbers are $\alpha_1 n_r$ and $\alpha_2 n_w$, where α_1 and α_2 are two coefficients that depend on how a transaction accesses its objects. If transactions access objects uniformly, and gradually from beginning to the end, we have $\alpha_1 = \alpha_2 = 0.5$.

Therefore, roughly, the conflict probability can be estimated by:

$$p_{tm} = 1-(1-\frac{\alpha_2 n_w(n_t-1)}{DB\_size})^{n_r}(1-\frac{(\alpha_1 n_r+\alpha_2 n_w)(n_t-1)}{DB\_size})^{n_w}$$

If DB_size is much bigger than $(\alpha_1 n_r + \alpha_2 n_w)(n_t-1)$, we can approximate p by:

$$p_{tm} \approx 1-(1-\frac{\alpha_2 n_w(n_t-1)n_r}{DB\_size})(1-\frac{(\alpha_1 n_r+\alpha_2 n_w)(n_t-1)n_w}{DB\_size})$$

using approximation again:

$$p_{tm} \approx 1-(1-\frac{\alpha_2 n_r n_w(n_t-1)+n_w(\alpha_1 n_r+\alpha_2 n_w)(n_t-1)}{DB\_size})$$

or

$$p_{tm} \approx \frac{(n_t-1)n_w(\alpha_2 n_w+(\alpha_1+\alpha_2)n_r)}{DB\_size} \tag{4}$$

If we approximate $\alpha_1 = \alpha_2 = 0.5$:

$$p_{tm} \approx \frac{(n_t-1)n_w(n_w+2n_r)}{2DB\_size} \tag{5}$$

3.3 Spinlocks

The idea of using spinlocks to isolate concurrent transactions is straight-forward. We use a spinlock to protect each database object. Before accessing an object, transactions are required to hold the corresponding spinlock for that object. Locks are acquired and released following a 2-phase protocol.

Deadlock detection or prevention is an interesting problem for this approach. Since locks we hold here are physical locks, there is no list of which transaction is currently "spinning" on which lock. Therefore, there is no way to build the "waits-for" graph, and we cannot use the deadlock resolution used by most lock managers. In this paper, we consider two approaches for this problem.

In the first approach, we prevent deadlock by sorting all objects before acquiring corresponding locks for them. By sorting objects on their ids, at any time, active transactions are ordered so that circular wait does not occur. In this approach, when a transaction is trying to acquire a lock, if the lock is held by another transaction, the thread running the transaction simply does nothing but keeps trying until it succeeds. To employ this approach, we need to know all the objects a transaction will access before it begins execution.

In the second approach, we detect possible deadlocks by using a timeout mechanism. Each time a transaction spins to acquire a lock, we count the number of times it fails. If this number exceeds a threshold, we assume that deadlock has occurred, and we abort and restart the transaction.

The second approach differs from the first only in its deadlock handling policy. Since we did not need to refer to this aspect of its performance to analyze our experiments, due to space limitations, we only present a timing model for the first approach.

3.3.1 Timing model

In our timing model, the total time of using spinlocks to run a transaction can be represented by:

$$t_{spl} = t_0 + t_{overhead\_spl} + t_{cf\_spl} \tag{6}$$

$t_{overhead\_spl}$ includes the sorting time and the time for acquiring all spinlocks. Suppose t_{sort} is the for sorting all objects, and Δ_{spl} is the overhead of spinning a lock. We have:

$$t_{overhead\_spl} = t_{sort} + (n_r + n_w)\Delta_{spl} \tag{7}$$

t_{cf_spl} is the time spent for waiting other transactions if conflicts happen. Since transactions are ordered, at any given time, a transaction may need to wait for from 0 to $n_t - 1$ other transactions, where n_t is the number of threads defined in Section 3.2. Suppose that on average, the transaction needs to wait for $0.5(n_t - 1)$ other ones, and that p_{spl} is the probability that a transaction conflicts with another, the waiting time is approximated by:

$$t_{cf\_spl} \approx 0.5(n_t - 1)(t_0 + t_{overhead\_spl})p_{spl} \tag{8}$$

p_{spl} can be estimated in a way similar to the estimation of p_{cf_tm} in Section 3.2, but note that here we do not differentiate between read objects and write objects. To estimate the average number of locks a transaction holds through its timelife, we assume that transactions acquire locks in a fashion shown in Figure 1. The average percentage of the total locks a transaction is holding at a time is approximated by:

$$\beta \approx \frac{t_0 + 0.5(n_r + n_w)\Delta_{spl}}{t_{sort} + (n_r + n_w)\Delta_{spl} + t_0} \tag{9}$$

and p_{spl} is approximated by:

$$p_{spl} \approx \beta \frac{(n_r + n_w)^2}{DB\_size} \tag{10}$$

Combining equations 6, 8, and 10, we have

$$t_{cf\_spl} \approx \beta \frac{(n_t - 1)(n_r + n_w)^2}{2DB\_size}(t_0 + t_{overhead\_spl}) \tag{11}$$

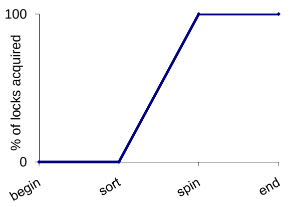


Figure 1: Percentage of locks a transaction holds through its time life

$$t_{spl} \approx (1 + \beta \frac{(n_t - 1)(n_r + n_w)^2}{2DB\_size})(t_0 + t_{overhead\_spl}) \tag{12}$$

3.4 Database locks

We implemented a simple version of database lock manager based on the lock manager in the Shore-MT storage engine [9]. Although our lock manager is much simpler than a real lock manager, we believe it serves our purpose, which is to provide a lower bound on the performance of database locking.

In our lock manager, for simplicity, we focus only on locking at the record-level since it is the only type of objects we consider in our experiments. The basic data structures include a lock table hashed on record ids, and a set of pending queues, one for each thread. Each lock contains the lock's mode, which can be read, write, or free, a latch, and a list of lock requests. Each entry of the request list consists of the transaction id, the lock mode that the transaction wants to acquire, and the status of the request, which is either pending or granted. A pending queue is a queue of pending transactions that are blocked. It contains a latch and a list of pending transactions. Each entry of this list consists of the transaction id, index of the object that the transaction has failed acquiring the lock for, and the status of transaction, which is either ready or blocked. When a thread accesses a pending queue, it needs to latch and unlatch the queue to synchronize with accesses from other threads.

In this implementation, for simplicity, we avoid deadlock by pre-sorting all records to be accessed by a transaction. Note that in our performance results, we omit the time for this sorting. Hence, we are overestimating the performance of the lock manager, because we are in essence assuming deadlock detection/avoidance is free. Again, this is reasonable since our goal is to provide a lower bound on the performance of a typical lock manager, not to accurately predict its absolute performance.

When a transaction requests a lock, the lock manager first looks it up in the lock table. If the lock is not found, the manager gets a new lock from the lock free list and inserts the lock into the lock table. Once the lock is located, the manager latches it, checks the compatibility between the request and the lock's current mode, and appends the request to the request list of the lock. If the request is incompatible with the lock's current mode, the transaction is blocked and put into the appropriate pending queue.

When the transaction completes, it releases locks one by one in reverse order. To release a lock, the lock manager first latches the lock and removes the corresponding request from the request list. If the list is empty, the lock is unlinked from the lock table and then returned to the lock free list. Otherwise, the lock manager goes through the list to decide the new lock mode and to find any pending requests which may be now granted. If there are such requests, a list of ready transactions are returned so that they are waken up later.

Because our experiments show that the lock manager is far from competitive with the other two approaches, due to space constraints we omit a timing model here.

