Exploiting Hardware Transactional Memory in Main-Memory Databases

Viktor Leis, Alfons Kemper, Thomas Neumann

Fakultät für Informatik
Technische Universität München
Boltzmannstraße 3, D-85748 Garching
`<lastname>@in.tum.de`

***Abstract*—So far, transactional memory—although a promising technique—suffered from the absence of an efficient hardware implementation. The upcoming Haswell microarchitecture from Intel introduces hardware transactional memory (HTM) in mainstream CPUs. HTM allows for efficient concurrent, atomic operations, which is also highly desirable in the context of databases. On the other hand HTM has several limitations that, in general, prevent a one-to-one mapping of database transactions to HTM transactions.**

In this work we devise several building blocks that can be used to exploit HTM in main-memory databases. We show that HTM allows to achieve nearly lock-free processing of database transactions by carefully controlling the data layout and the access patterns. The HTM component is used for detecting the (infrequent) conflicts, which allows for an optimistic, and thus very low-overhead execution of concurrent transactions.

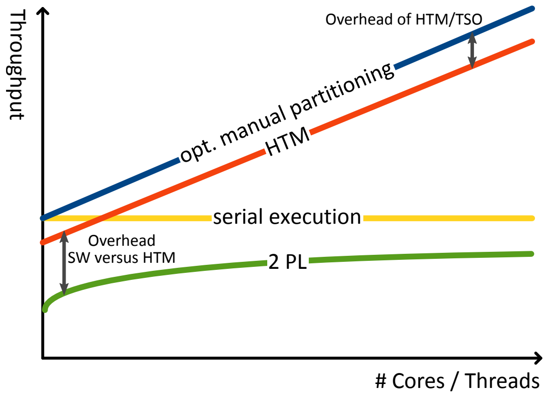


Figure 1. HTM versus 2PL, sequential, partitioned

I. Introduction

The upcoming support for hardware transactional memory (HTM) in mainstream processors like Intel's Haswell appears like a perfect fit for emerging main-memory database systems like H-Store/VoltDB [1], HyPer [2], SAP HANA [3], IBM solidDB [4], Microsoft Hekaton [5], etc. Transactional memory [6] is a very intriguing concept that allows for automatic atomic and concurrent execution of arbitrary code. Transactional memory allows for code like this:

transaction {
  $a = a - 10;$
  $b = b + 10;$
}
Transaction 1

transaction {
  $c = c - 20;$
  $a = a + 20;$
}
Transaction 2

Semantically, this code behaves quite similar to database transactions. The code sections are executed atomically and in isolation from each other. In the case of runtime conflicts (i.e., read/write conflicts or write/write conflicts) a transaction might get aborted, undoing all changes performed so far. The transaction model is a very elegant and well understood idea that is much simpler than the classical alternative, namely fine-grained locking. Locking is much more difficult to formulate correctly. Fine-grained locking is error prone and can lead to deadlocks due to differences in locking order. Coarse-grained locking is simpler, but greatly reduces concurrency. Transactional memory avoids this problem by keeping track of read and write sets and thus by detecting conflicts on the memory access level. In the upcoming Haswell architecture this is supported by hardware, which offers excellent performance.

Figure 1 sketches the benefits of our HTM-based transaction manager in comparison to other concurrency control mechanisms that we investigated. For main-memory database applications the well-known Two Phase Locking scheme was proven to be inferior to serial execution [7]! However, serial execution cannot exploit the parallel compute power of modern multi-core CPUs. Under serial execution, scaling the throughput in proportion to the number of cores would require an optimal partitioning of the database such that transactions do not cross these boundaries. This allows for "embarrassingly" parallel execution—one thread within each partition. Unfortunately, this is often not possible in practice; therefore, the upper throughput curve "opt. manual partitioning" of Figure 1 is only of theoretical nature. HTM, however, comes very close to an optimal static partitioning scheme as its transaction processing can be viewed as an adaptive dynamic partitioning of the database according to the transactional access pattern.

However, transactional memory is no panacea for transaction processing. First, database transactions also require properties like durability, which are beyond the scope of transactional memory. Second, at least the current hardware implementations of transactional memory are limited. For the Haswell microarchitecture the scope of a transaction is limited, because the read/write set, i.e., every cache line a transaction accesses, has to fit into the L1 cache with a capacity of 32KB. Furthermore, HTM transactions may fail due to a number

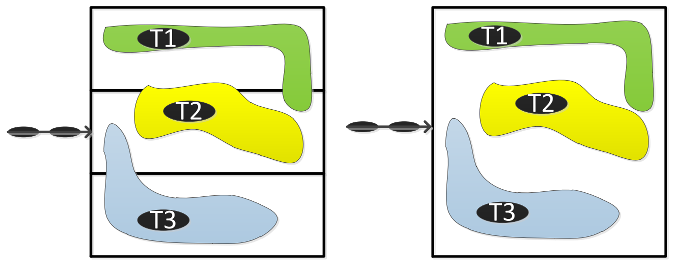


Figure 2. Static partitioning (left), Optimistic concurrency control via HTM resulting in dynamic partitioning (right)

of unexpected circumstances like collisions caused by cache associativity, hardware interrupts, etc. Therefore, it does not seem to be viable to map an entire database transaction to a single monolithic HTM transaction. In addition, one always needs a "slow path" to handle the pathological cases (e.g., associativity collisions).

We therefore propose an architecture where transactional memory is used as a building block for assembling complex database transactions. Along the lines of the general philosophy of transactional memory we start executing transactions optimistically, using (nearly) no synchronization and thus running at full clock speed. By exploiting HTM we get many of the required checks for free, without complicating the database code, and can thus reach a much higher degree of parallelism than with classical locking or latching. In order to minimize the number of conflicts in the transactional memory component, we carefully control the data layout and the access patterns of the involved operations, which allows us to avoid explicit synchronization most of the time.

Note that we explicitly do not assume that the database is partitioned in any way. In some cases, and in particular for the well-known TPC-C benchmark, the degree of parallelism can be improved greatly by partitioning the database at the schema level (using the *warehouse* attribute in the case of TPC-C). Such a static partitioning scheme is exemplified on the left-hand side of Figure 2. VoltDB for example makes use of static partitioning for parallelism [1]. But such a partitioning is hard to find in general, and users usually cannot be trusted to find perfect partitioning schemes [8]. In addition, there can always be transactions that cross partition boundaries, as shown by the partition boundary overlapping transactions T1, T2, and T3 in Figure 2 (left-hand side). These transactions have to be isolated with a serial (or locking-based) approach as the static partitioning scheme cannot guarantee their isolation. If available, we could still exploit partitioning information in our HTM approach, of course, as then conflicts would be even more unlikely. But we explicitly do not assume the presence of such a static partitioning scheme and rely on the implicit adaptive partitioning of the transactions as sketched on the right-hand side of Figure 2.

II. Background and Motivation

As databases are expected to offer ACID transactions, they have to implement a mechanism to synchronize concurrent transactions. The traditional concurrency control method used

Table I
Transaction rates for various synchronization methods

synchronization method	1 thread	4 threads
2PL	50,541	108,756
serial execution	129,937	(129,937)
manually partitioned, serial	119,232	369,549

in most database systems is some variant of two-phase locking (2PL) [9]. Before accessing a database item (tuple, page, etc.), the transaction acquires a lock in the appropriate lock mode (shared, exclusive, etc.). Conflicting operations, i.e., conflicting lock requests, implicitly order transactions relative to each other and thus ensure serializability.

In the past this model worked very well. Concurrent transaction execution was necessary to hide I/O latency, and the costs for checking locks was negligible compared to the processing costs in disk-based systems. However, this has changed in modern systems, where large parts of the data are kept in main memory, and where query processing is increasingly CPU bound. In such a setup, lock-based synchronization constitutes a significant fraction of the total execution time, in some cases even dominates the processing [7], [10].

This observation has motivated some main-memory based systems to adopt a serial execution model [7]: Instead of expensive synchronization, all transactions are executed serially, eliminating any need for synchronization. And as a main-memory based system does not have to hide I/O latency, such a model works very well for short, OLTP-style transactions.

