Appears in the 20th International Symposium On High-Performance Computer Architecture, Feb. 15 - Feb. 19, 2014.

Improving In-Memory Database Index Performance with Intel® Transactional Synchronization Extensions

Tomas Karnagel[†⋆], Roman Dementiev[†], Ravi Rajwar[†], Konrad Lai[†],
Thomas Legler[‡], Benjamin Schlegel[⋆], Wolfgang Lehner[⋆]

[†]Intel Corporation, Munich, Germany and Hillsboro, USA
[‡]SAP AG, Database Development, Walldorf, Germany
[⋆]TU Dresden, Database Technology Group, Dresden, Germany
[†]first.last@intel.com, [‡]first.last@sap.com, [⋆]first.last@tu-dresden.de

Abstract

The increasing number of cores every generation poses challenges for high-performance in-memory database systems. While these systems use sophisticated high-level algorithms to partition a query or run multiple queries in parallel, they also utilize low-level synchronization mechanisms to synchronize access to internal database data structures. Developers often spend significant development and verification effort to improve concurrency in the presence of such synchronization.

The Intel® Transactional Synchronization Extensions (Intel® TSX) in the 4th Generation Core™ Processors enable hardware to dynamically determine whether threads actually need to synchronize even in the presence of conservatively used synchronization. This paper evaluates the effectiveness of such hardware support in a commercial database. We focus on two index implementations: a B+Tree Index and the Delta Storage Index used in the SAP HANA® database system. We demonstrate that such support can improve performance of database data structures such as index trees and presents a compelling opportunity for the development of simpler, scalable, and easy-to-verify algorithms.

1. Introduction

The increasing number of cores every generation poses new challenges for the implementation of modern high-performance in-memory database systems. While these database systems use sophisticated algorithms to partition a query or run multiple queries across multiple cores, they also utilize low-level synchronization mechanisms such as latches and locks[1]to synchronize access from concurrently executing threads to internal in-memory data structures, such as indexes. Database developers often spend significant development effort and resources to improve concurrency in the presence of such synchronization. Numerous implementations have been proposed over the years with the goal to improve scalability and concurrency, and systems today employ a wide range of sophisticated implementations. Unfortunately, such implementations end up being hard to verify and developers must deal with numerous corner cases and deadlock scenarios. As Graefe [10] has noted, "*Perhaps the most urgently needed future direction is simplification. Functionality and code for concurrency control and recovery are too complex to design, implement, test, debug, tune, explain, and maintain.*" The development of simple and scalable easy-to-verify implementations remains challenging.

With the increasing dependence on parallelism to improve performance, researchers have investigated methods to simplify the development of highly-concurrent multi-threaded algorithms. Transactional Memory, both hardware [13] and software [23], aims to simplify lock-free implementations. However, integrating it into complex software systems remains a significant challenge [12]. Lock elision [19, 20] proposes hardware mechanisms to improve the performance of lock-based algorithms by transactionally executing them in a lock-free manner when possible. In such an execution, the lock is only read; it is not acquired nor written to, thus exposing concurrency. The hardware buffers transactional updates and checks for conflicts with other threads. If the execution is successful, then the hardware makes all memory operations performed within the region to appear to occur instantaneously. The hardware can thus dynamically determine whether threads need to synchronize and threads perform serialization only if required for correct execution.

However, it has so far been an open question as to whether modern databases can take advantage of such hardware support in the development of simple yet scalable concurrent algorithms to manage internal database state [17, 14, 10].

This paper addresses that question and evaluates the effectiveness of hardware supported lock elision to enable simple yet scalable database index implementations. For this paper, we use the Intel® Transactional Synchronization Extensions (Intel® TSX) in the Intel 4th Generation Core™ Processors but our findings are applicable to other implementations.

[1]We use the terms latches and locks interchangeably to represent low-level synchronization mechanisms. Locks here do not refer to database locks used to provide isolation between high-level database transactions.

Paper Contributions. This paper presents the first experimental analysis of the effectiveness of hardware supported lock elision to enable the development of simple yet scalable index implementations in the context of a commercial in-memory database system.

- Highlights the scalability issues that arise from contended atomic read-modify-write operations in modern high-performance systems.
- Applies Intel TSX-based lock elision to two index implementations, the common B+Tree index and a more complex Delta Storage Index used in the SAP HANA® in-memory database system and demonstrates how such hardware support can enable simple yet scalable algorithms.
- Shows how applying well-known transformations can help overcome hardware limitations and increase the effectiveness of such hardware support.

Section 2 analyzes the concurrency behavior of two index implementations: A common B+Tree index and a more complex Delta Storage Index used in SAP HANA. The two represent different examples of access patterns and foot prints for index traversals. We find that even with a reader-writer lock and read-only accesses, the communication overheads associated with atomic read-modify-write operations on the reader lock can limit scalability on modern systems. Further, even a modest fraction of insert operations (5%) can degrade performance. These results motivate the use of lock elision to eliminate the cache-to-cache transfers of the reader-writer lock, even with only readers.

Section 3 describes the Intel Transactional Synchronization Extensions, its implementation and programming interface, and provides key guidelines to effectively use Intel TSX.

Section 4 applies Intel TSX-based lock elision to the two indexes. For this study, we use a mutual-exclusion spin lock for the Intel TSX version and compare against a more complex reader-writer lock. We find that a simple spin lock when elided provides better scalability across a range of configurations. The initial performance gains are significant for the B+Tree index but more modest for the Delta Storage Index. We investigate this further and identify cases where lock elision was unsuccessful. Section 5 presents simple transformations to make the Delta Storage Index more elision friendly. With these transformations, a spin lock with lock elision significantly outperforms a reader-writer lock.

While we have focused on complex database data structures such as index trees, the key concepts outlined in the paper are applicable to other similar data structures that are accessed concurrently by multiple threads but where the probability of an actual data conflict is relatively low.

