# To Lock, Swap, or Elide: On the Interplay of Hardware Transactional Memory and Lock-Free Indexing

Darko Makreshanski[1]*  Justin Levandoski[2]  Ryan Stutsman[3]

[1]ETH Zurich, [2,3]Microsoft Research
[1]darkoma@inf.ethz.ch, [2]justin.levandoski@microsoft.com, [3]rystutsm@microsoft.com

## ABSTRACT

The release of hardware transactional memory (HTM) in commodity CPUs has major implications on the design and implementation of main-memory databases, especially on the architecture of high-performance lock-free indexing methods at the core of several of these systems. This paper studies the interplay of HTM and lock-free indexing methods. First, we evaluate whether HTM will obviate the need for crafty lock-free index designs by integrating it in a traditional B-tree architecture. HTM performs well for simple data sets with small fixed-length keys and payloads, but its benefits disappear for more complex scenarios (e.g., larger variable-length keys and payloads), making it unattractive as a general solution for achieving high performance. Second, we explore fundamental differences between HTM-based and lock-free B-tree designs. While lock-freedom entails design complexity and extra mechanism, it has performance advantages in several scenarios, especially high-contention cases where readers proceed uncontested (whereas HTM aborts readers). Finally, we explore the use of HTM as a method to simplify lock-free design. We find that using HTM to implement a multi-word compare-and-swap greatly reduces lock-free programming complexity at the cost of only a 10-15% performance degradation. Our study uses two state-of-the-art index implementations: a memory-optimized B-tree extended with HTM to provide multi-threaded concurrency and the Bw-tree lock-free B-tree used in several Microsoft production environments.

## 1. INTRODUCTION

Recently, each generation of CPU has increased the number of processors on a chip, resulting in a staggering amount of parallelism. Transactional memory (TM) [17, 28] has been proposed as one solution to help exploit this parallelism while easing the burden on the programmer. TM allows for atomic execution of all of the loads and stores of a critical section, thereby relieving the programmer from thinking about fine-grained concurrency – a notoriously difficult engineering task. With the emergence of hardware transactional memory (HTM) shipping in commodity CPUs, we now have a promising transactional memory implementation that achieves great performance and is widely available.

In main-memory databases, multi-core scalability is paramount to achieving good performance. This is especially important at the access method (indexing) layer since it is the hot path for data manipulation and retrieval. For decades, fine-grained locking (latching)[1] protocols [13, 22] were the method for achieving index concurrency. In main-memory systems, however, locks (latches) are a major bottleneck since there is no I/O on the critical path [10, 15]. This has led to the design and implementation of "lock-free" indexing methods. Several commercial systems ship with lock-free indexes. For instance, MemSQL uses lock-free skip-lists [30], while Microsoft's Hekaton main-memory engine uses the Bw-tree [10, 27], a lock-free B-tree. To achieve lock-freedom, these designs use atomic CPU hardware primitives such as compare-and-swap (CAS) to manipulate index state. While efficient, lock-free designs are difficult to design and engineer since atomic instructions are limited to a single word, and non-trivial data structures usually require multi-word updates (e.g., B-tree splits and merges).

Until recently, lock-free index designs were the only way to achieve great multi-threaded performance in main-memory systems. Current designs employ HTM to seamlessly transform single-threaded implementations into high performance multi-threaded indexes by speculatively and concurrently executing operations [21, 24]. This approach greatly simplifies the design and implementation of the index; it provides multi-word atomic updates and pushes conflict adjudication into the hardware transaction. However, it is currently unclear if HTM is a "cure all" general-purpose technique that achieves the same performance goals as lock-free designs, and if not, what role HTM might play in the design and implementation of high-performance indexing.

This paper studies the interplay of HTM and lock-free indexing methods. Our study answers three fundamental questions. (1) Does HTM obviate the need for crafty lock-free index designs? Our answer is no. HTM has several limitations and pathologies – such as high abort rates due to capacity limits – that make it unattractive as a general solution for production systems that encounter a wide variety of data and workloads. (2) How does HTM differ from lock-free index designs? We find that HTM performs poorly in high contention scenarios. An HTM-based approach aborts readers that touch a "hot" data item, but lock-free designs do not block nor abort readers for any reason. (3) Given that lock-free designs are still relevant, can HTM help simplify lock-free design techniques while maintaining good performance? Our answer is yes. We find that using HTM as a building block to create a multi-word CAS instruction greatly helps simplify lock-free indexing design with

*Work performed while at Microsoft Research

[1]The database community uses the term "latch" to refer to what is known as a "lock" elsewhere. We use the term "lock" in this paper.

*Proceedings of the VLDB Endowment,* Vol. 8, No. 11
Copyright 2015 VLDB Endowment 2150-8097/15/07.

minimal performance impact. Based on our study, we also provide an HTM "wish list" of features we hope to see in the future.

We make the following contributions through an empirical evaluation of indexing methods based on HTM and lock-free designs.

- *An evaluation of HTM as a drop-in concurrency solution* (Section 3). We use HTM to provide concurrency in a "traditional" B-tree architecture using the cpp-btree [7]. We find that this approach provides great scalability for moderately sized data sets with small (8-byte) fixed-size key and payloads. However, for data that mirrors many real-world deployments (variable-length keys, large record counts), this HTM-based approach performs poorly; in some cases performing worse than serialized performance. We also find that HTM is not yet suitable for implementing fine-grained B-tree concurrency techniques such as lock-coupling [13].

- *A study of fundamental differences between HTM-based and lock-free index designs* (Section 4). We compare the HTM-based B-tree to the Bw-tree [27], a lock-free B-tree used in several Microsoft products including the Hekaton main-memory DBMS [10]. Lock-freedom requires extra mechanism (e.g., epoch protection for memory safety and maintenance due to copy-on-write). However, a lock-free approach never aborts readers (due to copy-on-write semantics). This feature is advantageous to read performance, especially when reading data with a high update rate. Meanwhile, an HTM-based approach that performs update-in-place will abort readers if their read set overlaps with a write, degrading read performance by up to 4x in the worst case.

- *A study of how HTM can help lock-free designs* (Section 5). Using HTM as a method for performing a multi-word CAS operation (MW-CAS) helps simplify lock-free design, since it allows atomic installation of operations that span multiple arbitrary locations (e.g., for B-tree page splits and merges). We find that the MW-CAS is a great application of HTM since it avoids many abort pathologies (since transactions are small) and provides good performance.

We end by providing a summary discussion of our findings and provide a wish list for HTM features we hope to see in the future. The rest of this paper is organized as follows. Section 2 provides an overview of HTM, the indexing architectures we evaluate, and our experiment environment. Section 3 provides an experimental evaluation of HTM used as a drop-in solution for multi-threaded scalability. We discuss the fundamental differences of HTM-based and lock-free designs in Section 4. Section 5 discusses how to simplify lock-free indexing design using HTM. Section 6 provides a discussion and our wish-list for future HTM features. Finally, Section 7 covers related work while Section 8 concludes the paper.

## 2. OVERVIEW

Our goal is to: (1) evaluate HTM as a drop-in solution for concurrency in main-memory indexing, (2) understand the differences of HTM-based and lock-free index designs, and (3) explore how HTM might help simplify state-of-the-art lock-free index designs. We do not aim to find the "fastest" main-memory index, nor do we propose a new index design for current HTM offerings. The rest of this section provides an overview of HTM, the B-tree implementations we evaluate, and our experiment environment.

```
atomic {
  Withdraw(A,X)
  Deposit(B,X)
}
```

**(a) Transactional Memory**

```
AcquireElided(Lock)
  Withdraw(A,X)
  Deposit(B,X)
ReleaseElided(Lock)
```

**(b) Lock Elision**

**Figure 1: Overview of Programming Models**

### 2.1 Hardware Transactional Memory

Developing high-performance parallel access schemes for main-memory data-structures is often a tedious and error prone task leading to deadlocks and race conditions. Transactional memory aims to ease this burden by delegating conflict detection and resolution from the developer to the system. Using transactional memory, a programmer specifies a set of CPU operations (Figure 1a) and the system ensures atomic and isolated execution of these operations, or aborts on conflict.

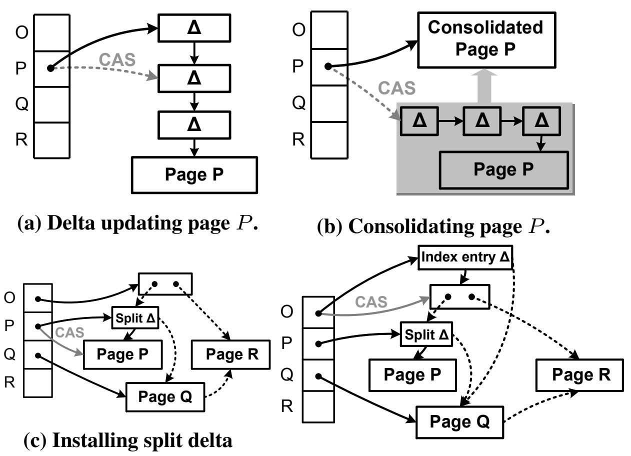
Hardware transactional memory (HTM) piggybacks on existing features in CPU micro-architectures to support transactions [17]. First, CPU caches can be used to store transaction buffers and provide isolation. Second, the CPU cache coherence protocol can be used to detect conflicting transactional accesses. With these modifications, CPUs can provide hardware support for transactional memory with low overhead to runtime. There are constraints, however, that limit HTM's usefulness. A primary constraint is that the read and write set of a transaction must fit in cache in order for it to be executed. Thus many properties may limit a transaction's size including: cache capacity, cache set associativity, hyper-threading, TLB capacity and others. Another constraint is on transaction duration. Many hardware events, such as interrupts, context switches or page faults, will abort a transaction (many of these limits were experimentally verified in previous work [24]). Furthermore, conflict detection is usually done at the granularity of a cache line. This may lead to cases of false sharing where aborts occur due to threads accessing and modifying separate items on the same cache line.