Table I shows TPC-C transaction rates under these two models. We used our HyPer system [2] as the basis for the experiments. Clearly, the serial execution mode outperforms 2PL. Due to the inherent overhead of maintaining a synchronized lock manager in 2PL, serial execution achieves 2.6 times the transaction rate of 2PL. This is a strong argument in favor of the serial execution mode proposed by [7]. On the other hand, the figure also shows the weakness of serial execution: Increasing the degree of parallelism in 2PL increases the transaction rate. Admittedly the effect is relatively minor in the TPC-C setting, using 4 threads results in a speedup of only 2, but there still is an effect. Serial execution cannot make use of additional threads, and thus the transaction rate remains constant. As the number of cores in modern systems grows while single-threaded performance stagnates, this becomes more and more of a problem.

Systems like H-Store/VoltDB [1] or HyPer [2] tried to solve this problem by partitioning the data. Both systems would partition the TPC-C workload along the *warehouse* attribute, and would then execute all transactions concurrently that operate on separate warehouses. If transactions access more than one warehouse, the system falls back to the serial execution model. In the TPC-C benchmark this occurs for ca. 11% of the transactions. Nevertheless, this model works relatively well for TPC-C, as shown in Figure I, where it is about 3 times faster than serial execution for 4 threads. But it is not very satisfying to depend on static partitioning.

First of all, it needs human intervention. The database administrator has to specify how the data should be partitioned; HyPer has no automatic mechanism for this, whereas in H-Store there were attempts to derive such partitioning schemes automatically, e.g., Schism [11]. But, as mentioned by Larson et al. [8], a good partitioning scheme is often hard to find, in particular when workloads may shift over time. For TPC-C the partitioning schema is obvious—as it was (artificially) specified as a schema tree—but for other schemata it is not. Second, the partitioning scheme breaks if transactions frequently cross their partition boundaries. For TPC-C this is not much of a problem, as only relatively few transactions cross partition boundaries and the workload does not change, but in general it is hard to find partitioning schemes fully coherent with the transaction's access patterns. And it is important to note that partition-crossing does not necessarily imply conflicting! In the static partitioning execution model two transactions will be serialized if they access the same partition, even if the data items they access are completely distinct. This is highlighted in Figure 2 where all three transactions on the left-hand side are viewed as potentially conflicting as they (occasionally) cross their partition boundaries.

As this state of the art is not very satisfying, we will in the following develop a synchronization mechanism that is as fine-grained as 2PL and, in terms of overhead, nearly as cheap as serial execution. With our HTM-supported, dynamically-partitioned execution model the transactions shown on the right-hand side of Figure 2 are executed in parallel without conflicts as their read/write-sets do not overlap.

Note that in this paper we concentrate on relatively short, non-interactive transactions. The methods we propose are not designed for transactions that touch millions of tuples or that wait minutes for user interaction. In HyPer such long-running transactions are moved into a snapshot with snapshot-isolation semantics [2], [10]. As these snapshots are maintained automatically by the OS, there is no interaction between these long-running transactions and the shorter transactions we consider here. In general, any system that adopts our techniques will benefit from a separate snapshotting mechanism to avoid the conflicts with long-running transactions, such as OLAP queries and interactive transactions.

III. Transactional Memory

The synchronization mechanisms discussed above are usually implemented using some form of mutual exclusion (mutex). For 2PL, the DBMS maintains a lock structure that keeps track of all currently held locks. As this lock structure is continuously updated by concurrent transactions, the structure itself is protected by one (or more) mutexes [12]. On top of this, the locks themselves provide a kind of mutual exclusion mechanism, and block a transaction if needed.

The serial execution paradigm is even more extreme, there one lock protects the whole database (or the whole partition for partitioned execution). The problem with these locks is that they are difficult to use effectively. In particular, finding the right lock granularity is difficult. Coarse locks are cheap, but limit concurrency. Fine-grained locks allow for more concurrency, but are more expensive and can lead to deadlocks.

For quite some time now, transactional memory is being proposed as an alternative to fine grained locking [6]. The key idea behind transactional memory is that a number of operations can be combined into a transaction, which is then executed atomically. Consider the following small code fragment for transferring money from one account to another account (using GCC 4.7 syntax):

```
transfer(from,to,amount)
  __transaction_atomic {
    account[from]-=amount;
    account[to]+=amount;
  }
```

The code inside the *atomic* block is guaranteed to be executed atomically, and in isolation. In practice, the transactional memory observes the read set and write set of transactions, and executes transactions concurrently as long as the sets do not conflict. Thus, transfers can be executed concurrently as long as they affect different accounts, they are only serialized if they touch a common account. This behavior is very hard to emulate using locks. Fine-grained locking would allow for high concurrency, too, but would deadlock if accounts are accessed in opposite order. Transactional memory solves this problem elegantly.

Transactional memory has been around for a while, but has usually been implemented as Software Transactional Memory (STM), which emulated this behavior in software. Although STM does remove the complexity of lock maintenance, it causes a significant slowdown during execution and thus had limited practical impact [13].

A. *Hardware Support for Transactional Memory*

This will change with the Haswell microarchitecture from Intel, which offers Hardware Transactional Memory [14]. Note that Haswell is not the first CPU with hardware support for transactional memory, for example IBM's Blue Gene/Q supercomputers [15] and System z mainframes [16] offered it before, but it is the first mainstream CPU to implement HTM. And in hardware, transactional memory can be implemented much more efficiently than in software: Haswell uses its highly optimized cache coherence protocol, which is needed for all multi-core processors anyway, to track read and write set collisions [17]. Therefore, Haswell offers HTM nearly for free.

Even though HTM is very efficient, there are also some restrictions. First of all, the size of a hardware transaction is limited. For the Haswell architecture it is limited to the size of the L1 cache, which is 32 KB. This implies that, in general, it is not possible to simply execute a database transaction as one monolithic HTM transaction. Even medium-sized database transactions would be too large. Second, in the case of conflicts, the transaction *fails*. In this case the CPU undoes all changes, and then reports an error that the application has to handle. And finally, a transaction might fail

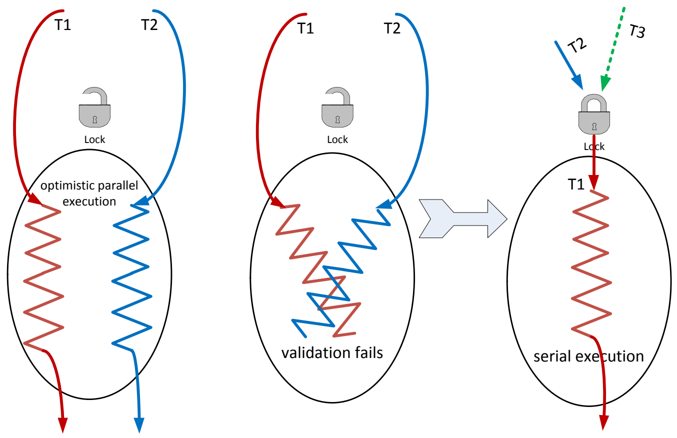


Figure 3. Lock elision (left), conflict (middle), and serial execution (right)

due to spurious implementation details like cache associativity limits, certain interrupts, etc. So, even though in most cases HTM will work fine, there is no guarantee that a transaction will ever succeed (if executed as an HTM transaction).

Therefore, Intel proposes (and explicitly supports by specific instructions) using transactional memory for lock elision [17]. Conceptually, this results in code like the following:

```
transfer(from,to,amount)
  atomic-elide-lock (lock) {
    account[from]-=amount;
    account[to]+=amount;
  }
```

Here, we still have a lock, but ideally the lock is not used at all—it is elided. When the code is executed, the CPU starts an HTM transaction, but does *not* acquire the lock as shown on the left-hand side of Figure 3. Only when there is a conflict the transaction rolls back, acquires the lock, and is then executed *non-transactionally*. The right-hand side of Figure 3 shows the fallback mechanism to exclusive serial execution, which is controlled via the (previously elided) lock. This lock elision mechanism has two effects: 1) ideally, locks are never acquired and transactions are executed concurrently as much as possible 2) if there is an abort due to a conflict or hardware-limitation, there is a "slow path" available that is guaranteed to succeed.

B. Caches and Cache Coherency

Even though Intel generally does not publish internal implementation details, Intel did specify two important facts about Haswell's HTM feature [17]:

- The cache coherency protocol is used to detect transactional conflicts.
- The L1 cache serves as a transactional buffer.