2. SAP HANA Delta Storage Index

The SAP HANA Database System is a read optimized column store system. Each column consists of a Main Storage and a Delta Storage as shown in Figure 1. The Main Storage is

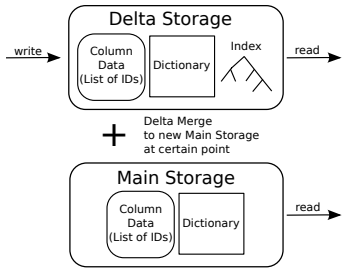


Figure 1: Delta and Main Storage in SAP HANA®

a read-only data structure and maintains highly compressed permanent data optimized for querying. The Delta Storage supports insert and read operations and tracks deletions using visibility flags for each row. The Delta Storage tracks all changes to a column and is optimized for writes. If the Delta Storage exceeds a certain size, a merge operation joins the Delta Storage with the actual column.

The Main Storage and the Delta Storage both maintain dictionaries for compression. The database uses the dictionary to replace the stored data with an identifier (ID) that points to the dictionary entries. The Main Storage dictionary is sorted to provide optimized search performance and high compression. The Delta Storage on the other hand is optimized for efficient insert operations and thus maintains an unsorted dictionary where new values can only be appended. It therefore maintains a secondary index tree, the Delta Storage Index, for fast searches on this unsorted dictionary. Figure 2 shows an example organization. The column data on the left consists of IDs that point to entries in the dictionary containing unsorted distinct values and IDs. The Delta Storage Index on the right contains IDs. The order of the IDs in the index depends on the dictionary entries represented. Storing only the IDs of entries ensures a fixed node size and constant cost for copying data.

A query always starts at the root node and looks up the dictionary for every relevant ID in a node. For example, a query for "Oakland" looks up ID 4 and 7 of the root node. ID 4 represents "Fresno" and ID 7 represents "Palo Alto". In alphabetical order "Oakland" lies between the two. The search continues to the second child node and looks up ID 4 and 8. "Oakland" lies in the second position (ID 8). This is done for every search or insert operation. If an entry being inserted already exists in the index, the ID is added to the column data, otherwise the entry is added to the dictionary and an ID is assigned. This ID is also added to the index and column data list. While the index supports efficient search and insert, it can also be used to sort the dictionary if needed. Reading all the IDs of all the leaf nodes in order (e.g., 0;5;4;8;7;6;1;3;2) returns the IDs in alphabetical order of their entries.

2.1. Experimental Analysis

We first study the concurrency performance of the B+Tree and Delta Storage Index.

2.1.1. Index Tree Synchronization Extensive work exists in concurrency control for index trees [10]. These proposals vary

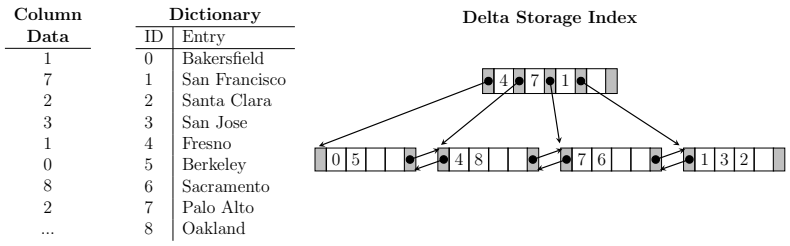


Figure 2: Example of a delta storage, including column data, dictionary, and index.

widely in their concepts and implementation, often employing various mechanisms to protect index data structures or taking advantage of higher-level information and structure to ensure content consistency of the indexes. Such scalable concurrency control implementations are often complex and require significant effort to verify, especially when integrating into existing database systems. We discuss this further in Section 6.

In this paper, we are concerned with the challenge to develop scalable yet easy-to-verify implementations. Keeping that goal in mind, we adopt a simpler baseline concurrency control implementation for index trees and investigate the impact of using hardware support to improve its scalability.

The Delta Storage Index implementation in SAP HANA uses a reader-writer lock to protect the index. A reader-writer lock allows multiple readers to concurrently access the tree but serializes access when a writer accesses the tree. The choice of the synchronization mechanism (concurrent readers, serial writers) in the SAP HANA index tree implementation is driven by multiple factors, primary being the need for an easy-to-verify implementation for integration into a complex commercial database system with established interfaces.

The baseline synchronization implementation we use is a reader-writer lock (referred to here as **RW Lock**). The RW lock is from the Intel® Thread Building Blocks (Intel® TBB) suite[2]. The RW lock is a high-performance, spinning reader-writer lock with backoff and writer-preference.

In addition, we also study the scalability of a conventional mutual-exclusion spin lock (referred to here as **Spin Lock**) that provides exclusive access to the tree. The spin-lock we use is similar to the one from the Intel® TBB suite.

2.1.2. Experimental Setup We use a 2-socket (Intel® Xeon® CPU E5-2680) server system running the Linux operating system. Each test initializes the database index with 20 million even numbers starting from 0. This is a typical size for the Delta Storage Index in real-world SAP HANA configurations. The search and insert queries have a uniform random key distribution between the minimum and maximum tree limits. Search keys have even values to make every search lookup successful. We insert only odd keys, such that the queries will most likely result in a tree modification. With a large key range of the index, a small probability exists that random inserts attempt to add the same key several times. However, duplicate inserts will not result in a write operation.