3.5 Physical vs. logical issues

We note that here is a semantic mismatch between database locks and transactional memory: while the database lock manager locks logical objects, TM "tracks" references to physical addresses, not objects. This means that for correctness, when using TM to manage transactions over objects larger than a single memory word, one needs to ensure that the transactions that read or write anywhere within an object also read or write a specified "dummy" word somewhere in the object. (If we do not do this, it is possible that transactions could simultaneously write different parts of a large field of an object without conflicting.)

4. EXPERIMENTS AND RESULTS

4.1 Experiment settings

The database in our experiments consists of a single table. Each row of a table is a record consisting of *id*, *key*, and *balance* fields, and the size of each record is 64B.

In our experiments, we fixed the total number of records to be 1000. Of course, this is a tiny database; however, the critical factor for transactional memory is the size of the transactions, not the size of the database. This is because transactional memory saves and intersects read sets and write sets, and of course never interacts with data that is not accessed. In fact, the only way in which our performance depends upon database size is that a smaller database is more challenging than a larger one, since for the same randomized workload the conflict rates are higher over smaller data sets. Finally, our reason for assuming only 1000 records is very simple — the simulator we used in some of our experiments was impossibly slow on larger data sets.

The workload is a collection of transactions specified by a transaction id, a set of read object ids, and a set of write object ids. The reads read a single one-word field in the record, whereas the writes write a singe one-word field in the record.

In our experiments, the workload is created in advance and distributed equally into a set of transaction queues, one reserved for each thread. At the runtime, each thread is bound to a core of the processor. When the workload is executed, for the TM and spinlock approaches, each thread simply picks up transactions one by one from its transaction queue, and executes them until completion. For the database lock approach, each thread first looks up its corresponding pending queue (see Section 3.4) to see if there is any ready transaction. If yes, it removes the first ready transaction from the pending queue and resumes that transaction. Otherwise, it gets a new transaction from its transaction queue. The execution terminates when there are no more transaction in both the pending queue and the transaction queue.

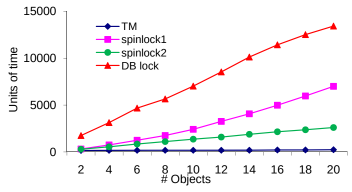
To control the level of contention of the workload, we use another parameter, the "partitioned probability" p_{par}, which is the probability that a transaction only touches records belonging to a portion of the database that corresponds to the thread executing the transaction. Otherwise, records are chosen randomly from the entire database. When p_{par} is equal to 1.0, the workload is completely partitioned, and conflicts do not occur.

At first, we planned to run all experiments on a hardware prototype supporting TM, called TM0 [1]. However, TM0, being the first prototype implementation of HTM, is rather limited. In particular, it implements what is known as Best-Effort HTM. In Best-Effort HTM, the allowable size of read-sets and write-sets is bounded, and forward-progress is not guaranteed, i.e., transactions may be restarted forever. In addition, in the TM0 transactional memory implementation, the transactions are very brittle in that they can be restarted by TLB misses, function calls, cache evictions, and overflow (see [3] for more experience of working with the prototype.) We emphasize that these are properties of the specific early implementation of HTM in TM0, not general properties of HTM. While these limitations made experimenting with TM frustrating and limited the set of experiments we could perform, TM0 still demonstrates the extremely low overhead of HTM and hints at the performance that might be obtainable from future, more full-function implementations of HTM.

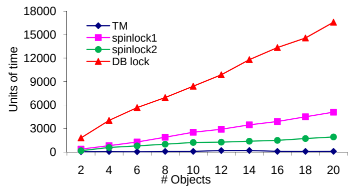
Because of the limitations of TM0, we also ran experiments on a simulator of HTM. We chose GEMS, a general execution-driven multiprocessor simulator [12], because it is widely used in the hardware community (there are now more than 80 published works using this simulator.) The TM version supported in GEMS is LogTM, i.e., log-based Transactional Memory. More details of this implementation are given in [13].

In our experiments, we used both TM0 and GEMS for experiments with low contention workloads with small read sets and write sets, and we compared results from the hardware and the simulator. Under high contention workloads with big read sets and write sets, due to the limitations of TM0, we could only use the simulator.