Therefore, it is crucial to understand Intel's cache architecture and coherency protocol.

Because of the divergence of DRAM and CPU speed, modern CPUs have multiple caches in order to accelerate memory accesses. Intel's cache architecture is shown in Figure 4, and consists of a local L1 cache (32 KB), a local L2 cache (256 KB), and a shared L3 cache (2-30 MB). All caches use

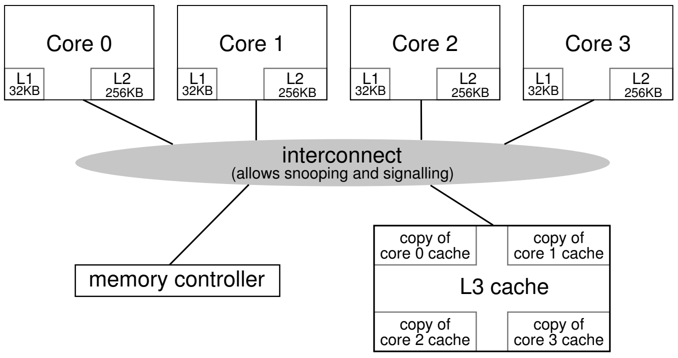


Figure 4. Intel cache architecture

64 byte cache blocks (lines) and all caches are transparent, i.e., programs have the illusion of having only one large main memory. Because on multi-core CPUs each core generally has at least one local cache, a cache coherency protocol is required to maintain this illusion.

Both Intel and AMD use extensions of the well-known MESI protocol [18]. The name of the protocol derives from the four states that each cache line can be in (Modified, Exclusive, Shared, or Invalid). To keep multiple caches coherent, the caches have means of intercepting ("snooping") each other's load and store requests. For example, if a core writes to a cache line which is stored in multiple caches (Shared state), the state must change to Modified in the local cache and all copies in remote caches must be invalidated (Invalid state). This logic is implemented in hardware using the cache controller of the shared L3 cache that acts as a central component where all coherency traffic and all DRAM requests pass through.

The key insight that allows for an efficient HTM implementation is that the L1 cache can be used as a local buffer. All transactionally read or written cache lines are marked and the propagation of changes to other caches or main memory is prevented until the transaction commits. Read/write and write/write conflicts are detected by using the same snooping logic that is used to keep the caches coherent. And since the MESI protocol is always active and commits/aborts require no inter-core coordination, transactional execution on Haswell CPUs incurs almost no overhead. The drawback is that the transaction size is limited to the L1 cache. This is fundamentally different from IBM's Blue Gene/Q architecture, which allows for up to 20 MB per transaction using a multi-versioned L2 cache, but has relatively large runtime overhead [15].

Besides the nominal size of the L1 cache, another limiting factor for the maximum transaction size is cache associativity. Caches are segmented into sets of cache lines in order to speed up lookup and to allow for an efficient implementation of the pseudo-LRU replacement strategy (in hardware). Haswell's L1 cache is 8-way associative, i.e., each cache set has 8 entries. This has direct consequences for HTM, because all transactionally read or written cache lines must be marked and kept in the L1 cache until commit or abort. Therefore, when a transaction writes to 9 cache lines that happen to reside in the same cache set, the transaction is aborted. And since the mapping from memory address to cache set is deterministic

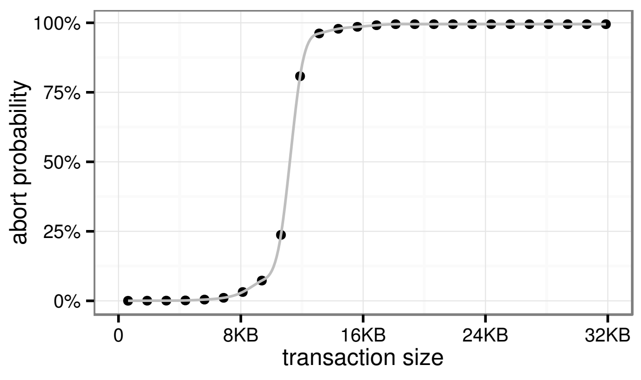


Figure 5. Aborts from random memory writes

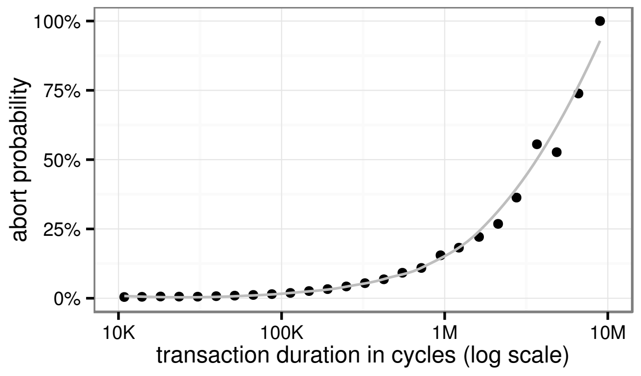


Figure 6. Aborts from transaction duration

(bits 7-12 of the address are used), restarting the transaction does not help, and an alternative fallback path is necessary for forward progress.

In practice, bits 7-12 of memory addresses are fairly random, and aborts of very small transactions are unlikely. Nevertheless, Figure 5 shows that the abort probability quickly rises when more than 128 random cache lines (only about one quarter of the L1 cache) are accessed[1]. This surprising fact is caused by a statistical phenomenon related to the birthday paradox: For example with a transaction size of 16 KB, *for any one* cache set it is quite unlikely that it contains more than 8 entries. However, at the same time, it is likely that *at least one* cache set exceeds this limit. An eviction of a line from the cache automatically leads to a failure of this transaction as it would become impossible to detect conflicting writes to this cache line.

The previous experiment was performed with accesses to memory addresses fully covered by the translation lookaside buffer (TLB). TLB misses do *not* immediately cause transactions to abort, because, on x86 CPUs, the page table lookup is performed by the hardware (and not the operating system). However, TLB misses do increase the abort probability, as they cause additional memory accesses during page table walks.

Besides memory accesses, another important reason for transactional aborts is interrupts. Such events are unavoidable in practice and limit the maximum duration of transactions. Figure 6 shows that transactions that take more than 1 million CPU cycles (about 0.3 ms) will likely be aborted, even if they only compute and execute no memory operations. These results clearly show that Haswell's HTM implementation cannot be used for long-running transactions but is designed for short critical sections.

Despite these limitations we found that Haswell's HTM implementation offers excellent scalability as long as transactions are short. Furthermore, one has to keep in mind that Haswell is Intel's first HTM implementation. It is therefore not unlikely that in future CPU generations HTM will further be improved, e.g., the larger L2 cache may serve as an additional transactional buffer.

IV. HTM-supported Transaction Management

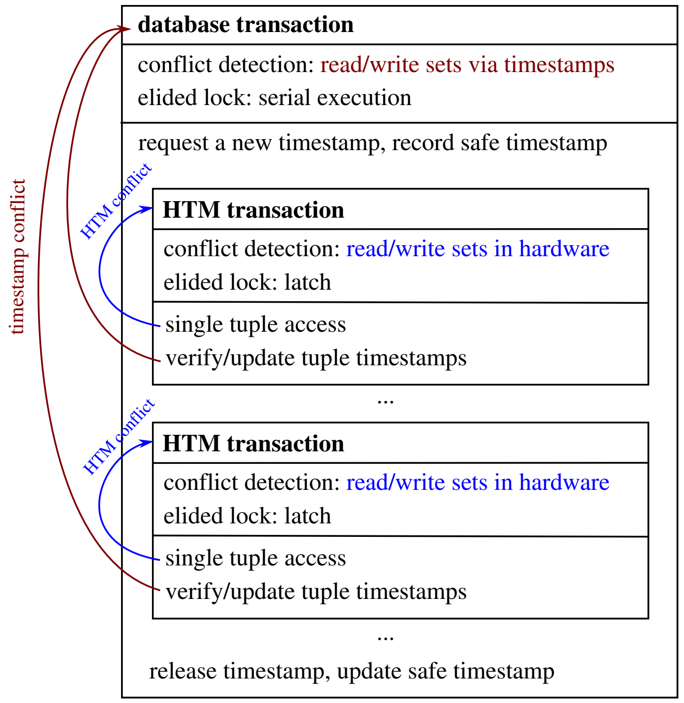
Writing scalable *and* efficient multithreaded programs is widely considered a very difficult problem. In particular, it is very hard to decide at which granularity latching/locking should be performed: if very fine-grained latching is used, the additional overhead will annihilate any speedup from parallelism; with coarse-grained latches, parallelism is, limited. For non-trivial programs, this is a very difficult problem, and the most efficient choice can often only be decided empirically. The granularity problem is even more difficult for a database system because it must efficiently support arbitrary workloads. With hardware support, transactional memory offers an elegant solution: As long as conflicts are infrequent, HTM offers the parallelism of fine-grained latching, but without its overhead; if hotspots occur frequently, the best method in main-memory databases is serial execution, which is exactly the fallback path for HTM conflicts. Therefore, HTM is a highly promising building block for high performance database systems.