2.1.3. Index Tree Scaling We next study scaling on single socket and dual socket configurations.

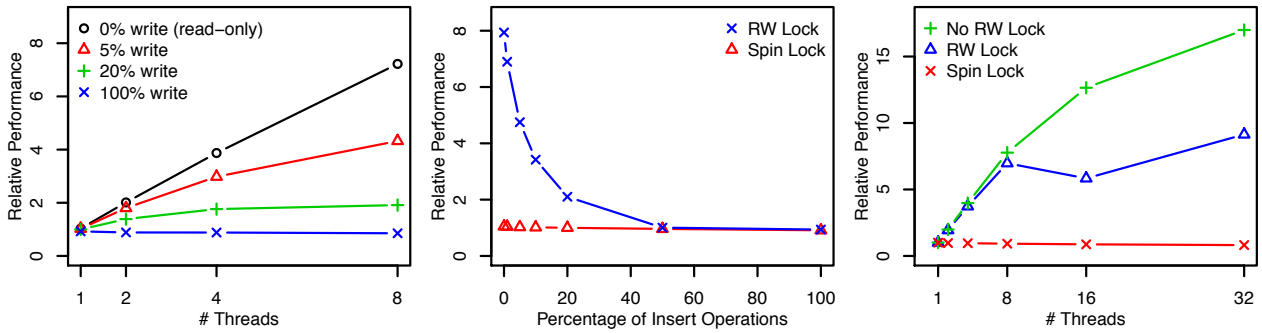
Single Socket. Figure 3a plots the relative performance of a Delta Storage Index using the baseline RW lock for 0%, 5%, 20%, and 100% writes. Figure 4a plots the same for a B+Tree. For the 0% (read-only) case, RW lock scales as expected because it allows concurrent readers without serialization. However, as the insert percentage increases, performance degrades with additional threads. This is because, unlike for readers, the writer lock provides only exclusive access to the tree. As expected, Spin Lock does not scale since it serializes all accesses. Figure 3b plots the Delta Storage Index relative performance for different insert percentages on a single socket 8 thread configuration. Figure 4b plots the same for a B+Tree. As we can see, with increasing insert percentage, the performance of the RW lock drops to that of a spin lock. This is also expected as increasing writers serialize and block readers and synchronization communication overhead starts to play a significant role.

Dual Socket. Figure 3c shows the performance of read-only accesses with increasing threads on a multi-socket server configuration. Figure 4c plots the same for a B+Tree. Interestingly, even though the RW lock shows good scaling for up to 8 threads in the Delta Storage Index, scaling drops as we increase the number of threads to 16 due to the impact of cross-socket communication on the shared RW lock. The 32 thread configuration supports simultaneous multithreading through Intel® Hyper-Threading (HT) Technology where each core can run two simultaneous threads. Scaling for the RW lock from 16 to 32 slightly improves due to memory-level parallelism introduced by HT.

2.1.4. Understanding Scalability The 100% read experiments provide a good baseline to understand the impact of communication in modern high-performance systems. For these experiments, the only inter-thread communication is the atomic operation on a shared location for the reader lock. Since multiple readers can access the tree there are no retries to acquire the reader lock and the operation itself is non-blocking.

For the 100% read experiment, we observe good scaling for the Delta Storage Index (Figure3a) but lower scaling for B+Tree (Figure 4a) for the 8T single socket and worse scaling for the cross-socket configurations.

[2]Source at http://www.threadingbuildingblocks.org/. Intel TBB libraries are extensively used in numerous high-performance multi-threaded software.



(a) Scaling for different insert ratios with a RW lock on a single socket.

(b) Performance for varying insert ratios on a 8-thread single socket.

(c) Scaling for 100% read access for different lock implementations on multiple sockets.

Figure 3: Delta Storage Index Concurrency Performance

To better understand the drop in scaling even for the read-only configuration, we experimented with an index implementation that does not implement any locks to protect the index. This is shown with the No RW Lock label. Since the experiment performs 100% reads, correctness is always maintained. The difference between the No RW Lock line and the RW lock lines shows the performance overhead of performing lock operations across multiple threads. This is because even with an RW lock, acquiring the reader lock requires an atomic read-modify-write operation on the variable (implemented through a LOCK prefixed instruction in the Intel® 64 Architecture [2]). This requires hardware to request exclusive ownership of the cache line containing the variable prior to the update. When multiple threads acquire a reader lock and update the same variable, it causes each processor to request exclusive ownership for the line, resulting in the cache line with the variable moving around the system from one processor to another.

This overhead manifests itself in the performance gap shown in Figures 3c and 4c. While the overhead is not significant in the single socket case, the latencies due to crossing multiple sockets result in a larger performance gap in the multi-socket case. The B+Tree does not scale as well as the Delta Storage Index in general because unlike the Delta Storage Index, the B+Tree by itself is a simple and small critical section putting more stress on the lock variable cache-to-cache transfer overhead.

Further, modern high-performance processors execute such critical sections with low latency which exposes the cache-to-cache transfer latencies. These processors also execute the single thread workload efficiently with low latency in the absence of cache-to-cache communication. As a result, observed scaling behavior is a strong function of the performance of the underlying processor, and the computational characteristics of the synchronizing threads.

This data highlights an often overlooked aspect of atomic read-modify-write operations. Even though a reader lock enables concurrent accesses, with increasing cores the overhead of the atomic read-modify-write operation itself can contribute to scalability issues due to the inherent communication and cache-to-cache transfers required. Proposals to improve performance must minimize such cross-thread communication.

2.2. Concurrency Analysis

Insert operations require locks because they change the tree's internal node structures. However, the probability of an insert conflicting with other operations should be low for large trees. Eliding these locks can be quite effective to enable read and insert operations to occur concurrently. To motivate lock elision for index trees, we first calculate the conflict probability.

We define two probabilities, P_{ACCESS} and P_S for a tree where leaf nodes are on $L = 0$ and root node is on $L = (height - 1)$.