4.2 Experiment 1: Overhead



(a) TM0



(b) LogTM

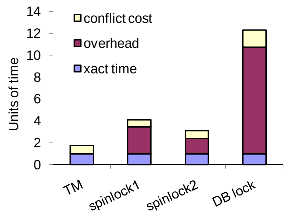
Figure 2: Overhead of CC approaches

In this experiment, we examine the overhead of each approach in both the hardware machine and the simulator. To do this, we first created a workload with no contention by setting p_{par} equal to 1.0. We then ran the workload with each CC approach as well as with no CC, and computed the differences. We varied the number of objects accessed by transactions to see how the overhead of each approach scales. The results are presented in Figure 2.

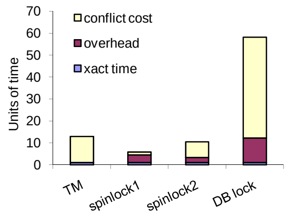
The figure shows that the overhead of TM is negligible compared to spinlocks and database locks. The overhead of both spin locks and database locks scale nearly linearly with the number of objects both on the real hardware and the simulator.

We implemented two versions of the spinlock approach, described in Section 3.3: one with deadlock prevention by sorting, called *spinlock1*, and the other with deadlock detection by timeout, called *spinlock2*. In Figure 2, the spinlock2 line corresponds to the cost of acquiring spinlocks, and the difference between spinlock1 and spinlock2 corresponds to the sorting time.

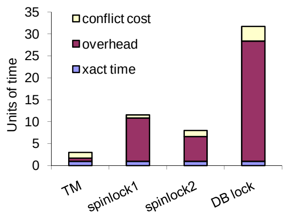
4.3 Experiment 2: Time Breakdown



(a) 4 reads, 4 writes on LogTM



(b) 20 reads, 20 writes on LogTM



(c) 4 reads, 4 writes on Rock

Figure 3: Time breakdown

In this experiment, we measured the transaction time, overhead, and conflict resolution cost for each CC approach under different workloads. To measure these time components for a given CC approach, we first created a no contention workload by setting p_{par} equal to 1.0. Then we ran it with the CC approach and with no CC to obtain the transaction time and the overhead, as described from Section 4.2. Next, we created a full contention workload by setting p_{par} equal to 0.0, and ran it. The difference between times running these two workloads is the conflict cost of the given CC approach. each CC approach as well as no CC, similar to Section 4.2.

We executed each CC approach under two workload conditions: a low conflict rate (where each transaction reads and writes four objects) and a high conflict rate (where each transaction reads and writes 20 objects.) In the former case, we ran CC approaches on both environment while in the later case, we could only use the simulator because TM0 is not stable at such high conflict rates. We used 8 threads in all experiments in this section.

Figure 3 shows the results with times normalized by transaction time. Under the low contention workload, the overhead plays a very important role, as illustrated in Figure 3(a) and 3(c). Since the TM offers very low overhead, it is the best approach.

However, under high contention workloads, the conflict costs of TM, spinlock2 and database lock dominate the other components. This means that these approaches are vulnerable to conflicts. However, it turns out that under high contention, spinlock1 appears to be the best approach since it handles contention very well. These results can be explained as follows.

First, Equation 3 shows that when p_{tm_cf} approaches 1, the conflict cost of TM goes to infinity. In our situation, this probability is close to 1, so that the conflict cost is about ten times bigger than the transaction time. Since spinlock2 uses failure-retry to resolve suspected deadlocks, its conflict cost is also high under high contention.

With the database lock approach, if the contention is high, and transactions keep coming to the system at a high arrival rate, there is a reciprocal relationship between the conflict rate and the number of blocked transactions. When the conflict rate gets higher, the chance for a transaction to be blocked is also bigger. When a transaction is blocked and de-scheduled, it does not release locks that it already holds. Therefore, if transactions keep coming to the system at a high arrival rate, more transactions will be blocked, which will increase the number of locks currently held and thus, increase the conflict rate. This reciprocal effect makes the request list and transaction queues longer, causing the searching cost to be more expensive. Higher conflict also leads to more frequent accesses to shared data structures, which increases the synchronization cost. Therefore, the high contention workload leads to poor performance in the database lock approach, similar to results shown in [14].