A. Mapping Database Transactions to HTM Transactions

As the maximum size of hardware transactions is limited, only a database transaction that is small can directly be mapped to a single hardware transaction. Therefore, we assemble complex database transactions by using hardware transactions as building blocks, as shown in Figure 7. The key idea here is to use a customized variant of timestamp ordering (TSO) to "glue" together these small hardware transactions. TSO is a classic concurrency control technique, which was extensively studied in the context of disk-based and distributed database systems [19], [20]. For disk-based systems, TSO is not competitive to locking because most read accesses result in an update of the read timestamp, and thus a write to disk. These timestamp updates are obviously much cheaper in RAM. On the opposite, fine-grained locking is much more expensive than maintaining timestamps in main memory, as we will show in Section VI.

Timestamp ordering uses read and write timestamps to identify read/write and write/write conflicts. Each transaction is associated with a monotonically increasing timestamp, and whenever a data item is read or updated its associated timestamp is updated, too. The *read timestamp* of a data item records the youngest reader of this particular item,

[1]The Haswell system is described in Section VI.



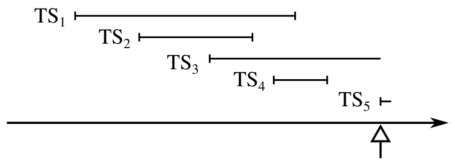
database transaction
conflict detection: read/write sets via timestamps
elided lock: serial execution
request a new timestamp, record safe timestamp
HTM transaction
conflict detection: read/write sets in hardware
elided lock: latch
single tuple access
verify/update tuple timestamps
...
HTM transaction
conflict detection: read/write sets in hardware
elided lock: latch
single tuple access
verify/update tuple timestamps
...
release timestamp, update safe timestamp
timestamp conflict
HTM conflict
HTM conflict

Figure 7. Transforming database transactions into HTM transactions

and the *write timestamp* records the last writer. This way, a transaction recognizes if its operation collides with an operation of a "younger" transactions (i.e., a transaction with a larger timestamp), which would be a violation of transaction isolation. In particular, an operation fails if a transaction tries to read data from a younger transaction, or if a transaction tries to update a data item that has already been read by a younger transaction. Note that basic TSO [19] has to be refined to prevent phantoms. Furthermore, some care is needed to prevent non-recoverable schedules, as by default transactions are allowed to read data from older, but potentially non-committed, transactions.

To resolve both issues (phantoms and dirty reads), we deviate from basic TSO by introducing a "safe timestamp", i.e., a point in time where it is known that all older transactions have already been committed. With classical TSO, when a transaction tries to read a dirty data item (marked by a dirty bit) from another transaction, it must wait for that transaction to finish. In main-memory database systems running at full clock speed, waiting is very undesirable.

We avoid both waiting and phantom problems with the *safe timestamp* concept. The safe timestamp $\mathrm{TS}_{\mathrm{safe}}$ is the youngest timestamp for which holds: All transactions with an older timestamp $\mathrm{TS}_{\mathrm{old}}$ with $\mathrm{old} \leq \mathrm{safe}$ have already been committed or aborted. While regular TSO compares transaction timestamps directly, we compare timestamps to the safe timestamp of each transaction: Everything that is older than the safe timestamp can be safely read, and everything that has been read only by transactions up to the safe timestamp can safely be modified. Note that we could also access or modify some tuples with newer timestamps, namely those from transactions that have already committed in between this transaction's begin and now. But this would require complex and expensive checks during tuple access, in particular if one also wants to prevent phantoms. We therefore use the safe timestamp as a cheap, though somewhat conservative, mechanism to ensure serializability. In the scenario



the safe timestamp of TS_5 would be set to TS_2. So transaction TS_5 would validate its read access such that only data items with a write timestamp $\mathrm{TS}_{\mathrm{W}} \leq \mathrm{TS}_2$ are allowed. Write accesses on behalf of TS_5 must additionally verify that the read timestamp of all items to be written satisfies the condition $\mathrm{TS}_{\mathrm{R}} \leq \mathrm{TS}_2$. Obviously, a read or write timestamp $\mathrm{TS} = \mathrm{TS}_5$ is permitted as well—in case a transaction accesses the same data item multiple times.

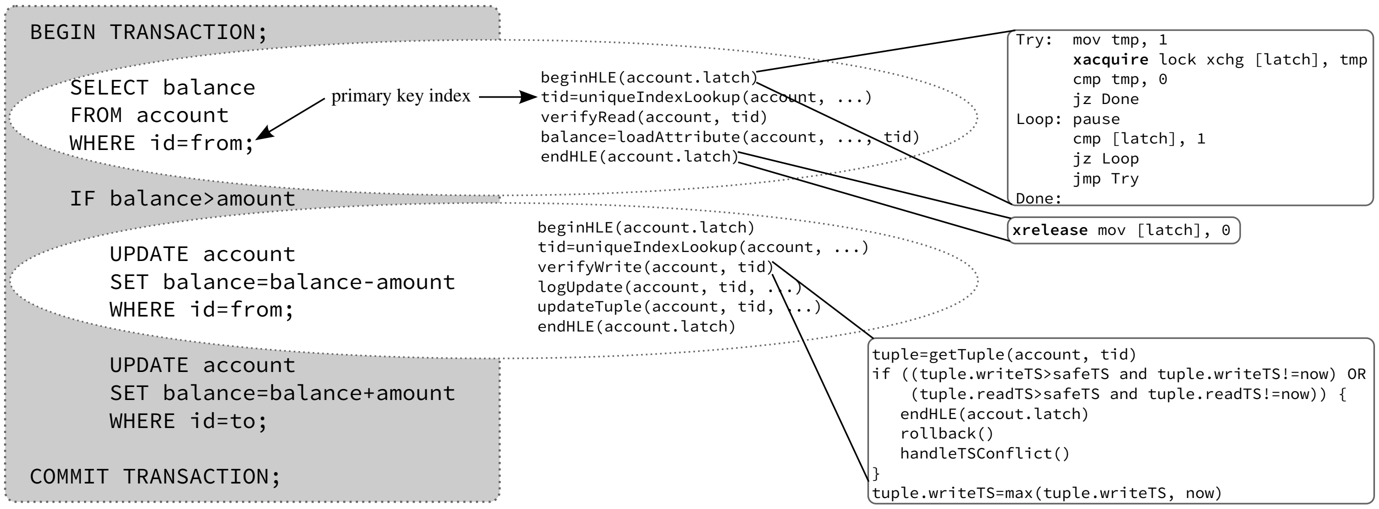
B. *Conflict Detection and Resolution*

In our scheme, the read and the write timestamps are stored at each tuple. After looking up a tuple in an index, its timestamp(s) must be verified and updated. Each single tuple access, including index traversal and timestamp update, is executed as a hardware transaction using hardware lock elision. The small granularity ensures that false aborts due to hardware limitations are very unlikely, because Haswell's hardware transactions can access dozens of cache lines (cf. Section III).

Nevertheless, two types of conflicts may occur: If HTM detects a conflict (short blue arrows in Figure 7), the hardware transaction is restarted, but this time the latch is acquired. Rolling back a hardware transaction is very cheap, as it only involves invalidating the transactionally modified cache lines, and copies of the original content can still be found in the L2 and/or L3 cache.