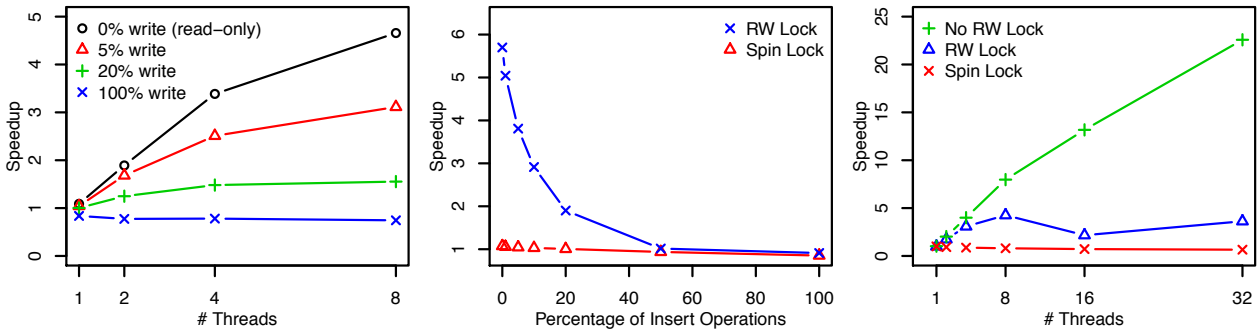
$$P_{ACCESS} = \frac{1}{\text{No. Nodes on } L} \qquad \text{and} \qquad Ps = \frac{1}{0.69b}^{L} \text{ [8]}$$

For random accesses, P_{ACCESS} is the probability that two operations access the same node at tree level L and P_S is the probability that an insert operation propagates to level L [8]. For example, if $L = 1$, then P_S is the probability that an insert splits a leaf node and inserts an ID to the tree level above the leaf nodes. Node splits cause inserts to the parent node, which themselves can cause node splits and propagate further up the tree. If no node splits occur, then a conflict can only occur at the leaf node level. If a node split occurs, then a conflict can occur at levels above the leaf nodes.

Consider an index tree with a maximal branching factor $b = 31$ and a height of 5, and two concurrent operations to the tree where one is an insert and the other is a read. These parameters represent a common Delta Storage Index configuration. Table 1 lists the computed probabilities and the average node count for every level L. The highest P_{ACCESS} is 1 for the root node ($L = 4$) and the lowest for the leaf node level. In contrast, P_S is 1 for leaf nodes and lower for the levels above.

The conflict probability, P_C, is computed for the case where the insert operation propagates to a level L and the other operation accesses this node at level L while traversing the tree. Overall, the probability of a conflict for a tree of this size is fairly low (maximum value of $6.39 \cdot 10^{-6}$).

We next discuss applying Intel TSX to index tree locks.



(a) Scaling for different insert ratios with a RW lock on a single socket.

(b) Performance for varying insert ratios on a 8-thread single socket.

(c) Scaling for 100% read access for different lock implementations on multiple sockets.

Figure 4: B+Tree Concurrency Performance

L	Avg No. Nodes	Prob. P_{ACCESS}	Prob. P_S	Prob. P_C
4	1	1	$4.78 \cdot 10^{-6}$	$4.78 \cdot 10^{-6}$
3	16	$6.25 \cdot 10^{-2}$	$1.02 \cdot 10^{-4}$	$6.39 \cdot 10^{-6}$
2	496	$2.72 \cdot 10^{-3}$	$2.19 \cdot 10^{-3}$	$5.94 \cdot 10^{-6}$
1	15100	$1.18 \cdot 10^{-4}$	$4.68 \cdot 10^{-2}$	$5.52 \cdot 10^{-6}$
0	465000	$5.14 \cdot 10^{-6}$	1	$5.14 \cdot 10^{-6}$

Table 1: P_{ACCESS}, P_S, and P_C for an average tree of height 5.

3. Intel Transactional Synchronization Extensions

Intel Transactional Synchronization Extensions (Intel TSX) consist of hardware support to improve the performance of the lock-based programming model. With these extensions, the processor can dynamically determine whether threads need to serialize the execution of lock-protected code regions, and to perform serialization only when required, through a technique known as **lock elision**.

With lock elision, the processor executes the programmer-specified lock-protected code regions (also referred to as **transactional regions**) transactionally. In such a transactional execution, the lock is only read; it is neither acquired nor written to, thus exposing concurrency. With transactional execution, the hardware buffers any transactional updates and checks for conflicts with other threads.

On a successful transactional execution, the hardware performs an **atomic commit** to ensure all memory operations performed within the transactional region appear to have occurred instantaneously when viewed from other processors. Any updates performed within the transactional region are made visible to other processors only on an atomic commit. Since a successful transactional execution ensures an atomic commit, the processor can execute the programmer-specified code section optimistically without synchronizing through the lock. If synchronization was unnecessary, the execution can commit without any cross-thread serialization.

If the transactional execution is unsuccessful, the processor performs a **transactional abort** since it cannot commit the updates atomically. On a transactional abort, the processor discards all updates performed in the region, restores architectural state to appear as if the optimistic execution never occurred, and resumes execution non-transactionally. Depending on the policy in place, lock elision may be retried or the lock may be explicitly acquired.

Transactional aborts can happen for numerous reasons. A common cause is conflicting accesses between the transactionally executing processor and another processor. Memory addresses read within a transactional region form the read-set of the transactional region and addresses written within the transactional region form the write-set of the transactional region. A conflicting data access occurs if another processor either reads a location that is part of the transactional region's write-set or writes a location that is part of either the read- or write-set of the transactional region. We refer to this as a **data conflict**. Such conflicting accesses may prevent a successful transactional execution. Since hardware detects data conflicts at the granularity of a cache line, unrelated data locations placed in the same cache line will be detected as conflicts. Transactional aborts may also occur due to limited capacity. For example, the amount of data accessed in the region may exceed an implementation-specific capacity. Additionally, some instructions and system events may cause transactional aborts.