With spinlock1, from Equation 8, we have:

$$\frac{t_{cf\_spl}}{t_0 + t_{overhead\_spl}} \approx 0.5(n_t - 1)p_{spl}$$

Since transaction time and spinning time are small compared to sorting time, Equation 9 shows that β is relatively small, which makes p_{spl} not too big. Therefore, the conflict cost of spinlock1 is not too high even under high contention.

Impacts of transaction time: Observe that given fixed sizes of read sets and write sets, transaction time and the overhead of any CC approach are independent of each other, and that the conflict cost depends on both transaction time and the overhead. We analyze the normalized total time of a given CC approach as follows:

$$\frac{t_{total}}{t_0} = \frac{(t_0 + t_{overhead} + t_{cf})}{(t_0 + t_{overhead})}\frac{(t_0 + t_{overhead})}{t_0}$$

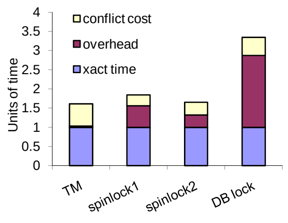
or

$$\frac{t_{total}}{t_0} = (1 + \frac{t_{cf}}{t_0 + t_{overhead}})(1 + \frac{t_{overhead}}{t_0}) \qquad (13)$$

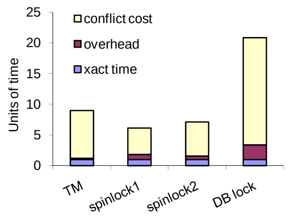
Thus, when we increase the transaction time, the second factor of the equation decreases, making the impact of the overhead less important.

In our experiments, we increased transaction time by adding some "dummy" computation inside transactions. Figure 4 are results of experiments similar to those for Figure 3 but with transactions extended to be five time longer. Figure 4(a) shows that the differences among different approaches under the low contention workload now are not as clear as those in Figure 3(a), especially between TM and spinlock. Figure 4(b) shows that the ratio of the conflict cost to the sum of transaction time and the overhead for TM, spinlock2 and database lock are not changed much, but that ratio for spinlock1 increases.

We can explain the result for the conflict cost of spinlock1 as follows. From Equation 9, we see that when we increase the transaction time t_0, β is also increased. That makes p_{spl}, and, thus, the conflict cost also increase. However, Equation 8 shows that even at the extreme case where $p_{spl} \approx 1$, the conflict cost is still bounded by $0.5(n_t - 1)(t_0 + t_{overhead\_spl})$.



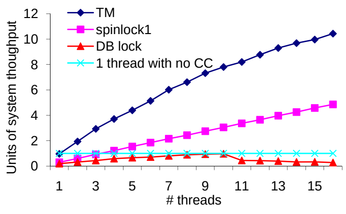
(a) 4 reads, 4 writes on LogTM



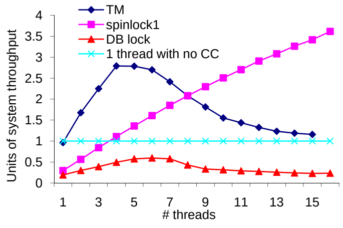
(b) 20 reads, 20 writes on LogTM

Figure 4: Time breakdown with transactions five times longer than those in Figure 3

4.4 Experiment 3: Scalability



(a) 95% partitioned workload



(b) 0% partitioned workload

Figure 5: Scalability on LogTM with 10 reads and 10 writes

In this experiments, we examine how well each approach scales as a function of the number of threads under different conditions. In the experiment, we created workloads with 10 reads and 10 writes for each transaction, and we use p_{par} to control the contention level. Specifically, we set p_{par} to 0.0 for the high contention workload and to 0.95 for the low contention workload. The results are presented in Figure 5 with throughput normalized by the throughput of the base solution where we use only one thread to run transactions serially without any CC.