#### *2.1.1 Lock Elision*

These issues make it difficult for HTM implementations to guarantee that a transaction will ever succeed even if it is infinitely retried. Therefore, to guarantee forward progress a non-transactional fallback must be provided. One solution is to use lock elision [33] that guarantees progress by falling back to non-transactional lock-based synchronization. A nice feature of lock elision is that it is identical to programming with locks (Figure 1b). The difference between traditional lock-based synchronization is that lock elision first attempts to execute a critical section transactionally, and only if the transaction aborts will it execute the critical section by acquiring the lock. The benefit of lock elision is that it provides optimistic concurrency for programs that use simple coarse grain locks. The hardware ensures that as long as concurrent threads execute critical sections that do not have conflicting accesses, they can run in parallel, thus achieving performance similar to using fine-grained synchronization. In lock elision, the lock word needs to be included in the read set of a transaction, so that the transaction aborts when another thread acquires the lock (thus causing a conflict). Hence, once a thread resorts to non-transactional execution by taking the lock, all other concurrently executing transactions will abort, stalling overall progress.

#### *2.1.2 Intel TSX*

Starting with the Haswell CPU, Intel supports transactional memory, representing the first mainstream CPU to include such



**(a) Delta updating page $P$.** **(b) Consolidating page $P$.**

**(c) Installing split delta** **(d) Installing index entry delta**

**Figure 2: Bw-tree lock-free page update and split. Split is broken into two atomic operations using a CAS on the mapping table.**

functionality[2]. From what is known, it resembles the approach described above: a 32KB L1 8-way set associative cache buffers read and write sets and conflict detection is done at cacheline granularity. According to Intel [18], there is no guarantee that a transaction will eventually succeed even if it is infinitely retried. Therefore, lock elision or some other form of non-transactional fallback must be provided in all cases to ensure forward progress. Intel's transactional synchronization extensions (TSX) provide two interfaces.

**Hardware Lock Elision (HLE)** adds two new instruction prefixes (`XACQUIRE` and `XRELEASE`) meant to be used in conjunction with instructions that implement a lock, thus providing the lock elision functionality as explained above.

**Restricted Transactional Memory (RTM)** adds several instructions (`XBEGIN`, `XEND` and `XABORT`) to compose custom transaction logic. RTM allows the specification of a custom fallback code path in case a transaction aborts. One way of using RTM is to have more flexibility when implementing lock elision. For instance, a critical section may be transactionally retried a specified number times before resorting to acquiring the lock (instead of relying on Intel's default HLE implementation).

## 2.2 Index Implementations

We limit our study to main-memory B+-tree implementations due to its ubiquity in database systems and to make an apples-to-apples experimental comparison for HTM-based versus lock-free indexing. Such a comparison is currently hard with other high performance main-memory indexing methods. For instance, ART [23] exhibits great single-threaded performance, but does not have a multi-threaded lock-free counterpart.

### 2.2.1 *Traditional B+-Tree Design*

The cpp-btree [7] is a high performance memory-optimized B+-tree. It supports single-threaded access. The cpp-btree does not contain the fine-grained locking techniques and concurrency protocols (e.g., page locks or lock-coupling) common in many commercial B+-tree implementations. Thus it works well for our study, since we find that HTM is incompatible with fine-grained concurrency techniques (see Section 3). Plus, the promise of HTM is to seamlessly provide scalable multi-threaded performance to non-thread-safe data structures. Internally, the cpp-btree is a typical B+-tree. Data is stored within the leaf nodes, and internal index nodes contain separator keys and pointers to child pages.

[2]IBM Blue Gene/Q and System z mainframes include HTM but are high end specialized systems for HPC and scientific computing.

| | |
|---|---|
| Processor | Intel® Xeon® E3-1245 v3 ("Haswell")<br>3.4 GHz, up to 3.8 GHz turbo<br>4 cores, 8 hardware threads total |
| Caches | 64 B cacheline size<br>Private 32 KB L1D per core, 8-way set associative<br>Private 256 KB L2 per core, 8-way set associative<br>Shared 8 MB LLC, 16-way set associative |
| TLBs | L1-DTLB 64 4 KB, 32 2 MB, and 4 1 GB pages<br>L2-Combined TLB 1024 4 KB or 2 MB pages |
| DRAM | PC3-12800 DDR3 (800 MHz), 2 channels |
| OS | Windows® Server 2012 R2 |

**Table 1: Configuration of the machine used in experiments.**

### 2.2.2 *The Bw-tree*

The Bw-tree is a completely lock-free B+-tree, meaning threads never block for any reason when reading or writing to the index. It is currently shipping within a number of Microsoft products including SQL Server Hekaton [10] and Azure DocumentDB [12]. The key to the Bw-tree's lock-freedom is that it maintains a *mapping table* that maps logical page identifiers (LPIDs) to virtual addresses. All links between Bw-tree nodes are LPIDs, meaning a thread traversing the index must go through the mapping table to translate each LPID to a pointer to the target page.

**Lock-free updates.** The Bw-tree uses copy-on-write to update pages. An update creates a *delta record* describing the update and prepends it to the target page. Delta records allow for incremental updates to pages in a lock-free manner. We install the delta using an atomic compare-and-swap (CAS) that replaces the current page address in the mapping table with the address of the delta. Figure 2a depicts a delta update to page $P$; the dashed line represents $P$'s original address, while the solid line represents $P$'s new address. If the CAS fails (e.g., due a concurrent update to the page winning the CAS) the losing updater must retry. Pages are consolidated once a number of deltas accumulate on a page to prevent degradation of search performance. Consolidation involves creating a new compact, search-optimized page with all delta updates applied that replaces the old page version using a CAS (Figure 2b).

**Structure modifications.** A main difficulty in lock-free B+-tree design is that structure modification operations (SMOs) such as page splits and merges introduce changes to more than one page, and we cannot update multiple arbitrary pages using a single CAS. The Bw-tree breaks an SMO into a sequence of atomic steps; each step is installed using a CAS to a single page. We briefly describe a lock-free page split; page deletes are described elsewhere [27]. The split works in two phases and is based on the B-link design [22], as depicted at the bottom of Figure 2. We split an existing page $P$ by first creating a new page $Q$ with records from the upper half of $P$'s key range. Next, a "split delta" is installed on $P$ (Figure 2c) that logically describes the split and provides a side-link to the new sibling $Q$. We then post a (search key, PID) index term for $Q$ at parent $O$ with a delta record, again using a CAS (Figure 2d). In order to ensure that no thread has to wait for an SMO to complete, a thread that encounters a partial SMO in progress will complete it before proceeding with its own operation (see Section 5). The lock-free design is very efficient, however this performance comes at a cost: it is very difficult to design and implement non-trivial lock-free data structures such as a B+-tree.

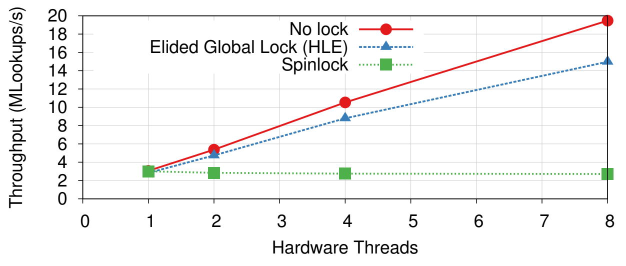


**Figure 3: Read-operation throughput against an in-memory B+-Tree as the number of hardware threads is increased when no lock, a global elided lock, and a spinlock are used for synchronization.**

## 2.3 Experimental Environment

Our experiments compare and contrast the cpp-btree and Bw-tree implementations under several workloads in order to highlight the fundamental differences between using the HTM and lock-free approaches to concurrent B-Trees. Each experiment pre-loads one of the two types of trees. Then, a fixed number of threads are each assigned to continuously perform either lookup operations or update operations. We vary record count, record size, key size, access skew, and lookup/update thread count to highlight the pathologies of each structure. We also compare several approaches for HTM conflict handling and lock-free techniques against basic spinlocks. Our results primarily examine operation throughput and hardware transaction abort rates.

Unless otherwise stated, workloads focus on trees of 4 million records either using 8-byte keys and 8-byte payloads (61 MB total) or 256-byte payloads (1.2 GB total). Experiments use 4 hardware threads issuing lookups and 4 hardware threads issuing updates.