The Intel® 64 Architecture Software Developer Manual has a detailed specification for Intel TSX and the Intel TSX web resources site[3] presents additional information.

3.1. Programming Interface For Lock Elision

Intel TSX provides programmers two interfaces to implement lock elision. Hardware Lock Elision (HLE) is a legacy compatible extension that uses the XACQUIRE and XRELEASE prefixes. Restricted Transactional Memory (RTM) is a new instruction set extension that uses the XBEGIN and XEND instructions. Programmers who also want to run Intel TSX-enabled software on legacy hardware would use the HLE

[3] http://www.intel.com/software/tsx

interface to implement lock elision. On the other hand, programmers who do not have legacy hardware requirements and who deal with more complex locking primitives would use the RTM interface to implement lock elision.

To use the HLE interface, programmers add prefixes to existing synchronization routines implementing the lock acquire and release functions. To use the RTM interface, programmers augment the routines to support an additional code path implementing lock elision. In this path, instead of acquiring the lock, the routine uses the XBEGIN instruction and provides a **fallback handler** (the operand of the XBEGIN instruction) to execute if a transactional abort occurs. In addition, the lock being elided is tested inside the transactional region to ensure it is free. This action also adds the lock address to the read set, thus allowing for any conflicts on the lock variable to be detected. If the lock was not free, then the RTM-based execution must abort since some other thread has explicitly acquired the lock. With both the HLE and RTM interfaces, the changes to enable lock elision are localized to synchronization routines; the application itself does not need to be changed.

3.2. Hardware Implementation

The first implementation of Intel TSX on the 4th Generation Core™ Processor uses the first level 32-KB data cache (L1) to track the transactional region's read- and write-sets[18]. All address tracking and data conflict detection occurs at a cache-line granularity, using physical addresses and the cache coherence protocol. Evicting a transactionally written line from the L1 cache causes an abort. Evicting a line that is only read transactionally does not cause an abort; the line is tracked in a secondary structure for conflict detection.

3.3. Handling Transactional Aborts

A successful transactional execution elides the lock and exposes concurrency. However, when a transactional abort occurs, an implementation policy determines whether the lock elision is retried or the lock is acquired. Prior work has suggested the benefit of retrying lock elision when aborts are frequent, since many aborts can be transient [19, 20, 9]. This retry may be implemented in hardware or in software.

In the first Intel TSX implementation, following a transactional abort, an HLE-based execution restarts from the lock instruction that initiated transactional execution and acquires the lock. This minimizes wasted work and is effective when aborts are infrequent. On the other hand, an RTM-based execution restarts at the fallback address provided with the XBEGIN instruction. This flexibility is possible because unlike the legacy-compatible HLE interface, the RTM interface uses new instructions. These new instructions provide an interface for programmers to implement a flexible retry policy for lock elision in the fallback handler.

Retrying lock elision instead of immediately acquiring the lock can be beneficial, since if the lock is acquired, then all other threads eliding that lock will also abort. As a result, the executions will serialize on the lock, thus idling all contending threads for the duration of that critical section.

3.3.1. Analyzing Transactional Aborts We use an extensive performance monitoring infrastructure to understand and analyze transactional aborts. Detailed information can be found in the Intel 64 Architecture Software Developer Manual and the Chapter 12 of the Intel 64 Architecture Optimization Reference Manual [1]. For RTM-based implementations, the fallback handler can also track retries. Computing effective abort rates in the presence of retries requires care. For example, the handler would acquire the lock only after a threshold of retries. However, if the region commits successfully following retries and before reaching the threshold, then that would still be beneficial since the lock wasn't acquired.

3.3.2. Fallback Handler Retry Policy How the fallback handler implements a retry is critical. There is no benefit to attempt lock elision if the lock is already held by another thread. It can actually be counter-productive. Consider the case where a transactional execution aborts sufficient number of times such that the retry threshold is reached and the thread acquires the lock. This causes other threads to also abort. If the other threads immediately retry lock elision, they will find the lock acquired and will abort. Subsequent continuous attempts to immediately try lock elision will result in the threads reaching their retry threshold without having had an opportunity to transactionally execute with lock elision; they would all have found the lock to be already held. As a result, all threads will transition to an execution where they explicitly acquire the lock sequentially and no longer try lock elision, a phenomenon called the *lemming effect*. This may result in the threads entering long periods of execution without attempting lock elision. A simple but effective way to avoid this is to retry lock elision only when the lock is free. Such a test prior to acquiring the lock is already common practice in conventional locking. This requires the fallback handler to simply test the lock and wait until it is free before re-attempting lock elision. With HLE, the insertion of a pause instruction in the path of the spin loop, a common practice, avoids such pathologies.

3.3.3. Dealing with High Data Conflict Rates Data structures that do not encounter significant data conflicts are excellent candidates, whereas data structures that repeatedly conflict do not see concurrency benefit. Performance under data conflicts has been extensively studied [19, 20, 7] and how hardware deals with data conflicts is implementation specific. Frequent data conflicts also often imply frequent lock contention and serialization, and thus baseline performance would also degrade. However, for hardware implementations with simple hardware conflict resolution mechanisms, there are cases where, under heavy data conflicts, retrying multiple times before acquiring the lock can actually degrade performance with respect to a baseline where the lock is immediately acquired. As we see later in Section 5, this may require changes to the data structure.

Scenarios exist where conflicts may be a function of the

data itself. Section 2.2 computed the conflict probability for random data accesses. Insertion of sorted data will result in frequent data conflicts because the execution will insert into the same or neighboring leaf node, and attempting lock elision would not help. To deal with such situations, we introduce adaptivity in the elision algorithm. A heuristic in the fallback path can address this issue. The fallback path maintains statistics about transactional aborts. If aborts reach a certain threshold, the fallback path disables subsequent lock elision for a period of time; the synchronization routine simply avoids the lock elision path and follows the conventional lock acquire path. The condition is periodically re-evaluated to adapt to the workload and lock elision is re-enabled in the synchronization library. Adaptivity did not trigger in our experiments.