The graphs show that under low contention workloads, TM scales fairly well with the number of threads. However, under high contention, it only scales up to 4 threads. After that, it even starts going down gradually. These results can be explained by our analytical model as follows.

We approximate the throughput of the TM approach by using Equations 1 and 5:

$$perf_{tm}(n_t) = \frac{n_t}{t_{tm}} \approx \frac{2n_t(1 - \frac{n_w(n_w+2n_r)}{2DB\_size}(n_t - 1))}{t_0 + \Delta_{tm}} \quad (14)$$

Therefore, under low contention, Equation 14 predicts that the throughput of the TM approach will have the shape of the left part of an up-side-down parabola. This matches the results shown in Figure 5.

Moreover, by taking the derivative of the right part of equation 14, we can predict the maximum point:

$$n_{tmax} \approx \frac{DB\_size}{n_w(n_w + 2n_r)} + 0.5 \quad (15)$$

With $n_r = n_w = 10$ and $DB\_size = 100$, we have $n_{tmax} = 3.83$, which is very close to the maximum point of 4 shown in the graph.

With spinlock1, using Equations 6 and 8, the system throughput is approximated by:

$$perf_{spl}(n_t) \approx \frac{n_t}{t_{tm}} = \frac{n_t}{(t_0 + t_{overhead\_spl})(1 + 0.5p_{spl}(n_t - 1))} \quad (16)$$

Thus, the performance of the spinlock will have the shape of a hyperbola. When p_{spl} is small enough, the curve looks almost like a straight line. Figure 5(a) demonstrates this prediction. In Figure 5(b), although contention increases with the number of threads, spinlock1 still scales nearly linearly. That is because the small transaction time makes the value of β also small, which leads to a small value of p_{spl}, as explained in the previous section.

With database locks, the throughput only scales up to a certain number of threads, where the reciprocal effect happens, as explained in Section 4.3. After that point, the throughput starts to go down. Comparing Figure 5(a) to Figure 5(b), we see that under low contention, the approach scales better than under high contention. Figure 5 also shows that the maximum performance of database locks in both situations is still smaller than the base approach of using one thread with no CC. This suggests that for workloads with very short transactions, on our hardware and simulator, it is better to run serially on only one core than to employ a database lock manager.

With spinlock2, we first expected that its trend would be similar to that of TM. However, when we ran the experiment, we did not get the peak. We speculated that the contention was not high enough, so we repeated the experiment with 20 reads and 20 writes in each transaction. With that contention level, we observed the maximum of throughput at n_t of 5. The results are shown in Figure 6.

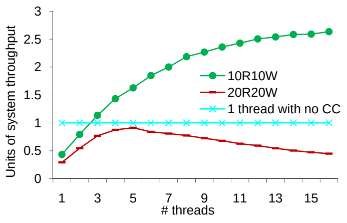
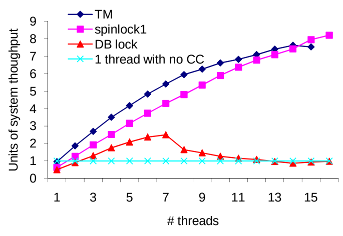


Figure 6: Scalability for spinlock2 on LogTM with 0% partitioned workload

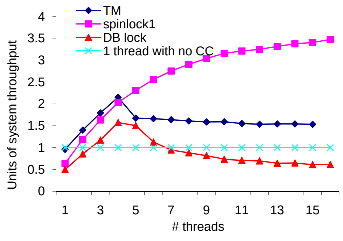
Impacts of transaction time: Similar to Section 4.3, we study the impact of transaction time by repeating the experiments after adding extra computation inside transactions to make them about five times longer. The experimental results are presented in Figure 7.