For timestamp conflicts, which are detected in software (long red arrows in Figure 7), the system must first roll back the database transaction. This rollback utilizes the "normal" logging and recovery infrastructure of the database system, i.e., the undo-log records of the partial database transaction are applied in an ARIES-style compensation [21]. Then, the transaction is executed serially by using a global lock, rolling the log forward again. This requires logical logging and non-interactive transactions, as we cannot roll a user action backward or forward, but, as already mentioned in Section II, we use snapshots to isolate interactive transactions from the rest of the system [10]. The fallback to serial execution ensures forward progress, because in serial execution a transaction will never fail due to conflicts. Note that it is often beneficial to optimistically restart the transaction a number of times instead of resorting to serial execution immediately, as serial execution is very pessimistic and prevents parallelism.

Figure 8 details the implementation of a database transaction using hardware lock elision and timestamps. The splitting



```
BEGIN TRANSACTION;

  SELECT balance
  FROM account
  WHERE id=from;

  IF balance>amount

    UPDATE account
    SET balance=balance-amount
    WHERE id=from;

    UPDATE account
    SET balance=balance+amount
    WHERE id=to;

COMMIT TRANSACTION;
```

```
beginHLE(account.latch)
tid=uniqueIndexLookup(account, ...)
verifyRead(account, tid)
balance=loadAttribute(account, ..., tid)
endHLE(account.latch)
```

```
Try:  mov tmp, 1
      xacquire lock xchg [latch], tmp
      cmp tmp, 0
      jz Done
Loop: pause
      cmp [latch], 1
      jz Loop
      jmp Try
Done:
```

```
xrelease mov [latch], 0
```

```
beginHLE(account.latch)
tid=uniqueIndexLookup(account, ...)
verifyWrite(account, tid)
logUpdate(account, tid, ...)
updateTuple(account, tid, ...)
endHLE(account.latch)
```

```
tuple=getTuple(account, tid)
if ((tuple.writeTS>safeTS and tuple.writeTS!=now) OR
    (tuple.readTS>safeTS and tuple.readTS!=now)) {
   endHLE(accout.latch)
   rollback()
   handleTSConflict()
}
tuple.writeTS=max(tuple.writeTS, now)
```

Figure 8. Implementing database transactions with timestamps and hardware lock elision

of stored procedures into smaller HTM transactions is fully automatic (done by our compiler) and transparent for the programmer. As shown the example, queries or updates (within a database transaction) that access a single tuple through a unique index are directly translated into a single HTM transaction. Larger statements like non-unique index lookups should be split into multiple HTM transactions, e.g., one for each accessed tuple. The index lookup and timestamp checks are protected using an elided latch, which avoids latching the index structures themselves. The latch is implemented as a lightweight spinlock in x86 assembler. The `xacquire` and `xrelease` special instruction prefixes for Haswell [14] cause the latch to be elided on the first try, and an HTM transaction to be started instead. On older processors these prefixes are ignored and the latch is immediately acquired, making this code backwards compatible.

C. Optimizations

How the transaction manager handles timestamp conflicts is very important for performance. If the conflict is only caused by the conservatism of the safe timestamp (i.e., regular TSO would have no conflict), it is sometimes possible to avoid rolling back the transaction. If the conflicting transaction has a smaller timestamp *and* has already finished, the apparent conflict can be ignored. This optimization is possible because the safe timestamp cannot overtake a currently running transaction's timestamp.

As mentioned before, it is often beneficial to restart an aborted transaction a number of times, instead of immediately falling back to serial execution. In order for the restart to succeed, the safe timestamp must have advanced past the conflict timestamp. Since this timestamp is available (it triggered the abort), the transaction can wait, while periodically recomputing the safe timestamp until it has advanced sufficiently. Then the transaction can be restarted with a new timestamp and safe timestamp. The disadvantage of this approach is that during this waiting period no useful work is performed by the thread.

A more effective strategy is to suspend the aborted transaction and execute other transactions instead. Once the safe timestamp has advanced past the conflicting transaction's timestamp that transaction can be resumed. This strategy avoids wasteful waiting. We found rollback and re-execution to be quite cheap because the accessed data is often in cache. Therefore, our implementation immediately performs an abort after a timestamp conflict, as shown in Figure 8, and executes other transactions instead, until the safe timestamp has sufficiently advanced. We additionally limit the number of times a transaction is restarted before falling back to serial execution—thus ensuring forward progress.

While our description here and also our initial implementation uses both read and write timestamps, it is possible to avoid read timestamps. Read timestamps are a bit unfortunate, as they can cause "false" HTM conflicts due to parallel timestamp updates, even though the read operations themselves would not conflict. Semantically the read timestamps are used to detect if a tuple has already been read by a newer transaction, which prohibits updates by older transactions (as they would destroy serializability). However, the read timestamps can be avoided by keeping track of the write timestamps of all data items accessed (read or written) by a certain transaction. Then, at commit time, the transaction re-examines the write timestamps of all data items and aborts if any one of them has changed [22], ensuring serializability. We plan to implement this technique in future work, and expect to get even better performance in the case of read hotspots.

It is illustrative to compare our scheme to software transactional memory (STM) systems. Indeed, our scheme can be considered an HTM-supported implementation of STM. However, we get significantly better performance than pure STM by exploiting DBMS domain knowledge. For example, index structures are protected from concurrent modifications by the HTM transaction, but are not tracked with timestamps, as full transaction isolation would in fact be undesirable there. This is similar to B-tree latching in disk-based systems—

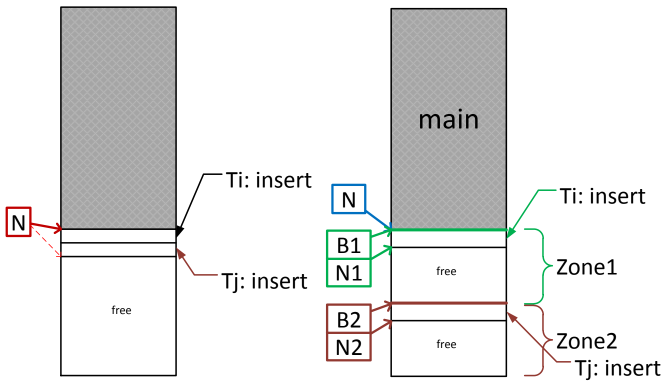


Figure 9. Avoiding hotspots by zone segmentation

however, at minimal cost. The indexed tuples themselves are isolated via timestamps to ensure serializable transaction behavior. Note further that our interpretation of timestamps is different from regular TSO [19]: Instead of deciding about transaction success and failure as in TSO, we use timestamps to detect intersecting read/write sets, just like the hardware itself for the HTM part. In the case of conflicts, we do not abort the transaction or retry with a new timestamp an indefinite number of times, but fall back to the more restrictive sequential execution mode that ensures forward progress and guarantees the eventual success of every transaction.

V. HTM-Friendly Data Storage

Transactional memory synchronizes concurrent accesses by tracking read and write sets. This avoids the need for fine-grained locking and greatly improves concurrency as long as objects at different memory addresses are accessed. However, because HTM usually tracks accesses at cache line granularity, false conflicts may occur. For example, if the two data items A and B happen to be stored in a single cache line, a write to A causes a conflict with B. This conflict would not have occurred if each data item had its own dedicated lock. Therefore, HTM presents additional challenges for database systems that must be tackled in order to efficiently utilize this feature.

A. Data Storage with Zone Segmentation

With a straightforward contiguous main-memory data layout, which is illustrated on the left-hand side of Figure 9, an insert into a relation results in appending the tuple to the end of the relation. It is clear that such a layout does not allow concurrent insertions, because each insert writes to the end of the relation. Additionally, all inserts will try to increment some variable N which counts the number of tuples. The memory location at the end of the table and the counter N are hotspots causing concurrent inserts to fail.

In order to allow for concurrent inserts, we use multiple zones per relation, as shown on the right-hand side of Figure 9. Each relation has a constant number of these zones, e.g., two times the number of hardware threads. A random zone number is assigned to each transaction, and all inserts of that transaction use this local zone. The same zone number is also used for inserts into other relations. Therefore, with an appropriately chosen number of zones, concurrent inserts can proceed with only a small conflict probability, even if many relations are affected. Besides the relatively small insert zones, each relation has a main zone where, for large relations, most tuples are stored.