Table 1 describes the machine used for all of the experiments presented in this paper. This Haswell generation CPU is equipped with Intel (TSX), which represents the first wide deployment of HTM in commodity machines. New features, pipeline changes, and cache hierarchy changes may significantly influence the expressiveness and performance of HTM. For example, AMD published a design for its Advanced Synchronization Facility (ASF) [6] in 2010 that provides a different HTM interface, but ASF-equipped CPUs are not yet available.

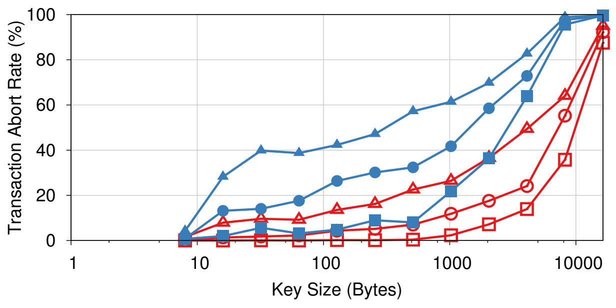
## 3. LOCK-BASED INDEXING WITH HTM

B-Trees are good candidates to take advantage of hardware lock elision; many state-of-the-art B-Tree implementations use spin-locks or read-write locks for multi-core parallelism [13]. HTM lock-elision requires little effort and overhauls existing lock-based data structures with optimistic synchronization often just by replacing the underlying synchronization library.

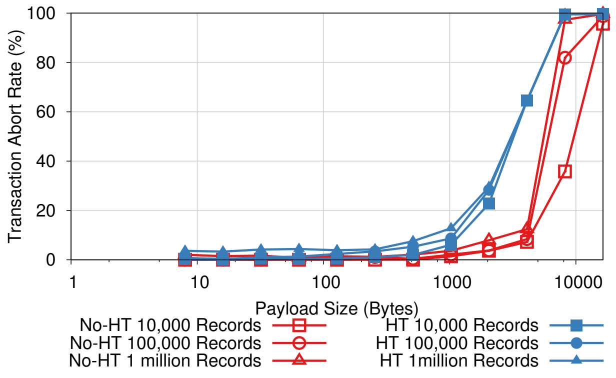
This section explores the potential of lock elision for B-Trees by looking at how it fits with two lock-based synchronization schemes: a single global lock and fine-grained lock-coupling. First, we will see that HTM effectively parallelizes simple B-Trees that use a single (elided) global lock, though with some important limitations. Finally, we find that Intel's current HTM interface is incompatible with lock-coupling.

## 3.1 Global Lock

Hardware lock elision promises the simplicity of coarse-grain locks with the performance of fine-grain locks or lock-free programming. Therefore, a natural approach is to wrap every B-Tree operation in a critical section protected by one global lock. Then, lock acquisition can be hardware elided to provide inter-operation



**(a) Abort rate versus key size with fixed-size 8-byte payloads.**



**(b) Abort rate versus payload size with fixed-size 8-byte keys.**

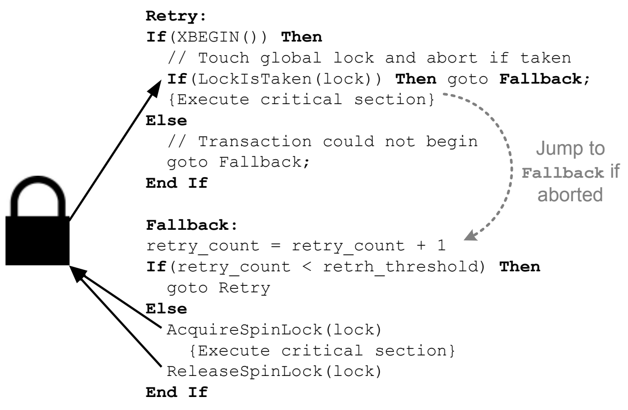
**Figure 4: Transaction abort rates for various key and payload sizes.**

parallelism. This is the current state-of-the-art in HTM-enabled indexing [21, 24].

We evaluated this approach using an existing B+-Tree implementation (cpp-btree [7]) optimized for single-threaded in-memory access and parallelized it using a global elided lock. To understand the best-case potential of the approach and to understand its overheads, Figure 3 compares read operation throughput against this tree using an elided lock, a conventional spinlock, and no lock at all (which represents ideal performance and is only safe under a read-only workload). The tree is pre-filled with 4 million 8 byte keys and payloads, and the number of threads driving a read-only uniform-random access workload is varied. Our results confirm earlier studies of small workloads with fixed-length keys and payloads [21]: HTM provides high throughput with little effort. Using basic HLE is only about 33% slower than unsynchronized access.

### 3.1.1 *Effect of Key and Payload Sizes*

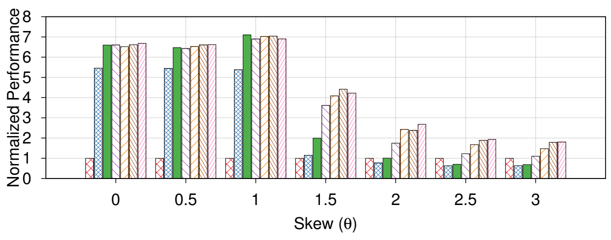
Unfortunately, the great results of the global elided lock on a simple workload do not hold in general. The first complicating issue stems from capacity limits on hardware transactions. Under the hood, hardware must track the read and write set of all the cachelines accessed by a thread in a transaction. Haswell's HTM implementation is not fully described by Intel, but it leverages its 32 KB L1 cache to buffer a transaction's writes and to track its read and write set. Any eviction of a cacheline from the transaction's write set will result in an abort. Hence, no transaction can write more than can fit in L1. In fact, associativity compounds this; for example, Haswell's L1 is 8-way associative, and Intel states that writes to 9 distinct cachelines that map to the same cache set will result in an abort [18]. Since read sets are also tracked in L1 they suffer from similar capacity constraints, though Intel indicates read sets may be protected by an unspecified second-level cache (potentially



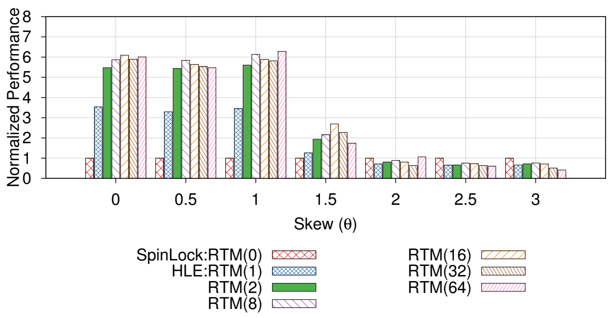
```
Retry:
If(XBEGIN()) Then
  // Touch global lock and abort if taken
  If(LockIsTaken(lock)) Then goto Fallback;
  {Execute critical section}
Else
  // Transaction could not begin
  goto Fallback;
End If

Fallback:
retry_count = retry_count + 1
If(retry_count < retrh_threshold) Then
  goto Retry
Else
  AcquireSpinLock(lock)
    {Execute critical section}
  ReleaseSpinLock(lock)
End If
```

**Figure 5: Lock elision using RTM with configurable retry threshold.**



**(a) 4 million records with 8-byte keys and 8-byte payloads**



**(b) 4 million records with 8-byte keys and 256-byte payloads**

**Figure 6: Update Performance when workload Skew and number of transactional attempts per transaction are varied.**

the load unit as in described [6]). Finally, hyper-threading can also induce capacity-related aborts, since hardware threads on a common core share an L1 cache and other resources.

Overall, these capacity constraints make HTM challenging to use when parallelizing B-Trees; many of the properties that determine the HTM abort rate for a given tree may not be known until runtime. A tree's key size, payload size, total size, and address access patterns all affect performance. For example, tree size is problematic because the number of nodes accessed during a traversal grows logarithmically with tree size, which increases the required transaction size as well. In the end, these HTM capacity constraints mean trees with large keys and/or large payloads do not parallelize well when using a global elided lock.

To investigate the impact of these limitations in practice we measured the percentage of transactions that abort due to capacity constraints for read-only tree traversals while varying the key and payload sizes. We pre-populated the tree with varying number of records and ran the workloads with hyper-threading both on and off. We measured the transaction abort rate which is correlated with the achieved parallelism. If the abort rate is close to 0% all operations are executed in a transaction and maximum parallelism is achieved (similar to the HLE performance trend in Figure 3). If the abort rate is close to 100% lock-elision always falls back to acquiring the global lock leading to no parallelism (similar to the spin lock performance trend in Figure 3).

The results in Figure 4 confirm that with this simple approach, even trees with relatively small keys and payloads cannot always parallelize. With Haswell's HTM almost all transactions abort with payloads larger than a few kilobytes (Figure 4(b)), even though the transaction buffers are stored in a 32 KB cache. This shows the limiting effect of the 8-way cache set associativity and cache sharing with hyper-threading. Key size is even more severely constrained, since a single transaction encounters many keys during each lookup. Abort rates can climb to 40% with just 64 byte keys in a 1 million record tree (Figure 4(a)).