4. Eliding Index Locks

In this section, we use Intel TSX to elide the B+Tree and Delta Storage Index locks. The hardware buffers memory updates and tracks memory accesses during the tree operations (reads and inserts) performed under lock elision.

4.1. Experimental Setup

We use an Intel® 4th generation Core™ processor (Intel® Core™ i7-4770 (3.4 GHz). The processor supports simultaneous multithreading through Intel® Hyper-Threading (HT) Technology. Each processor has 4 cores, with each core capable of running two simultaneous threads for a total of 8 simultaneous threads per processor. Each core has a 32-KB 8-way set-associative L1 data cache. In the 4-thread configuration, each thread is assigned to a separate core (HT disabled). In the 8-thread configuration, two threads are assigned to each core (HT enabled). In the latter configuration, the two threads assigned to the same core share the same L1 data cache.

We use the same database setup as described in Section 2.1.3. All threads periodically access the tree for a large number of iterations. Such concurrent access represents a common index usage in real-world configurations.

As in Section 2.1.3, we consider the two lock implementations (discussed in Section 2.1.1) for the B+Tree and the Delta Storage Index. We also compare against a version without any concurrency control.

- RW Lock: This is the spinning reader-writer lock where writers are preferred over readers.
- Spin Lock: This is the conventional spin lock that provides exclusive access to the tree.
- No Concurrency Control: This version implements no concurrency control. This is an unsafe execution and represents an upper bound for the index tree. The reported data points correspond to a correct execution.

We experiment with the two Intel TSX software interfaces to implement lock elision.

- Spin Lock Elision w/ TSX-R: Uses the RTM interface to implement lock elision for the conventional spin lock.
- Spin Lock Elision w/ TSX-H: Uses the HLE interface to implement lock elision for the conventional spin lock.

For the experiments labeled TSX-R, the fallback handler retries lock elision a certain number of times prior to explicitly acquiring the spin lock. For the experiments labeled TSX-H, the hardware immediately attempts to acquire the spin lock on a transactional abort. A limitation of using a conventional spin lock for lock elision is that on a transactional abort, the fallback path uses the spin lock. Unlike the RW lock, the spin lock does not provide reader concurrency. Thus, when the lock is frequently acquired due to aborts in read-heavy configurations, performance of a spin lock will be worse than a RW lock. This effect is more pronounced with TSX-H since on an abort, the hardware implementation immediately tries to acquire the lock without a retry. More sophisticated implementations for lock elision are possible [1].

We use both the 4-thread and 8-thread configurations to study the effect of increased pressure on transactional capacity due to the shared L1 data cache in HT enabled configurations.

4.2. Conventional B+Tree Results

Figure 5 shows the relative performance of the B+Tree index with various lock implementations (See Section 4.1) for different percentages of insert operations. The performance is normalized to the runtime of the spin lock with 0% insert operations. Figure 5a shows a 4-thread configuration (HT disabled) and Figure 5b shows the 8-thread configuration (HT enabled).

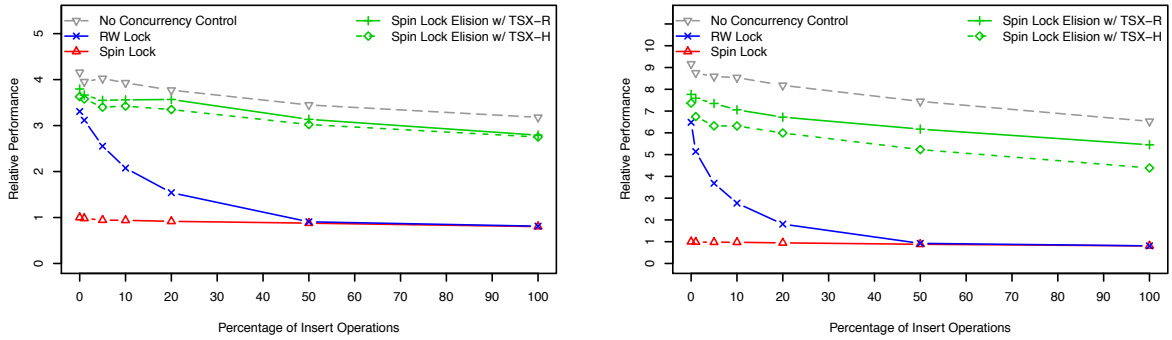
As expected, the performance of the spin lock remains flat because of serialization whereas the performance of the RW lock is significantly better than that of a spin lock for low percentage of inserts. However, RW lock performance degrades to that of the spin lock with increasing insert percentages.

Interestingly, the two lines labeled TSX-H and TSX-R show good performance across the entire range of insert percentages. This demonstrates the effectiveness of lock elision for a B+Tree where a simple spin lock with elision can outperform a more complex reader-writer lock. A Spin Lock with elision provides reader-reader concurrency similar to a RW lock, but it additionally also provides a high probability of reader-writer and writer-writer concurrency.

The performance drop between 0% and 100% insert is expected because inserts involve writing data, moving keys, and splitting nodes, and these operations have larger footprints than read-only operations. With few aborts with 0% inserts, the simple spin lock with elision is faster than the RW lock.

Further, the performance profile is quite similar to the upper-bound version without any concurrency control, and the performance gap is quite narrow across the entire range.

The performance difference between TSX-H and TSX-R is noticeable for the 8-thread configuration. Again, this is because on a transactional abort with TSX-H, the spin lock is immediately acquired. However, with TSX-R, the fallback



(a) 4 Threads (HT disabled)

(b) 8 Threads (HT enabled)

Figure 5: B+Tree Performance with Lock Elision

handler retries lock elision and avoids immediately transitioning to a serial spin lock execution on transient aborts.