The boundary is stored in a counter N. For each zone i, the base B_i and the next insert position N_i are maintained. When a zone becomes full (i.e., when N_i reaches B_{i+1}), it is collapsed into the neighboring zone, and a new zone at the end of the table is created. Note that no tuples are copied and the tuple identifiers do not change, only the sizes of zones need to be adjusted. As a consequence, collapsing zones does not affect concurrent access to the tuples. Eventually, the insert zones are collapsed with the large contiguous main area. For a main-memory databases this guarantees very fast scan performance at clock speed during query processing. The counters N_i and B_i should be stored in separate cache lines for each zone, as otherwise unnecessary conflicts occur while updating these values.

B. Index Structures

Besides the logical isolation of transactions using 2PL or TSO, database systems must isolate concurrent accesses to index structures. In principle, any data structure can be synchronized using HTM by simply wrapping each access in a transaction. In this section we first discuss how scalability can be improved by avoiding some common types of conflicts, before showing that HTM has much better performance than traditional index synchronization via fine-grained latches.

One common problem is that indexes often have a counter that stores the number of key/value pairs and prevents concurrent modifications. Fortunately, this counter is often not needed and can be removed. For data structures that allocate small memory chunks, another source of HTM conflicts is memory allocation. This problem can be solved by using an allocator that has a thread-local buffer[2].

Surrogate primary keys are usually implemented as ascending integer sequences. For tree-based index structures, which maintain their data in sorted order, this causes HTM conflicts because all concurrent inserts try to access memory locations in the same vicinity, as illustrated on the left-hand side of Figure 10. This problem is very similar to the problem of concurrently inserting values into a table discussed above, and indeed the solution is similar: If permitted by the application, the integer sequence is partitioned into multiple constant-sized chunks and values are handed out from one of the chunks

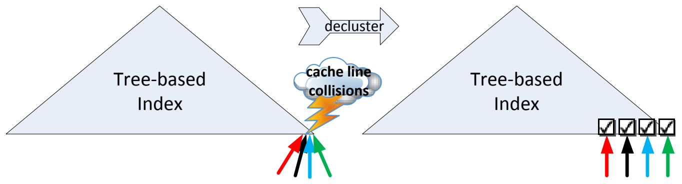


Figure 10. Avoiding hotspots by declustering surrogate key generation

[2]One such allocator is the open source library tcmalloc (Thread-Caching Malloc): http://goog-perftools.sourceforge.net/doc/tcmalloc.html

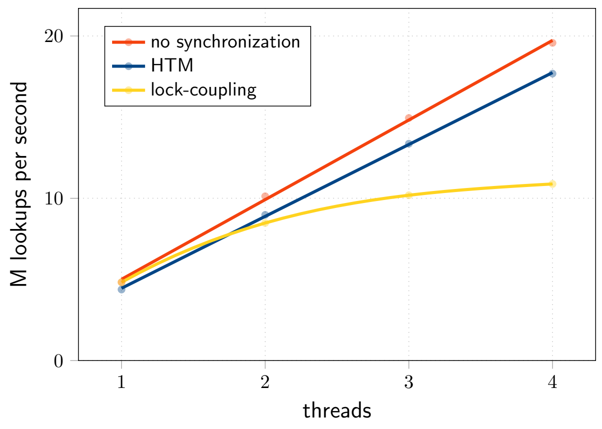


Figure 11. Synchronized index lookup, 16M sparse 4 byte keys

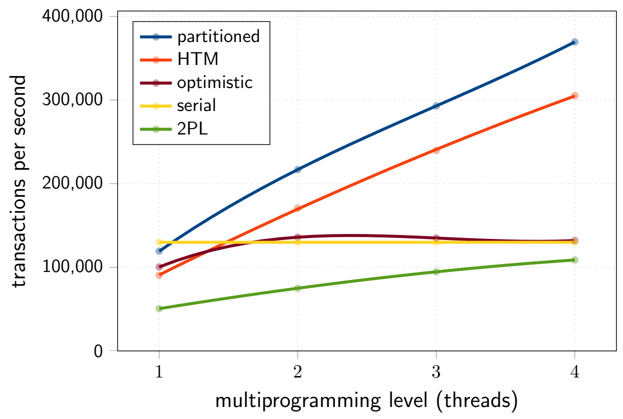


Figure 12. Scalability with TPC-C on desktop system

depending on the transactions' zone number. This prevents interference of parallel index tree insertions as they are spread across different memory locations—as shown on the right of Figure 10. Once all values from a chunk are exhausted, the next set of integers is assigned to it. Note that hash tables are not affected by this problem because the use of a hash function results in a random access pattern which leads to a low conflict probability. But of course, as a direct consequence of this randomization, hash tables do not support range scans.

The Adaptive Radix Tree (ART) [23] is an efficient ordered indexing structure for main-memory databases. Like B-Trees, ART can be synchronized with fine-grained latching, holding at most two short term node latches at any time ("lock-coupling"). We implemented this approach using efficient read-write spinlocks. Figure 11 shows the results for an experiment with 100% reads[3], i.e., no latch contention. Nevertheless, latching does not scale. The reason is that latching of tree structures causes many additional, unnecessary cache misses: Each time a thread acquires a read latch for a node (by writing to the node), all copies of that node are invalidated in the caches of all other cores. Thus, the frequently accessed nodes high up in the tree, which would normally be cached, are not in cache when accessed concurrently. HTM, in contrast, offers excellent performance and is only 10% slower than unsynchronized access.

VI. Evaluation

For most experiments we used an Intel i5 4670T Haswell processor with 4 cores, 6 MB shared L3 cache, and full HTM support through the Intel Transactional Synchronization Extensions. The maximum clock rate is 3.3 GHz, but can only be achieved when only one core is fully utilized. When utilizing all cores, we measured a sustained clock rate of 2.9 GHz. Therefore, the total clock rate is only 3.5 times higher with 4 cores, and a speedup of 4 is not realistic. The system has 16 GB of RAM and runs Linux 3.10.

By default, HyPer uses serial execution similar to VoltDB [1]; multiple threads are only used if the schema has been partitioned by human intervention. In the following we will call these execution modes *serial* and *partitioned*. Note that the partitioned mode used by HyPer (as in other systems) is somewhat cheating, since the partitioning scheme has to be explicitly provided by a human, and a good partitioning scheme is hard to find in general. In addition to these execution modes we included a *2PL* implementation, described in [10], as baseline for comparisons to standard database systems, and the hardware transactional memory approach (*HTM*) proposed here. We also include TSO with coarse-grained latches (*optimistic*) instead of HTM to show that TSO alone is not sufficient for good performance.

For most experiments, we used the well-known TPC-C benchmark as basis (without "think times", the only deviation from the official benchmark rules). We set the number of warehouses to 32, and for the partitioning experiments the strategy was to partition both the data and the transactions by the main warehouse. We used the Adaptive Radix Tree [23] as index structure, although the scalability is similar with hash tables and red-black trees. In the following, we first look at scalability results for TPC-C and then study the interaction with HTM in microbenchmarks.

A. TPC-C Results

In a first experiment, we ran TPC-C and varied the number of threads up to the number of available cores. The results are shown in Figure 12 and reveal the following: First, classical 2PL is clearly inferior to all other approaches. Its overhead is too high, and it is even dominated by single-threaded serial execution. The latching-based optimistic approach has less overhead than 2PL, but does not scale because the coarse-grained (relation-level) latches severely limit concurrency. Both the partitioned scheme and HTM scale very well, with partitioning being slightly faster. But note that this is a comparison of a human-assisted approach with a fully automatic approach. Furthermore, the partitioning approach works so well only because TPC-C is "embarrassingly partitionable" in this low MPL setup, as we will see in the next experiment.

The reason that partitioning copes well with TPC-C is that most transactions stay within a single partition. By default, about 11% of all transactions cross partition boundaries (and

[3]In this experiment each lookup key depends on the previous lookup, which hides out-of-order execution effects.