### *3.1.2 High-contention Performance*

Hardware capacity is just one source of pain for employing HTM; another difficulty lies in predicting performance due to transactional conflicts. The prior sections avoided conflicts and isolated the impact of hardware capacity by using a read-only workload. In practice, HTM is only needed when a workload has potentially conflicting updates. When a transaction aborts due to a true data conflict performance is naturally impacted. However, there are two other problematic ways that transaction aborts hurts performance. First, speculation is not free: transaction startup overhead and the resources it consumes while running can result in wasted work. Second is the so-called *lemming effect* [11]. Haswell's HTM requires all transactions to eventually fall back to using a lock when transactions abort, since it makes no forward progress guarantees. When a transaction falls back and acquires the lock, all other transactions in the critical section abort and cannot restart until the lock is released. The effect is that execution is fully serialized until the lock is released – even if the other transactions operate on non-conflicting cache lines. Consequently, concurrency is aggressively and often unnecessarily restricted. This problem becomes apparent in high skew workloads where execution is almost entirely serialized even for transactions that operate on low-contention values.

One way to mitigate the lemming effect is to have transactions retry more than once before falling back to acquire a lock. Retrying a contentious transaction might be costly, but the cost of acquiring the lock and serializing execution is (usually) worse. In contrast to HLE, Haswell's RTM instructions provide a flexible interface that allows custom code to be executed when a transaction aborts. As an optimization, Intel suggests using RTM to retry transactional execution of a critical section multiple times before resorting to acquiring the lock [18]. Figure 5 provides a schematic for how to perform lock-elision with retry using the RTM instructions.

Figure 6a shows the promise of this approach by comparing performance as workload skew and per-transaction optimistic attempt count are varied. Here, the B-Tree is pre-populated with 4 million items of 8-byte keys and payloads; 8 threads execute a workload with 50% lookups and 50% updates. The workload is Zipfian distributed with a skew parameter ($\theta$ [14]) varied between 0 to 3 (uniform random through extremely high skew), and the number of transactional attempts per transaction is varied from 0 to 64, where 0 corresponds to synchronization with a spin-lock. Interestingly, a retry count of 1 exhibits performance that corresponds to Intel's default lock-elision implementation (abbr. HLE). The performance metric is throughput normalized to the throughput of a spin-lock.

The results show that as workload skew increases the performance of lock-elision drops sharply. At some point lock-elision performs even worse than spin-locks, achieving no parallelism from the multi-core hardware. Increasing the number of transactional attempts delays the performance cliff, and leads to a more graceful degradation of performance as skew increases.

### *3.1.3 Optimal Number of Transactional Attempts*

Unfortunately, moving to larger retry limits indefinitely does not work in general; retrying doomed transactions has a cost and blindly retrying transactions can lead to performance that is worse than serialized execution with a spinlock. The root of the problem is that with the existing HTM conflict resolution strategies (seemingly "attacker-wins" [11]) transactions may continuously abort one another without any guarantee of global progress. The result is that for a set of concurrent transactions, it may be possible that none of them commit: a situation worse than using a spinlock.

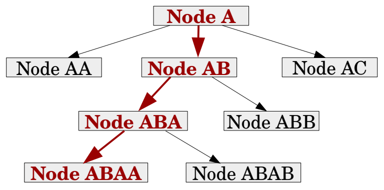
One factor that contributes to this pathology is the position of the first conflicting access within transactions. In the previous experiment, conflicting accesses occur when the 8-byte payload is updated at the end of a tree traversal operation, which is late within the transaction. If the size of a payload is significantly larger than 8 bytes, updating it becomes a longer operation, which shifts the first conflicting access earlier within the transaction. This increases the probability that retries will repeatedly interfere with one another. For example, a thread $T_1$ may transactionally find a value in a tree and update its payload in place. In the meantime, a thread $T_2$ may attempt an update of the same payload, aborting $T_1$. While $T_2$ is still updating the payload, $T_1$ may have restarted; if updating the payload takes a long time compared to index search, then $T_1$ may cause an abort of $T_2$'s transaction. Even with just these two threads there is no guarantee the update will ever be completed.

Figure 6b highlights this effect; it re-runs the experiment from Figure 6a but with 256-byte (instead of 8-byte) payloads. It shows that as the number of transactional attempts increases, the performance reaches a peak and then drops (even below serialized performance). The optimal number of transactional attempts depends on workloads and varies highly. In absence of a smarter solution that chooses the number of transactional attempts dynamically depending on the workload, for all future experiments we fix the number of retry attempts to 8, which provides good general performance and avoids collapse under the most realistic skews.

## 3.2 Lock-coupling

A single global lock works when HTM is used for concurrency, but it is prohibitive without HTM. Practical high-concurrency, high-performance B-Trees instead rely on fine-grained locking. Efficient fine-grained locking on B-Trees is notoriously hard to get right. Many techniques exist, but lock-coupling is one of the most widely used approaches (see [13, 36] for an overview). In lock-coupling a pair of locks are held as a worker traverses pages: one on a "source" page and another on a "target" page. As the traversal proceeds, a lock on the target page in the traversal is first acquired and only afterward the lock on the source page is released (Figure 7b). This careful ordering avoids races between reading the source page and accessing the target page (for example, this prevents a target page from disappearing as a traversal moves to it).

Ideally, lock-coupling could also be applied to parallelize B-Trees using HTM. This has the potential to improve both capacity and conflict aborts. Transactions could maintain a smaller, constant-sized read set as they traverse down the tree, and they would avoid conflicts on higher levels of the tree as they work downward. Significantly, these constant-sized read sets would effectively eliminate the effect of tree size on abort rates.



**(a) Sample B-Tree**

```
Lock(A)
    Read(A)
Lock(AB);Unlock(A)
    Read(AB)
Lock(ABA);Unlock(AB)
    Read(ABA)
Lock(ABAA);Unlock(ABA)
    Read(ABAA)
Unlock(ABAA)
```

**(b) Lock-coupling**

```
BeginTransaction()
    Read(A)
    Read(AB)
  Release(A)
    Read(ABA)
  Release(AB)
    Read(ABAA)
CommitTransaction()
```

**(c) Transactional Lock-coupling**

**Figure 7: Transactional Lock-coupling Approach**

Unfortunately, Haswell's HTM interface is currently too restrictive to support lock-coupling. HTM transactions on a single thread can be nested but cannot be overlapped as lock-coupling's non-two-phase pattern requires. To make this work, TSX would need to give threads control over their read set. For example, AMD's proposed ASF instruction set [6] includes "release" instruction that removes a specified item from the read set of the executing transaction (see Figure 7c); however, AMD has not yet released a CPU with ASF support. In the end, without such support, lock-coupling with an elided page lock would perform no better under Haswell's HTM than using a single elided global lock.

## 3.3 Summary

Using lock elision (via HLE or RTM) works well for simple B-tree indexes with predictable cache footprints. However, abort rates are sensitive to many parameters some of which may not be known until runtime; data structure size, access skew, access rate, physical address patterns, and false sharing can all play a role. Second, in some cases, using HTM as a global locking approach can result in performance worse than single-threaded execution (or serialized execution with a spin lock) due to wasted work on aborts and transaction initialization overheads. Finally, existing commodity HTM interfaces are incompatible with fine-grained locking techniques like lock-coupling, which could otherwise reduce conflicts and eliminate the dependency between aborts and data structure size.

Clearly, in its current form HTM is not yet a robust solution for achieving multi-threaded scalability within main-memory B-trees. In the next section, we compare and contrast an HTM-based B-tree to a state-of-the-art lock-free B-tree design.

## 4. HTM VS LOCK FREE APPROACHES

It is tempting to view HTM as hardware-accelerated lock-free programming, but the approaches are fundamentally different and each makes tradeoffs. HTM provides an atomic update-in-place abstraction; lock-free programming techniques use limited (often single word) atomic updates in place and must compose larger operations using copy-on-write. This has important ramifications that give each approach a different set of costs and pathologies.

| | Read-Write Conflict | Write-Write Conflict |
|---|---|---|
| **HTM** (Atomic update-in-place) | Abort/retry | Abort/retry |
| **Lock-free** (Copy-on-write and publish) | OK | Retry |

**Table 2: HTM aborts whenever a read cacheline is concurrently modified, but lock-free techniques generally do not interfere with readers.**

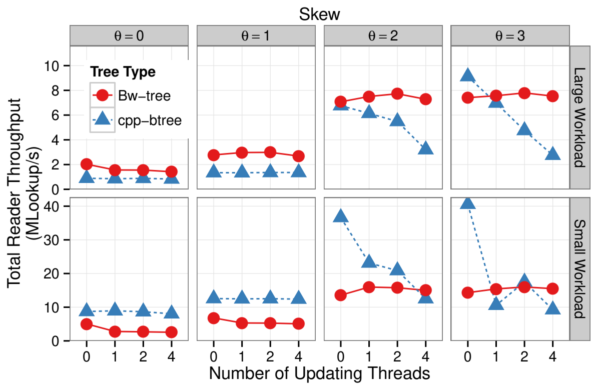


**Figure 8: Impact of concurrent writers on readers for two different types of B-Trees. Reader performance suffers under concurrent updates with the RTM-based cpp-btree, whereas readers are immune to updates with the lock-free Bw-tree.**

## 4.1 The Costs of HTM under Contention

Because the two approaches perform updates differently, the most significant performance differences between them are due to the impact updates have on concurrent operations, specifically, how writes impact concurrent reads. Table 2 explains: write-write conflicts cause retries and wasted work under both approaches; however, lock-free techniques avoid wasted work under read-write conflicts that HTM cannot. By avoiding update-in-place, lock-free updates via pointer publishing never disrupt reads. Old value remains intact for concurrent reads while later reads find new version(s).