Under low contention, the throughput lines of the three approaches are brought close together since the effect of the overhead of spinlocks and database locks is lessened. The performance of spinlock1 is very close to that of TM, and the trends of spinlock1 and TM lines shows that spinlock1 starts beating TM after 15 threads. Under high contention, even after only four threads, spinlock1 appears to be the best approach.

When the transaction time increases, the value of β, and thus, the value of p_{spl}, are also increased, making the graph for Equation 16 have the shape of a hyperbola more apparently. The performance line of the spinlock1 in Figure 7(b) clearly illustrates this prediction. Equation 16 and the ex-



(a) 95% partitioned workload



(b) 0% partitioned workload

Figure 7: Scalability on LogTM with 10 reads and 10 writes and transactions five times longer than those in Figure 5

periment results also appear to agree with each other that the thoughput of the spinlock is bounded by a maximum value when n_t is large enough. This maximum value can be computed from the equation as $\frac{2}{p_{spl}(t_0+t_{overhead\_spl})}$.

For the database lock approach, the performance line starts going above the base line for certain values of n_t. That means database lock is still a good approach for long-running transactions if the transactions do not come to the system at high enough arrival rate to make the reciprocal effect happen.

5. CONCLUSIONS

Our goal in this paper was to explore hardware support for concurrency control for workloads consisting of very short transactions. One can consider our work as exploring the performance of various options at the extreme limit of stripped-down transaction processing, where almost all overhead related to transaction processing in today's general purpose DBMSs has been removed. Our intention is to shed light on processing this subset of transaction workloads, rather than to draw conclusions that are applicable to more general classes of workloads.

We found that transactional memory can indeed be exploited to run these workloads at close to "hardware speed," with the caveat that the performance of transactional memory deteriorates at very high contention. At high contention, spinlocks perform surprisingly well (especially as compared to traditional database locking) but there is a catch: it is unclear how to efficiently resolve deadlocks, since detection is not an option. If the transactions can predict their read and write sets ahead of execution, deadlock can be avoided and spinlocks appear to be the mechanism of choice for high contention.

The fundamental question of whether or not hardware support for extreme transaction processing is worth pursuing is a complex question we cannot answer in this paper — for one thing, it depends on whether or not workloads of extremely simple, low overhead transactions are important. There is some evidence that they are — witness the growth of the "Extreme Transaction Processing (XTP)" marketplace. The question further depends upon whether hardware architects continue to produce higher-functionality implementations of HTM that can handle more robust transaction profiles. However, from our study it appears that if one wants to "squeeze the last drop of performance" out of extremely short transactions, hardware assistance must play an important role.

6. REFERENCES

[1] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A high-performance SPARC CMT processor. *IEEE Micro*, 29(2):6–16, 2009.

[2] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *SOSP '09*, pages 133–146, 2009.

[3] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS*, pages 157–168, 2009.

[4] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Trans. Knowl. Data Eng.*, 4(6):509–516, 1992.

[5] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD Conference*, pages 981–992, 2008.

[6] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[7] R. Johnson, M. Athanassoulis, R. Stoica, and A. Ailamaki. A new look at the roles of spinning and blocking. In *DaMoN '09: Proceedings of the Fifth International Workshop on Data Management on New Hardware*, pages 21–26, New York, NY, USA, 2009. ACM.

[8] R. Johnson, I. Pandis, and A. Ailamaki. Critical sections: re-emerging scalability concerns for database storage engines. In *DaMoN*, pages 35–40, 2008.

[9] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *EDBT '09*, pages 24–35, 2009.

[10] E. P. C. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*, 2010.

[11] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. In *VLDB*, page 351, 1979.

[12] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[13] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. LogTM: log-based transactional memory. pages 254 – 265, feb. 2006.

[14] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-Oriented Transaction Execution. *Proceedings of the VLDB Endowment*, 3(1), 2010.

[15] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented DBMS. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[16] M. Stonebraker and U. Çetintemel. "one size fits all": An idea whose time has come and gone (abstract). In *ICDE*, pages 2–11, 2005.

[17] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.