However, even with these aborts, applying lock elision to a simple spin lock provides good performance across a wide range of insert percentages. Further, retrying on transactional aborts is fairly effective, especially on conflicts.

4.3. Delta Storage Index Results

Figure 6 shows the relative performance of the Delta Storage Index with various lock implementations (See Section 4.1) for different percentages of inserts. The performance is normalized to the runtime of the spin lock with 0% insert operations. Figure 6a shows a 4-thread configuration (HT disabled) and Figure 6b shows the 8-thread configuration (HT enabled).

Unlike with the B-Tree plots, the TSX-R and TSX-H executions for Delta Index encounter repeated capacity aborts. As a result, they see a performance degradation with respect to the RW Lock even with 0% inserts because the fall back spin lock serializes execution even for readers. For example, for 0% inserts in the 8-thread TSX-R plot, 2.5% of the traversals eventually resulted in a lock acquisition and thus serialization (for TSX-R, the execution retries multiple times before acquiring the lock). In contrast, the RW lock does not have to wait.

Additionally, these executions also see high conflict abort rates. For example, for 100% inserts in the 8-thread TSX-R case, nearly 75% of the traversals eventually resulted in a lock acquisition, thus serializing execution. This was surprising since the tree itself should not have such high conflicts. We discuss this further in Section 5.2

The performance impact is much more pronounced in the 8-thread configuration and particularly for the TSX-H plot. To understand this better, we add an additional plot, TSX-R-no-retries. This is a variant of TSX-R but without retries; on a transactional abort the thread immediately acquires the lock. As expected, TSX-H and TSX-R-no-retries are very similar in profile since they are frequently falling back to acquire the spin lock. Interestingly, even with such high abort rates, the TSX-R-no-retries and TSX-H still perform better than the RW lock with increasing insert percentages. Aborts do not always imply poor performance; a thread would have otherwise simply been waiting to acquire a lock. TSX-R, which retries lock elision following a transactional abort, achieves a good speedup compared to the RW lock for most of the insert percentages except for the 0% insert case and breaks even at around the 2% insert case.

This demonstrates the importance of retries. Even though the executions experience aborts due to capacity and conflict, often these aborts are transient. Even capacity aborts can be transient if the footprint is near the edge and the re-execution may experience a slightly different cache miss pattern due to simultaneous threads sharing the same core, or due to interactions with the data cache's replacement policies. Hence we don't automatically skip retry even when the retry bit is 0. On the other hand, if retries on capacity aborts are not managed effectively, they could result in an additional slow down. By retrying, the threads avoid acquiring the spin lock and continue to have lock elision opportunity. This is why TSX-R is able to outperform the RW lock on most of the insert percentages.

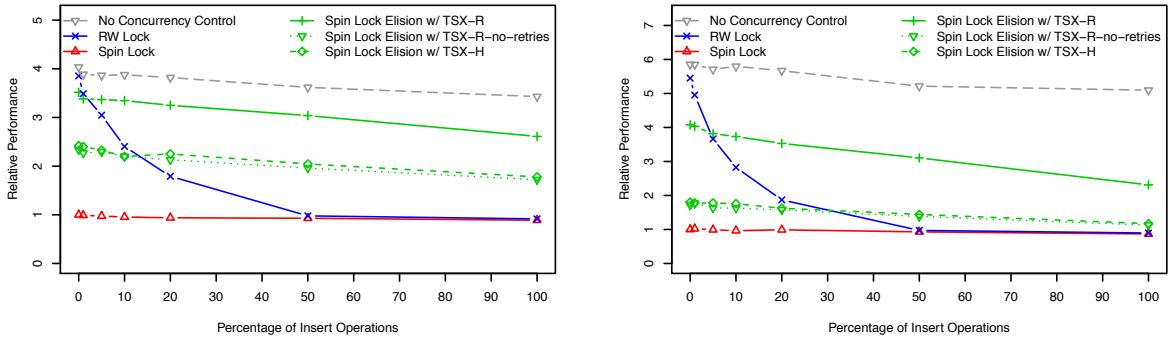
However, the gap with the upper bound version without concurrency control remains large, thus suggesting additional optimization opportunities.

5. Reducing Transactional Aborts

In this section, we investigate the causes for high abort rates with the Delta Storage Index and propose mitigations to make it more elision friendly. Two primary causes were: the traversal's large data footprint and data conflict aborts even when the data shouldn't conflict.

5.1. Reducing Data Footprint

The L1 data cache is 32 KB in size and has 512 cache lines. However, transactional aborts can occur with much smaller data footprints, primarily due to limited L1 cache set associativity and other transient interactions. This is exacerbated for the Delta Index. Even though the Delta Storage Index uses a B+Tree, its implementation also includes a dictionary. There-



(a) 4 Threads (HT disabled)

(b) 8 Threads (HT enabled)

Figure 6: Delta Storage Index Performance with Lock Elision

fore, the access operations are much more involved than for the conventional B+Tree and have a significant data footprint. Next, we analyze the probability of capacity aborts during the Delta Storage Index traversal.

5.1.1. Footprint Analysis Assume a large tree with a height $h = 6$ and a branching factor $b = 31$. Such a tree can store up to 859 million IDs. For the Delta Storage Index, this translates to internal nodes of $Ni = 6$ cache lines and leaf nodes of $Nl = 3$ cache lines. Each ID in the nodes results in a dictionary lookup. For the worst case, when the tree is filled 100%, a tree search traversal would read:

$$Ni*(h-1)+Nl+h*(b-1) = 6*5+3+6*30 = 213 \text{ cache lines}$$

Despite the worst case of 213 cache lines, an empirical analysis shows that a typical Delta Storage Index traversal accesses an average of 93 cache lines. In contrast, the B+Tree traversal typically accesses 20 cache lines. For an HT enabled configuration, 213 cache lines in Delta Index access would result in a worst case 426 cache lines being transactionally accessed using the shared L1 data cache, with an average of around 186 cache lines. This is in contrast to the worst case of 40 cache lines for the B+tree traversal.