Figure 8 explores this effect. It shows the total read throughput for four reader threads with both B-Tree implementations as the number of threads performing updates and workload skew is varied. In this experiment, each tree consists of 4 million records. The small workload uses 8 byte keys and 8 byte payloads; the large workload uses 30 to 70 byte variable length keys and 256 byte payloads. Keys are chosen according to a Zipfian distribution as described in [14].

As expected, for low contention workloads ($\theta = 0$, which is a uniform random access pattern) neither the HTM-enabled cpp-btree or the lock-free Bw-tree are significantly impacted by the presence of threads performing writes. However, for high contention workloads ($\theta \geq 2$) the reader throughput that the cpp-btree can sustain begins to drop. The Bw-tree, in contrast, gets a double benefit from the contentious write heavy workloads. First, readers benefit from high access locality, since writers do not hurt readers. Second, writers actually benefit readers: readers can read recent writes from cache (in addition to the benefit the locality skew gives).

## 4.2 The Overheads of Lock-Freedom

Lock-free techniques can reduce the impact of concurrent writers on readers; however, this benefit comes with three subtle costs: the need for a garbage-collection mechanism for memory safety, the need for indirection for atomic updates, and the cost of copy-on-write. These costs are highly intertwined: tradeoffs for each influence the cost of the others.

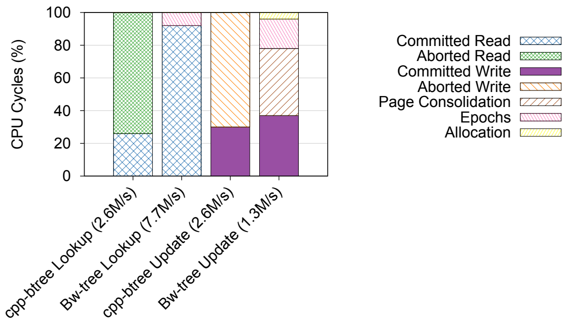


**Figure 9: Breakdown of CPU cycles spent by each core performing lookups and updates in the cpp-btree and Bw-tree under the "large" workload of Figure 8 with skew $\theta = 3$.**

### 4.2.1 *Lock-Free Memory Reclamation Overheads*

Each object unlinked from a lock-free structure may continue to be accessed by threads that hold references to it. This is a key benefit, but it also means that for safety the system must track when threads no longer hold references to an unlinked object. Otherwise, prematurely reusing an object's memory could result in threads reading corrupted data.

For example, the Bw-tree uses an epoch mechanism [27] that tracks when each thread is accessing the tree. Threads must place themselves on a list for the current epoch whenever they access the tree, and they only remove themselves after they drop all references to tree internal objects. When a page is unlinked it is placed on a queue in the current epoch until all threads have moved on to later epochs; after this point, the page is safe for reuse and may be freed.

Different schemes for providing this "pointer stability" have different performance tradeoffs, but all add overhead and most nontrivial lock-free data structures require some such mechanism. Figure 9 shows the overhead incurred in the Bw-tree due to epochs for the "large workload" of Section 4.1 with skew $\theta = 3$; cores spend 8% and 18% of their cycles on epoch protection for lookups and updates, respectively. All measurements of the Bw-tree in this paper include the overhead of its epoch protection.

### 4.2.2 *Additional Indirection*

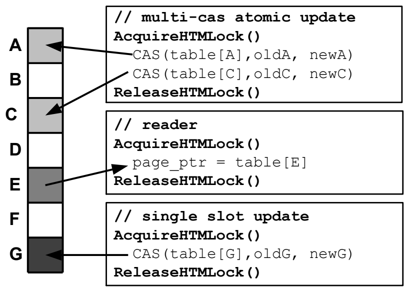
Most lock-free structures must also pay another subtle cost: lock-freedom influences the in-memory layout of structures. Lock-free data structures are carefully crafted to group together updates that need to be atomic, which are then published via a single compare-and-swap via a pointer-like field. This fuses synchronization and memory layout, and it forces additional indirection. Bw-tree's mapping table is an example of this; each page access must go through indirection via the mapping table, which effectively doubles the number of cachelines accessed when traversing the tree. In our experiences with the Bw-tree and more recent work on a lock-free transactional engine [26], we have found that this indirection is not onerous; it often naturally coincides with variable-length lists or data, which already require indirection to be handled efficiently.

This extra indirection also puts pressure on memory allocators, since updates cannot use the memory locations of the old values. In our experience, it may be necessary to resort to specialized and/or lock-free memory allocators to compensate for this.

### 4.2.3 *Copy Overhead*

Finally, lock-free structures also incur the cost of the additional copying required when using paged copy-on-write semantics. The costs of copy-on-write are not simple to characterize; page size, access skew, the uniformity of payload sizes, and the cost of allocation all play a role in the effectiveness of copy-on-write. For efficiency, data structures may need to amortize the cost of each full page copy over several lookups and updates.

Interestingly, copy-on-write can even improve performance in some cases. For example, Figures 8 and 9 show that copy-on-write improves the performance of lookups that are concurrent with updates, and Bw-tree's delta updates and blind writes can improve the performance of writers as well.



```
// multi-cas atomic update
AcquireHTMLock()
CAS(table[A],oldA, newA)
CAS(table[C],oldC, newC)
ReleaseHTMLock()

// reader
AcquireHTMLock()
page_ptr = table[E]
ReleaseHTMLock()

// single slot update
AcquireHTMLock()
CAS(table[G],oldG, newG)
ReleaseHTMLock()
```

**Figure 10: Global HTM lock protecting the page indirection mapping table. All reads and writes elide the lock before accessing the table.**

## 5. SIMPLER LOCK-FREEDOM VIA HTM

The evaluation in Section 4 shows that there are performance benefits to lock-free indexing designs. However, the gains come at a cost: it is very difficult to architect and build complex lock-free data structures such as a B+-tree. This section experimentally evaluates a middle ground that uses HTM to ease the difficulty of building lock-free indexes without sacrificing performance. We first discuss the difficulty in lock-free indexing design. We then evaluate approaches for using HTM to implement a multi-word compare-and-swap (MW-CAS) for use within the Bw-tree to atomically install multi-page structure modifications (split and merge). While we discuss the Bw-tree specifically, the approach applies more generally to data structures that uses indirection for lock freedom.

## 5.1 Lock-Free Programming Difficulties

A main difficulty with lock-free index designs stems from reliance on atomic CPU primitives – usually CAS or fetch-and-increment – to make state changes to the data structure. These instructions work at the granularity of a single word (generally a 64-bit word on today's CPU architectures). Life is easy if all operations on a data structure require only a single atomic instruction. However, this is rarely the case for non-trivial data structures. Difficulty arises when operations must span multiple atomic operations. For example, in the Bw-tree structure modification operations (SMOs) such as page splits and merges span multiple atomic steps. Splits involve *two* atomic steps, each installed using a CAS: one to install the split to an existing page $P$ with a new sibling $Q$, and another to install the new search key and logical page pointer for $Q$ at a parent page $O$ (see Figure 2). Page merges involve *three* steps: one to mark a page $P$ as deleted, the second to update $P$'s sibling $Q$ to merge any of $P$'s existing keys, and the third to delete $P$'s id and search key from the parent $O$.

When operations span multiple atomic steps, one problem that occurs is handling the case when other threads observe the operation "in progress". In lock-based designs, safety in such situations is guaranteed by a thread stalling on a lock set by a thread performing an SMO [36]. In lock-free scenarios this process is more difficult: one must handle both (a) how to detect such conflicts without locks and (b) what to do after detecting the conflict without blocking nor corrupting data. The Bw-tree design addresses these two issues as follows. A worker detects an in-progress split by finding that a page's boundary keys do not contain the search key; an in-progress delete is detected by traversing to a page containing a "page delete" delta record. Once any Bw-tree thread runs into an in-progress SMO, it helps along to complete the SMO before completing its own operation. This "help-along" protocol is necessary in many lock-free designs for performance, to guarantee progress, and correctness (e.g., to serialize SMOs that "run into" each other [27]). An alternate strategy would have a thread simply retry upon encountering an SMO; this is essentially a form of spinning to wait for the SMO. In this case the wait might be long: the SMO thread may get scheduled out by the OS, or "lose its way" in the tree and have to reposition itself to finish the SMO (e.g., when going up to the parent and finding it split). The "help-along" protocol guarantees that an SMO completes in a timely manner.