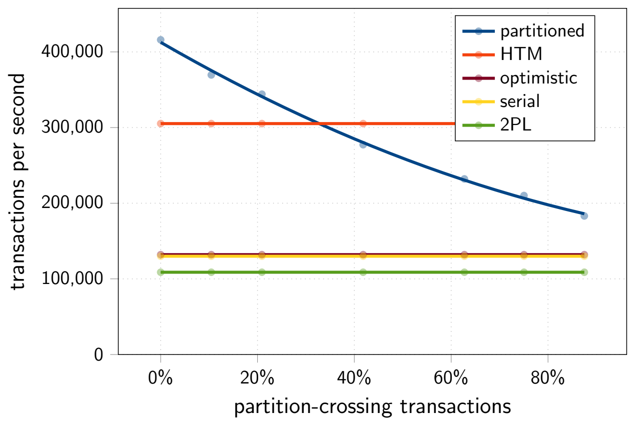


Figure 13. TPC-C with modified partition-crossing behavior

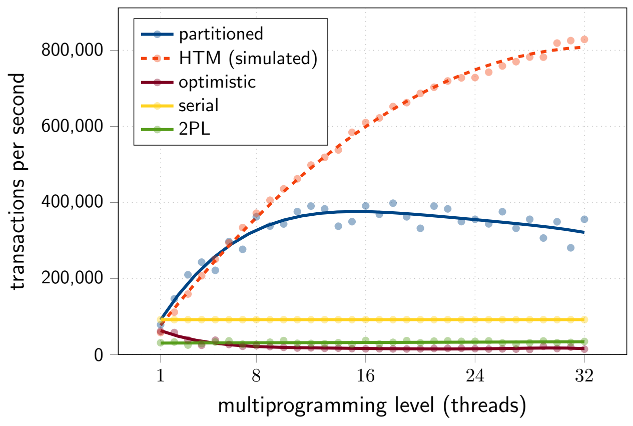


Figure 14. Scalability with TPC-C on 4-socket, 32-core server system

therefore require serial execution to prevent collisions in a lock-free system). The performance depends crucially on the ability of transactions to stay within one partition. As shown in Figure 13, varying the percentage of partition-crossing transactions has a very deteriorating effect on the partitioning approach, while the other transaction managers are largely unaffected because, in the case of TPC-C, partition-crossing does not mean conflicting. Therefore, picking the right partitioning scheme would be absolutely crucial; however, it is often hard to do—in particular if transactions are added to the workload dynamically.

So far, our experiments have been performed on a desktop system with 4 cores because larger systems with more cores and support for multiple sockets were not available at the time of writing. We expect such large multi-socket systems to benefit even more from HTM, because the cache coherency protocol can be used to efficiently implement HTM, whereas conventional, lock-based synchronization is even more expensive on such systems. Figure 14 shows the simulated performance of our HTM-based transaction manager on a 32 core system. The simulation uses read/write sets from binary instrumentation of the transaction code to decide which transactions can be executed in parallel and which have to be run serially. Figure 14 also shows that on large systems the performance of 2PL and the optimistic (latching-based) approach is even worse than on the desktop system. This is caused by the fact that lock acquisition results in many cache coherency misses, which are much more expensive on multi-socket systems because no cache is shared by all cores. The static partitioning approach performs better, as it avoids expensive synchronization, but does not scale to a large number of cores for another reason: partition-crossing transactions, which require serial execution, are the scalability bottleneck. By Amdahl's law, 11% serial transactions (as in TPC-C) result in a maximum speedup of 9.1.

B. *Microbenchmarks*

Our transaction manager was designed to be lightweight. Nevertheless, there is some overhead in comparison with an unsynchronized, purely single-threaded implementation. We determined the overhead by running the TPC-C benchmark using only one thread and enabling each feature separately: The HTM-friendly memory layout, including zone segmentation (with 8 zones), added 5% overhead, mostly because of reduced cache locality. The HLE spinlocks, which are acquired for each tuple access, added 7% overhead. Checking and updating the timestamps, slowed down execution by 10%. Finally, transaction management, e.g., determining the safe timestamp, the transaction ID, etc. caused 7% overhead. In total, these changes amounted to a slowdown of 29%. HyPer compiles transactions to very efficient machine code, so any additional work will have noticeable impact. However, this is much lower than the overhead of the 2PL implementation, which is 61%! And of course the overhead is completely paid off by the much superior scalability of the HTM approach.

One interesting question is if it would be possible to simply execute a database transaction as one large HTM transaction. To analyze this, we used binary instrumentation of the generated transaction code to record the read and write sets of all TPC-C transactions. We found that only the *delivery* and *order-status* transactions have a cacheline footprint of less than 7 KB and could be executed as HTM transactions. The other transactions access between 18 KB and 61 KB, and would usually exhaust the transactional buffer. Therefore, executing TPC-C transactions as monolithic HTM transactions is not possible. And other workloads will have transactions that are much larger than the relatively simple TPC-C transactions. Therefore, a mechanism like our timestamp scheme is required to cope with large transactions.

As we discussed in Section IV, there are two types of conflicts: timestamp conflicts and HTM conflicts. Timestamp conflicts must be handled by the transaction manager and usually result in a rollback of the transaction. We measured that 12% of all TPC-C transactions were aborted due to a timestamp conflict, but only 0.5% required more than 2 restarts. Most aborts occur at the *warehouse* relation, which has only 32 tuples but is updated frequently.

While HTM conflicts do not result in a rollback of the entire transaction, they result in the acquisition of relation-level latches—greatly reducing concurrency. Using hardware counters, we measured the HLE abort rate of TPC-C, and

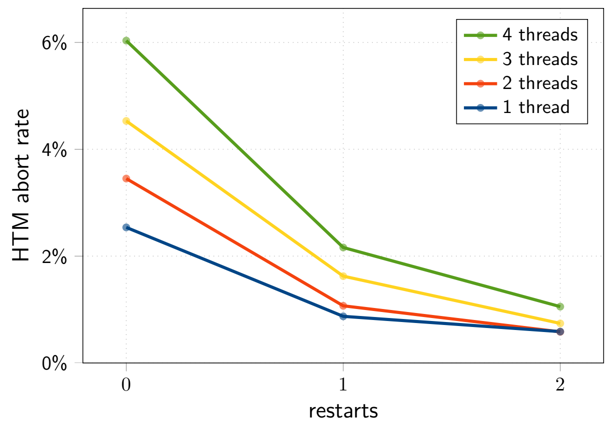


Figure 15. HTM abort rate with 8 declustered insert zones

found that 6% of all HLE transactions were aborted. This rate can be reduced by manually restarting transactions after abort by using Restricted Transaction Memory (RTM) instructions instead of HLE. As Figure 15 shows, the abort rate can be reduced greatly by restarting aborted transaction, i.e., most aborts are transient. With 4 threads, restarting has only a small positive effect on the overall transaction rate (1.5%), because a 6% abort rate is still "small enough" for 4 threads. On systems with more threads, restarting to avoid falling back to latch acquisition too often may become crucial.

These low abort rates are the outcome of our HTM-friendly storage layout from Section V. With only one zone, the HLE abort rate rises from 6% to 14%, and the clashes often do not vanish after restarts. Therefore, a careful data layout is absolutely mandatory to benefit from HTM. Note though that we did not decluster surrogate key generation, which makes conflicts even more unlikely, but would have required changes to the benchmark.

C. Lightweight Locking

2PL has been developed with disk-based databases in mind. We also implemented the pessimistic locking protocol "Very Lightweight Locking" (VLL) [24] which was designed for main-memory database systems. VLL avoids the bottleneck of a single lock manager structure, and instead adds counters to each tuple that indicate how many threads want to access the tuple. We evaluated the performance with the same microbenchmark, which updates 10 tuples per transaction (as in the original paper) and used 4 threads. For VLL we used lock-coupling or HTM for index synchronization:

method	low contention	high contention
VLL, lock-coupling	345,375	296,162
VLL, HTM	826,119	435,210
HTM (TSO)	1,181,964	661,700

We found that (i) VLL indeed has much smaller overhead than 2PL, (ii) when lock-coupling is used for index synchronization, the overall performance is dominated by the latching overhead, and (iii) the scalability is limited by the critical section in which all locks are acquired.