Lock-free designs also lead to subtle race conditions that are extremely difficult to reason about, engineer around, and debug. A prime example in the Bw-tree is that simultaneous splits and merges on the same page could collide at the parent, and without care lead to index corruption. This happens, for instance, when a thread $t_1$ sees an in-progress split of a page $P$ into $P'$ and $Q$ and attempts to help along by installing the new index term for $Q$ at the parent $O$. In the meantime, another thread $t_2$ could have deleted $Q$ and already removed its entry at $O$ (which was installed by another thread $t_3$). In this case $t_1$ must be able to detect the fact that $Q$ was deleted and avoid modifying $O$. As an additional anecdote, it took roughly three months to design and test a correct page delete protocol in the Bw-tree. There are several more difficult races and issues we encountered when building the Bw-tree, but this single example hopefully sheds light on the type of difficulties encountered when building lock-free infrastructure. While we use examples from building the Bw-tree, other practitioners surely have similar war stories.

In the rest of this section we aim to ease the burden of building lock-free data structures. We describe an approach that uses HTM to simplify lock-free designs by building a high performance multi-word compare and swap to compose operations that would otherwise require a series of multiple atomic steps.

## 5.2 Multi-Word CAS using HTM

As we see it, a major pain point for the Bw-tree lock-free design is handling operations that span multiple atomic operations on arbitrary locations in the indirection mapping table. The ability to perform an atomic multi-word compare-and-swap (MW-CAS) on arbitrary memory locations would greatly simplify the design and implementation of the Bw-tree. Fortunately, this is exactly what HTM provides: a method to atomically update arbitrary words.

Using HTM to implement an MW-CAS fits the sweet spot for current HTM implementations, especially on Intel's Haswell. To elaborate, most applications would only require the MW-CAS to span a handful of words. For instance the Bw-tree needs *at most* a triple word MW-CAS to install a page delete. This means that MW-CAS transactions will not suffer aborts due to capacity constraints even with today's stringent HTM limits. Further, MW-CAS transactions are short-lived (involving only a load, compare, and store for each word) and would avoid interrupts that spuriously abort longer running transactions [24].

## 5.3 Use of a Global Elided Lock

One approach we evaluate places a global elided lock over the Bw-tree indirection mapping table. To implement the global lock, we use the RTM-based approach discussed in Section 4, since it performs much better than default HLE. If a thread cannot make progress after its retry threshold, it acquires the global exclusive lock and executes the critical section in isolation. The rest of this section provides an overview of this approach.

### *5.3.1 Writes*

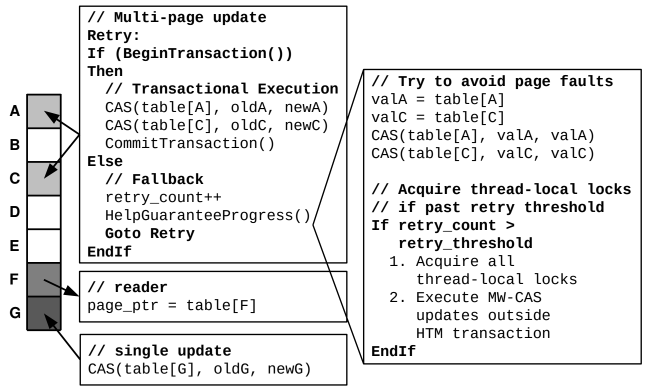
Writes to the mapping table bracket one or more compare and swaps within the acquisition and release of the HTM lock. Figure 10 depicts a multi-slot update to pages $A$ and $C$ in the mapping table along with another thread updating a single page $G$. Executing each CAS under the HTM lock ensures that if a conflict is detected, all changes to the mapping table will be rolled back; the transaction will eventually succeed on a retry (possibly acquiring the global lock if necessary). To avoid spurious aborts, we allocate and prepare all page data *outside* of the MW-CAS operation to avoid HTM aborts, e.g., due to shared access to the allocator or accessing random shared cache lines. For example, a thread installing a split in the Bw-tree would allocate and prepare both the split delta *and* the index term delta for the parent *before* performing the MW-CAS to install its two changes to the mapping table.

### *5.3.2 Reads*

**Bracketing index traversals.** We attempted to bracket multiple reads within a transaction representing an index traversal from root to leaf. This approach completely isolates index traversals from encountering in-progress SMOs. However, the approach increases the abort rate, since the transaction must contain logic to access page memory and perform binary search on internal index nodes. As discussed in Section 3, success rates for transactions at such a coarse grain depend on independent factors (e.g., page size, key size). The performance of this approach suffered in many of our experiments due to such spurious aborts.

**Singleton read transactions.** We instead evaluate an approach that places each read of a single 8-byte mapping table word in its own transaction. This avoids aborts due to cache capacity, since transactions only contain a single read. However, readers can encounter an "in-progress" SMO operation, for instance, when a split is installed between the time a reader access the "old" parent (without the split applied) and the "new" child (with the split applied). While we need to detect such cases, code to handle such cases becomes much simpler: we can just retry the traversal from a valid ancestor node. This approach guarantees the reader that the MW-CAS writes are atomic, thus SMOs are installed atomically. Therefore readers do not need to worry about complex situations such as helping along to complete an SMO, what to do when running into multiple in-progress SMOs that collide, etc.

**Non-transactional reads.** We also evaluate an approach that performs all reads non-transactionally. The advantage of this approach is that readers avoid setup and teardown time for hardware transactions. However, readers are *not* guaranteed that they will see SMOs installed atomically. This could happen when a reader observes writes from a transaction executing within its locked fallback path. Essentially, the reader can observe an index state where the writer is "in between" mapping table writes; this would not happen if a read were done inside a transaction or while holding the fallback lock. The result is that there is no reduction in code complexity, since non-transactional accesses must be prepared to help along to complete an SMO (or spin waiting for it to finish). Another issue is that the writer must be carefully order its stores in its fallback path, since non-transactional reads will see these stores in the order they occur. While this is fine for lock-free structures like the Bw-tree that already order SMO writes correctly, it is an issue to be aware of in the general case.



```
// Multi-page update
Retry:
If (BeginTransaction())
Then
  // Transactional Execution
  CAS(table[A], oldA, newA)
  CAS(table[C], oldC, newC)
  CommitTransaction()
Else
  // Fallback
  retry_count++
  HelpGuaranteeProgress()
  Goto Retry
EndIf
```

```
// reader
page_ptr = table[F]
```

```
// single update
CAS(table[G], oldG, newG)
```

```
// Try to avoid page faults
valA = table[A]
valC = table[C]
CAS(table[A], valA, valA)
CAS(table[C], valC, valC)

// Acquire thread-local locks
// if past retry threshold
If retry_count >
   retry_threshold
  1. Acquire all
     thread-local locks
  2. Execute MW-CAS
     updates outside
     HTM transaction
EndIf
```

**Figure 11: Multi-word CAS based on retry.**

## 5.4 Infinite Retry

We also evaluate an approach that removes both singleton reads and writes from HTM transactions. We call this the "infinite retry" approach. The approach takes advantage of the fact that singleton reads or updates (that are non-transactional) will still trigger the cache coherence protocol for their target cache lines. Since HTM transactions piggyback off this protocol, the multi-word CAS running within the transaction will see the changes to its write set. Unlike the non-transactional read approach discussed previously, this approach maintains the property that readers see atomic installation of MW-CAS writes. We also discuss methods to help guarantee progress of the transactional writes, in order to avoid spurious aborts as well as starvation due to continued data conflict with singleton (non-transcational) reads and writes.

### *5.4.1 Reads and Updates*

Figure 11 outlines the approach (for now, focus on the left-hand side of the figure). Singleton mapping table reads and updates do not operate within an HTM transaction, as depicted by the reader to slot $F$ and the update to slot $G$. Only multi-slot updates (the multi-slot update to $A$ and $C$) operate within a hardware transaction and execute the MW-CAS. The MW-CAS will abort if conflicting with a singleton read or update, detecting the conflict through the cache coherence protocol. The MW-CAS *continuously* retries the transaction in this case. Effectively, this places the MW-CAS at a lower priority compared to the singleton reads/updates, since they can abort the MW-CAS transaction, but not vice versa.

The MW-CAS cannot fall back on a single elided lock protecting the mapping table since the singleton reads/updates are not aware of the lock. One way around this is to simply let the transaction fail after it has retried a number of times; for the Bw-tree this effectively means abandoning an SMO, which will eventually be retried later by another thread (and hopefully succeed). However, if progress is absolutely necessary, the MW-CAS has to retry an infinite number of times until the transaction succeeds (thus the name of the approach). This means the MW-CAS has a chance of starvation, however this is true in general for all lock-free data structures.

### *5.4.2 Helping to Guarantee Progress*

Unfortunately, infinitely retrying a transaction does not guarantee an MW-CAS will succeed, since Intel does *not* guarantee a hardware transaction will *ever* succeed [18] (whereas for a single-word CAS, there is always a winner). This section discusses some approaches to help guarantee progress of this approach while trying to avoid a single elided lock. The right hand side of Figure 11 outlines these approaches.

**Capacity and time interrupts.** Since an MW-CAS transaction is short and touches a small number of cache lines, it avoids the likelihood of spurious aborts due to capacity constraints or interrupts due to long-running transactions. However, these are not the only reasons for aborts with Intel's current HTM implementation.

**Avoiding memory page faults.** When a transaction encounters a page fault it always aborts. Worse, speculative transactional execution suppresses the page fault event, so retrying the transaction speculatively will always fail without some outside help. Intel admits that synchronous exception events, including page faults, "are suppressed as if they had never occurred [18]." We confirmed this behavior on Haswell. Running a single transaction that performs a single access to a not-present page always aborts if a fallback lock is not used – the OS never receives the page fault event.

As a result, when omitting a fallback lock, a fallback code path must at least pre-fault the addresses that the transaction intends to access. Generally, the mapping table should be present in memory, but the correctness and progress of the system should not depend on it. The right hand side of Figure 11 shows how to safely induce the page faults. On the fallback path, the MW-CAS reads its target words (in this case mapping table slots $A$ and $C$) and performs a CAS for each of these words to assign the target word the same value as was just read. The CAS induces any page faults while ensuring that the same value is stored back to the memory location. Using a simple store might lead to incorrect values being stored back in mapping table slot under concurrent operations, and using a simple load might leave the page in a shared, copy-on-write state (for example, if the page was a fresh "zero page"). After executing this fallback path, the MW-CAS then retries its updates. We have yet to see an HTM transaction spin infinitely using this approach.

**Thread local read/write lock.** Of course, given Intel cannot guarantee hardware transactions commit, avoiding page faults even for short transactions does not guarantee progress. Two transactions with overlapping read/write sets can collide and continuously abort on data conflicts. Ultimately, a "stop the world" approach is needed to guarantee the MW-CAS can make progress by giving it exclusive access to update its target words non-transactionally. One way to achieve exclusivity while avoiding a global shared lock is to assign a thread-local read/write lock to each thread, as depicted on the right-hand side of Figure 11. The approach, commonly known as lockaside (the idea was also used in NUMA-aware locking [4]), maintains a read/write lock for each thread. Before starting an operation, a thread acquires exclusive access to its own lock. After an MW-CAS has retried a number of times with no progress, it attempts to gain exclusive access to the mapping table by acquiring all locks from other threads in the set in a deterministic order; this is the only time a lock is modified by another thread. Once the MW-CAS acquires all locks, it modifies its mapping table entries and then releases all locks. In the common case, this approach is very efficient since thread-local lock acquisition involves modifying a line already in CPU cache on the thread's local socket, thus avoiding ping-ponging across sockets (or cores). Scalability for lock acquisition (and release) may be an issue on large many-core machines such as Intel's Xeon Phi. For current processors with HTM (currently single-socket and soon to be dual-socket), lock count should be less of an issue, assuming the thread count is close to the number of cores.

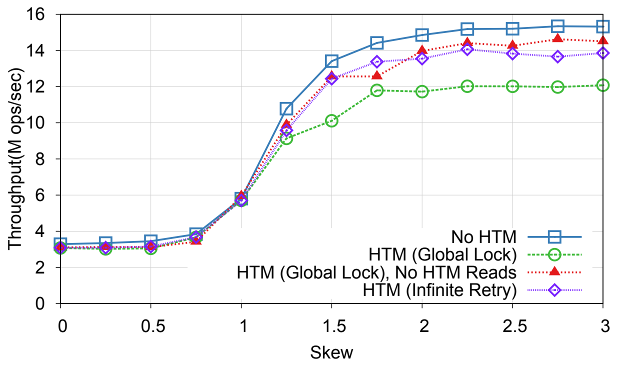


**Figure 12: Performance of two MW-CAS global lock and infinite retry approaches compared to the baseline lock-free performance.**

## 5.5 Evaluation

Figure 12 provides an experimental evaluation of the pure lock-free Bw-tree implementation (abbr. *No HTM*) along with the MW-CAS approaches discussed in Sections 5.3 and 5.4. The experimental workload contains 4 update threads and 4 read threads, where each thread selects the next record to update/read at random from a range of 10M keys using a Zipf distribution. We plot total throughput (y-axis) for various Zipf skew parameters (x-axis). We omit numbers for the approach that brackets entire tree traversals in a transaction, since its performance (due to aborts) does not come close to the other alternatives. Using the global elided lock to bracket all operations (even singleton reads, abbr. *Global Lock*) is simple to implement. However, performance degrades by up to 25% due the extra overhead of transaction setup and teardown (roughly 65 cycles on our experiment machine). Removing reads from the global elided lock (abbr. *No HTM Reads*) brings performance within 10% of *No HTM*. However, this is at the cost of no reduction in code complexity, as discussed in Section 5.3.2. The *Infinite Retry* approach exhibits similar performance to *No HTM Reads* while reducing code complexity, suggesting that removal of singleton updates from a transaction does not lead to better performance. This is because, unlike singleton reads, transaction setup/teardown time is not the main source of overhead; most of the overhead is due to the write itself.

## 6. DISCUSSION

This section extracts key lessons from our experiences evaluating HTM for main-memory indexing methods. First, we summarize the benefits and challenges of using current-generation Haswell HTM. Then, we give a "wish list" for features we hope to see in future CPUs that would improve the predictability and performance of high-performance main-memory indexes built with HTM.

## 6.1 HTM Benefits and Challenges

HTM allows easy parallel access to simple data structures. Section 3 showed that for a moderately-sized index small fixed-length key and payload sizes HTM allows for almost perfect thread scalability with little effort.

However, predicting HTM performance in indexes is hard and limits its usefulness. Many factors influence concurrency (and, in turn, performance), some of which vary at runtime. In the end, developers must take a conservative approach or risk performance collapse. For example, predicting abort rates combines the complexity of predicting page table walks, cache misses due to capacity and associativity (e.g., key and payload sizes in the index), application access patterns, and thread scheduling. Furthermore, the parameters for each of these variables change with each new CPU: TLB coverage, cache configuration, core count, and hardware thread configuration all vary from generation to generation.

With the current generation of HTM, its best to "keep it simple". The simplest and most predictable approach to leveraging HTM for main-memory indexing is as a small primitive where the number of cache lines accessed are constant per transaction rather than some function of data structure size or workload skew. This is exactly what we have done by using HTM as a multi-word CAS, as proposed in Section 5. We observed that almost every abort in these transactions is due to a data access conflict (rather than other spurious aborts) and the cost of retrying is low enough that it never performs worse than a spin-lock approach even with infinite retries.

## 6.2 HTM Wish List

Motivated by the findings in this paper, we now turn to a set of "wish list" items that we hope to see supported for future HTM release. While we justify our need for these items in the context of building high-performance main-memory indexes, we believe these features will benefit the general computing community as well.

**Selective choice of which accesses to add to a transaction**. Section 3.1 showed that key, payload, and index size all influence HTM abort rate, which affects indexing performance. Choice over which accesses are tracked by HTM could mollify this. Such a feature would allow a transaction to track just the accesses necessary to detect conflicts (e.g., payload data in the leaf page or a page "lock" or version to detect SMOs). This data is small compared to accesses done during binary search that are the source of capacity aborts and depend on factors like key and page size.

**Removal of words from a transaction.** The ability to remove words from a transaction would allow techniques like lock-coupling to use elided HTM locks efficiently. In most cases, coupled locks are held briefly during traversal and are not highly contended. Therefore the majority of overhead comes from the cache traffic involved in acquiring the lock. Using lock elisions would likely remove a majority of this overhead. The missing primitive to enable lock-coupling with HTM is an ability to remove a previous (source) lock in the traversal from the transaction read/write set. This feature would also allow the transactional read set for locks to remain constant size regardless of index size or depth.

AMD's unimplemented ASF instruction set describes such a "release," which removes a cacheline from a transaction's read set. A complication with release is that a cacheline may contain more than one word of a transaction's read set and would require great care or compiler support to use safely.

**Minimum capacity and liveness.** Section 5 explored multi-word CAS on the Bw-tree mapping table for atomic multi-page SMO updates. MW-CAS transactions are small and short-lived, and avoid many of the spurious abort conditions listed by Intel [18]. However, even by avoiding these conditions Intel does not guarantee a transaction will ever succeed, thus our more efficient design (the infinite-retry approach) had to resort to distributed thread-local locks to guarantee progress. Two features that would go a long way toward realizing a practical MW-CAS implementation on arbitrary memory locations would be (1) guarantee of a minimum transaction capacity; even a small capacity up to two to three words would be helpful for our purposes and (2) a liveness guarantee that at least one transaction would "win" during an MW-CAS operation (similar to single-word CAS) when concurrent transactions conflict.

## 7. RELATED WORK

Lomet proposed transactional memory [28], and Herlihy and Moss proposed the first practical implementation [17]. Until recently most research has focused on *software* transactional memory (STM) [35]. However, its high overheads have kept STM from gaining traction within database systems. This is especially true in main-memory systems; HyPeR recently realized only a 30% gain over single-threaded performance using STM to achieve parallelism [24]. Current HTM implementations [20, 21, 37] seem to be a viable, high performance method to achieve multi-threaded parallelism within database systems.

**HTM and Database Systems.** Hardware transactional memory has become a timely topic in the database community, especially in the area of main-memory systems. The HyPeR team explored how to exploit HTM to achieve concurrency within main-memory database systems [24]. The idea is to achieve thread parallelism by breaking a transaction into individual record accesses (read or update), where each access executes within a hardware transaction using lock elision. These accesses are "glued" together using timestamp ordering concurrency control to form a multi-step database transaction. This work also included an indexing microbenchmark on small (4-byte) fixed length keys that showed HTM scaling better than fine-grained locking, but its focus was achieving high concurrency during transaction execution. Our work focuses solely on main-memory indexing by evaluating HTM and lock-free designs.