D. Software Transactional Memory

Although the availability of HTM is a very recent development, there has been considerable research on software transactional memory. We used the state-of-the-art STM system tinySTM (version 1.0.3) [25]. However, we obtained only 30% of the single-threaded, unsynchronized performance when executing TPC-C directly using STM transactions. The results were disappointing although we used the TM-friendly data layout and were able to significantly reduce the number of memory accesses managed by STM using knowledge about thread-local data structures. The overhead of general purpose STM systems seems too high for main-memory databases.

VII. Related Work

Optimizing the transaction processing for modern multi-core and in-memory database systems is a vibrant topic within the database community. In the context of H-Store/ VoltDB [7], [26] several approaches for automatically deriving a database partitioning scheme from the pre-defined workload were devised [11], [27] and methods of optimizing partition-crossing transactions were investigated [28]. The partitioning research focused on distributed databases, but is also applicable to shared memory systems. Partitioning the database allows for scalable serial transaction execution as long as the transactions do not cross partition boundaries, which in general is hard to achieve. In [29] a data-oriented transaction processing architecture is devised, where transactions move from one processing queue to another instead of being assigned to a single thread. The locking-based synchronization is optimized via speculative lock inheritance [30]. Ren et al. [24] found that the lock manager is a critical performance bottleneck for main memory database systems. They propose a more lightweight scheme, where, instead of locks in a global lock manager data structure, each tuple has two counters that indicate how many transactions requested read or write access.

In the Crescando project [31] a multi-transaction/query processing model is used that executes all transactions of one epoch within one cyclical scan of the database table. This approach is highly scalable and has predictable response times – however it is geared towards simple one tuple update transactions; multi-step transactions are not supported. Hyder [32], OctopusDB [33], and LogBase [34] are scale-out in-memory database developments that utilize a log-file structure in which successful transactional modifications are accumulated. The client's view of the database state is constructed from this shared, distributed log file.

In an earlier evaluation we showed that timestamp-based concurrency control has become a promising alternative to traditional locking [35]. Lomet et al. [36] and Larson et al. [8] recently devised multi-version concurrency control schemes that, like our approach, use a timestamp-based version control to determine conflicting operations. Unlike our proposal, their concurrency control is fully software-implemented, therefore it bears some similarity to software transactional memory [37].

Herlihy and Moss [6] proposed HTM for lock-free concurrent data structures. Shavit and Touitou [38] are credited for the first STM proposal. A comprehensive account on transactional memory is given in the book by Larus and Rajwar [39]. Due to the entirely software-controlled validation overhead, STM found little resonance in the database systems community – while, fueled by the emergence of the now common many-core processors, it was a vibrant research activity in the parallel computing community [40]. Now, that hardware vendors, such as Intel [14], [41] and IBM [15], [16], realize transactional memory support in hardware, it is time for the database community to exploit this development.

VIII. Summary

There are two developments—one from the hardware vendors, and one from the database software developers—that appear like a perfect match: the emergence of hardware transactional memory (HTM) in upcoming processors like Haswell, and main-memory database systems. The data access times of these systems are so short that the concurrency control overhead, in particular for locking/latching, is substantial and can be optimized by carefully designing HTM-supported transaction management. Even though transactions in main-memory databases are often of short duration, the limitations of HTM's read/write set management precludes a one-to-one mapping of DBMS transactions to HTM transactions.

We therefore devised and evaluated a transaction management scheme that transforms a (larger) database transaction into a sequence of more elementary, single tuple access/update HTM transactions. Our approach relies on the well-known timestamp ordering technique to "glue" the sequence of HTM transactions into an atomic and isolated database transaction. Our quantitative evaluation on a mainstream Haswell processor showed that our approach has low overhead and excellent scalability.

References

[1] M. Stonebraker and A. Weisberg, "The VoltDB main memory DBMS," *IEEE Data Eng. Bull.*, vol. 36, no. 2, 2013.

[2] A. Kemper and T. Neumann, "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots," in *ICDE*, 2011.

[3] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "SAP HANA database: data management for modern business applications," *SIGMOD Record*, vol. 40, no. 4, 2011.

[4] J. Lindström, V. Raatikka, J. Ruuth, P. Soini, and K. Vakkila, "IBM solidDB: In-memory database optimized for extreme speed and availability," *IEEE Data Eng. Bull.*, vol. 36, no. 2, 2013.

[5] P.-Å. Larson, M. Zwilling, , and K. Farlee, "The Hekaton memory-optimized OLTP engine," *IEEE Data Eng. Bull.*, vol. 36, no. 2, 2013.

[6] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *ISCA*, 1993.

[7] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker, "OLTP through the looking glass, and what we found there," in *SIGMOD*, 2008.

[8] P.-Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling, "High-performance concurrency control mechanisms for main-memory databases," *PVLDB*, vol. 5, no. 4, 2011.

[9] G. Weikum and G. Vossen, *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[10] H. Mühe, A. Kemper, and T. Neumann, "Executing long-running transactions in synchronization-free main memory database systems," in *CIDR*, 2013.

[11] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden, "Schism: a workload-driven approach to database replication and partitioning," *PVLDB*, vol. 3, no. 1, 2010.

[12] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[13] C. Cascaval, C. Blundell, M. M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee, "Software transactional memory: why is it only a research toy?" *Commun. ACM*, vol. 51, no. 11, 2008.

[14] "Intel architecture instruction set extensions programming reference," http://software.intel.com/file/41417, 2013.

[15] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. M. Michael, "Evaluation of Blue Gene/Q hardware support for transactional memories," in *PACT*, 2012.

[16] C. Jacobi, T. J. Slegel, and D. F. Greiner, "Transactional memory architecture and implementation for IBM system z," in *MICRO*, 2012.

[17] R. Rajwar and M. Dixon, "Intel transactional synchronization extensions," http://intel.com/go/idfsessions, 2012.

[18] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach (5. ed.)*. Morgan Kaufmann, 2012.

[19] M. J. Carey, "Modeling and evaluation of database concurrency control algorithms," Ph.D. dissertation, 1983.

[20] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[21] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Trans. Database Syst.*, vol. 17, no. 1, 1992.

[22] P. Felber, C. Fetzer, P. Marlier, and T. Riegel, "Time-based software transactional memory," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 12, 2010.

[23] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: ARTful indexing for main-memory databases," in *ICDE*, 2013.

[24] K. Ren, A. Thomson, and D. J. Abadi, "Lightweight locking for main memory database systems," *PVLDB*, vol. 6, no. 2, 2013.

[25] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *PPOPP*, 2008.

[26] E. Jones and A. Pavlo, "A specialized architecture for high-throughput OLTP applications," HPTS, 2009.

[27] A. Pavlo, C. Curino, and S. B. Zdonik, "Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems," in *SIGMOD*, 2012.

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

[29] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki, "Data-oriented transaction execution," *PVLDB*, vol. 3, no. 1, 2010.

[30] R. Johnson, I. Pandis, and A. Ailamaki, "Improving OLTP scalability using speculative lock inheritance," *PVLDB*, vol. 2, no. 1, 2009.

[31] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann, "Predictable performance for unpredictable workloads," *PVLDB*, vol. 2, no. 1, 2009.

[32] P. A. Bernstein, C. W. Reid, M. Wu, and X. Yuan, "Optimistic concurrency control by melding trees," *PVLDB*, vol. 4, no. 11, 2011.

[33] J. Dittrich and A. Jindal, "Towards a one size fits all database architecture," in *CIDR*, 2011.

[34] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi, "LogBase: a scalable log-structured database system in the cloud," *PVLDB*, vol. 5, no. 10, 2012.

[35] S. Wolf, H. Mühe, A. Kemper, and T. Neumann, "An evaluation of strict timestamp ordering concurrency control for main-memory database systems," in *IMDM*, 2013.

[36] D. B. Lomet, A. Fekete, R. Wang, and P. Ward, "Multi-version concurrency via timestamp range conflict management," in *ICDE*, 2012.

[37] D. Dice, O. Shalev, and N. Shavit, "Transactional locking II," in *DISC*, 2006.

[38] N. Shavit and D. Touitou, "Software transactional memory," in *PODC*, 1995.

[39] J. R. Larus and R. Rajwar, *Transactional Memory*. Morgan & Claypool Publishers, 2006.

[40] M. Herlihy and N. Shavit, *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[41] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, "Performance evaluation of Intel® transactional synchronization extensions for high-performance computing," in *SC*, 2013.