Karnagel et al explored using HTM to improve concurrency in a B+-tree and in SAP Hana's delta storage index [21]. Similar to our study in Section 3, they found using HTM with a global lock leads to excellent performance for databases with small fixed-size keys but that care must be taken within the index implementation (in the case of the delta index) to avoid HTM aborts. This work mainly explored HTM as a drop-in solution to improve concurrency of existing index implementations. Our work explores this approach as well, but in addition show HTM performance is sensitive to properties of the indexed data (key, payload, and data set size). We also compare HTM-enabled index designs with state-of-the-art latch-free indexing approaches and explore how to use HTM to simplify latch-free programming using an efficient multi-word CAS.

**Main-Memory Optimized Indexing.** Our experiments use two implementations of a main-memory B+-tree. The Bw-tree, our latch-free implementation, is currently used within a number of Microsoft products [10, 12]. PLP [32] and Palm [34] are latch-free B+-trees. However, these designs use hard data partitioning to achieve latch freedom, where a single thread manages access to each partition. Since partitioned designs do not use any synchronization primitives (neither locks nor CAS), it is unclear how HTM might improve performance in these designs.

Other main-memory indexes achieve great performance. For example, ART [23] and Masstree [29] are based on tries (Masstree is a B-tree of tries). We deliberately do not to compare against these since our objective is (1) not to have a main-memory index performance bakeoff and (2) to get an apples-to-apples comparison of latch-free and HTM-based designs on the same data structure (a vanilla B+-tree). Further, ART and Masstree do not have latch-free or lock-based counterparts, respectively.

**Multi-Word Compare and Swap.** Section 5 evaluated a multi-word CAS (MW-CAS) implemented via HTM. Other works investigate implementing MW-CAS using software [2, 16, 19]. Most of these approaches, such as that proposed by Harris et al [16], use the hardware-provided single-word CAS as a building block to implement MW-CAS in software. We did not experiment with MW-CAS software implementations; we expect them to be less efficient than an HTM-based implementation. For instance, all software MW-CAS implementations we studied (supporting arbitrary words) contained an internal memory allocation, e.g., to manage a state descriptor for the multiple words [16].

For performance reasons, Section 5 also discusses methods for safe interaction between non-transactional and transactional code when implementing MW-CAS using a best-effort HTM. TM systems that enable such safe interaction (with strong isolation guarantees) are referred to as strongly atomic [3]. Related work on strongly atomic TM is focused either on STM [1, 8] or proposes new hardware features for fine-grain memory protection [31]. We are not aware of work that investigates mixing non-transactional code with an existing best-effort HTM implementation, such as Intel's TSX. Another line of work on overcoming problems of best-effort HTM includes safe mixing of software and hardware transactions, such as HyTM [9], PhTM [25] and recently Invyswell [5]. The STM implementations in these systems are costly, and cannot be used in our case since they are not strongly atomic.

## 8. CONCLUSION

This paper studied the interplay of hardware transaction memory (HTM) and lock-free data structures. We began by exploring the effect of using HTM on a "traditionally" designed high-performance main-memory B-tree and found that HTM, in its current form, is not yet suitable as a general-purpose solution for concurrency in high-performance data processing engines. HTM provides great scalability for modestly sized data sets with small fixed key and payload sizes. However, transaction abort rates rise for more realistic workloads with larger data sets containing larger variable-length keys and payloads, leading to poor performance. In addition, we find that HTM is not yet suitable for implementing fine-grained locking techniques such as lock-coupling that would greatly reduce HTM aborts due to capacity constraints. We next explored fundamental differences between HTM-based and lock-free B-tree designs. We find that lock-free designs are advantageous and still useful, particularly in high contention scenarios since readers never block, while an HTM-based approach will abort readers that touch "hot" data items. Finally, we explored the use of HTM within the lock-free Bw-tree in order to simplify its design and implementation, focusing on concurrency of its indirection mapping table that maps logical page ids to pointers. We explored several designs of a multi-word compare and swap (MW-CAS) that use HTM to arbitrate conflict on multiple arbitrary cache lines, and find that the MW-CAS is a great application of HTM: it avoids spurious aborts since transactions are small and short, provides good performance, and indeed simplifies lock-free data structure design.

## 9. REFERENCES

[1] M. Abadi, T. Harris, and M. Mehrara. Transactional Memory with Strong Atomicity Using Off-the-shelf Memory Protection Hardware. In *PPoPP*, pages 185–196, 2009.

[2] J. H. Anderson and M. Moir. Universal Constructions for Multi-Object Operations. In *PODC*, pages 184–193, 1995.

[3] C. Blundell, E. C. Lewis, and M. M. Martin. Subtleties of Transactional Memory Atomicity Semantics. *IEEE Comput. Archit. Lett.*, 5(2):–, Feb 2006.

[4] I. Calciu et al. NUMA-Aware Reader-Writer Locks. In *PPoPP*, pages 157–166, 2013.

[5] I. Calciu et al. Invyswell: A Hybrid Transactional Memory for Haswell's Restricted Transactional Memory. In *PACT*, pages 187–200, 2014.

[6] J. Chung et al. ASF: AMD64 extension for lock-free data structures and transactional memory. In *MICRO*, pages 39–50, 2010.

[7] cpp-btree. **`https://code.google.com/p/cpp-btree/`**.

[8] T. Crain, E. Kanellou, and M. Raynal. STM Systems: Enforcing Strong Isolation Between Transactions and Non-transactional Code. In *ICA3PP*, pages 317–331, 2012.

[9] P. Damron et al. Hybrid Transactional Memory. In *ASPLOS*, pages 336–346, 2006.

[10] C. Diaconu et al. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *SIGMOD*, pages 1243–1254, 2013.

[11] D. Dice et al. Applications of the Adaptive Transactional Memory Test Platform. In *TRANSACT Workshop*, 2008.

[12] Azure DocumentDB. **`https://www.documentdb.com`**.

[13] G. Graefe. A Survey of B-tree Locking Techniques. *ACM Trans. Database Syst.*, 35(3):16:1–16:26, 2010.

[14] J. Gray et al. Quickly Generating Billion-record Synthetic Databases. In *SIGMOD*, pages 243–252, 1994.

[15] S. Harizopoulos et al. OLTP Through the Looking Glass, and What We Found There. In *SIGMOD*, pages 981–992, 2008.

[16] T. L. Harris et al. A Practical Multi-Word Compare-and-Swap Operation. In *Distributed Computing*, pages 265–279. 2002.

[17] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *ISCA*, pages 289–300, 1993.

[18] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, September 2014.

[19] A. Israeli and L. Rappoport. Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives. In *PODC*, pages 151–160, 1994.

[20] C. Jacobi et al. Transactional Memory Architecture and Implementation for IBM System z. In *MICRO*, pages 25–36, 2012.

[21] T. Karnagel et al. Improving In-Memory Database Index Performance with Intel Transactional Synchronization Extensions. In *HPCA*, pages 476–487, 2014.

[22] P. L. Lehman and S. B. Yao. Efficient Locking for Concurrent Operations on B-Trees. *TODS*, 6(4):650–670, 1981.

[23] V. Leis et al. The Adaptive Radix Tree: ARTful Indexing for Main-memory Databases. In *ICDE*, pages 38–49, 2013.

[24] V. Leis et al. Exploiting Hardware Transactional Memory in Main-Memory Databases. In *ICDE*, pages 580–591, 2014.

[25] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *Transact*, 2007.

[26] J. Levandoski et al. High Performance Transactions in Deuteronomy. In *CIDR*, 2015.

[27] J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. In *ICDE*, pages 302–313, 2013.

[28] D. B. Lomet. Process Structuring, Synchronization, and Recovery Using Atomic Actions. In *Proceedings of the ACM Conference on Language Design for Reliable Software*, pages 128–137, 1977.

[29] Y. Mao, E. Kohler, and R. T. Morris. Cache Craftiness for Fast Multicore Key-Value Storage. In *EuroSys*, pages 183–196, 2012.

[30] The Story Behind MemSQLs Skiplist Indexes. **`http://blog.memsql.com/the-story-behind-memsqls-skiplist-indexes/`**.

[31] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *ISCA*, pages 69–80, 2007.

[32] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki. PLP: Page Latch-free Shared-everything OLTP. *PVLDB*, 4(10):610–621, 2011.

[33] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *MICRO*, pages 294–305, 2001.

[34] J. Sewall et al. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors. *PVLDB*, 4(11):795–806, 2011.

[35] N. Shavit and D. Toutitou. Software Transactional Memory. In *PODC*, pages 204–213, 1995.

[36] V. Srinivasan and M. J. Carey. Performance of B+ Tree Concurrency Algorithms. *VLDB Journal*, 2(4):361–406, 1993.

[37] R. M. Yoo et al. Performance Evaluation of Intel Transactional Synchronization Extensions for High-Performance Computing. In *SC*, pages 19–29, 2013.