Even though most memory accesses such as dictionary lookups and node accesses are random, they are not perfectly distributed. As a result, memory accesses map to some sets of the cache more frequently. This increased pressure results in increased transactional aborts.

This is similar to the bins and balls problem [3] where balls are placed in random bins with a certain capacity. The first bin overflow happens before all bins are filled equally. There are always some bins that are used more often than others. We use the same approach to model the pressure on the L1 data cache to see the probability of evictions. We modeled two scenarios – a 8-way set-associative cache and a 4-way set-associative cache of half the size. The 4-way cache configuration is a good representation of the behavior of a 8-way with HT enabled assuming that all threads run the same type of queries. We use discrete simulation with a random number generator to determine which entry (bin) an access (ball) maps to.

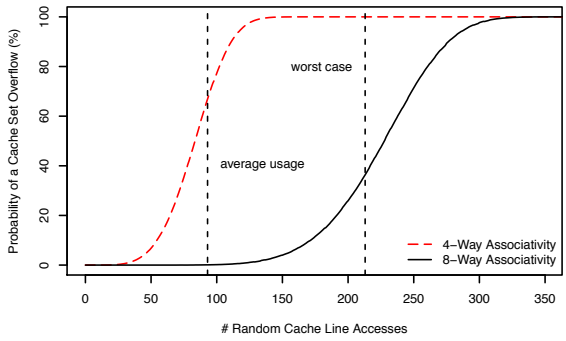


Figure 7: Capacity Probability Analysis

Figure 7 shows the analysis. For both configurations, the overflow probability rises from near zero to 100% when more random cache lines are buffered. For the 4-way cache, the overflow probability starts rising with less cache lines than for the 8-way cache, due to the smaller set-associativity and cache size. The cache line accesses are illustrated with the vertical dotted lines (average and worst case). With average cache line accesses, the capacity overflow probability remains small for the 8-way cache, but for the 4-way cache it is about 60%. The overflow probability for the worst case is significant in both scenarios. Additionally, the target transactional abort rate may not be 50%, but possibly in the 20% range. In such cases, the effective capacity is around one-third of the best case capacity. The effective abort rate cannot be derived easily from the curves as we also need the distribution and pattern of the working set. The simplest and quickest way is to run the application on the target hardware and collect information using the hardware performance monitoring support or additional code in the fallback handler in the case of TSX-R.

This increased pressure on a few sets suggests hardware improvements to handle such cases can mitigate this effect.

5.1.2. Reducing Dictionary Lookups Reducing data footprint through changes in the data structures layout is generally a hard problem. With random queries it is difficult to control the address access pattern itself. This suggests a higher level

transformation to reduce data footprints, specifically a change to the traversal algorithm itself. Instead of the linear search used by default, a binary search would reduce the accessed cache lines to 47 for an average case, with a worst case of 53 cache lines; a significant reduction in the data footprint, all without changing the data structure itself.

If nodes have a high branching factor, then a binary search improves performance. However, for small nodes, a binary search introduces branch operations and additional copy and store operations. Such operations can be more expensive than a simple loop employed by a linear search. The binary search avoids a much larger number of dictionary lookups which tend to be random accesses with poor cache locality. The negative effects are more than compensated due to the significant reduction in cache footprint and the resulting overflow probabilities (less than 10% for a 4-way cache).

5.2. Reducing Avoidable Data Conflicts

Our analysis pointed to another source of unexpected data conflicts even though the tree accesses were not conflicting: the dictionary and the memory allocator.

5.2.1. Dictionary Conflicts Even though the probability of an insert conflict in the tree structure is low, conflicts on the dictionary itself were causing excessive transactional aborts. While the index tree is the primary data structure, an index tree update also updates the dictionary. The dictionary is an array of elements globally available to all threads. On insert, a size variable is incremented and the value is written to the end of the array. Writing the size variable and the array position causes a data conflict when done simultaneously.

A simple transformation converts the global dictionary into a per-core dictionary and ensures conflicts only when the same tree node is being inserted to. Instead of maintaining one dictionary for the whole tree, the tree keeps one dictionary per core. The core affinity of a thread is used as distribution function. This function assigns the dictionary where a thread should insert a value.

Figure 8 shows the assignment for the four cores. Queries run on Core 0 and Core 3. Read operations are free to read from every dictionary depending on their tree path. Only appending an ID is assigned to a core-local dictionary. To identify the position of an entry, the ID is extended to encode the position in the dictionary and the dictionary number. During a lookup, the ID is evaluated to find the appropriate dictionary and the position of the entry within the dictionary. A data conflict is triggered by hardware and the lock is only acquired when the same tree node is being inserted to.

5.2.2. Memory Allocator Conflicts Insert operations can cause node creation. When a node is split, the new node allocation requires new memory. The allocator has either pre-allocated memory in a memory pool or new memory has to be retrieved from the system. Accessing pre-allocated memory results in updates to size variables in the allocator, which can cause transactional aborts if done simultaneously.

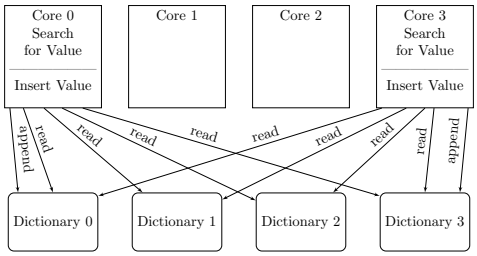


Figure 8: A Per-Core Dictionary Example.

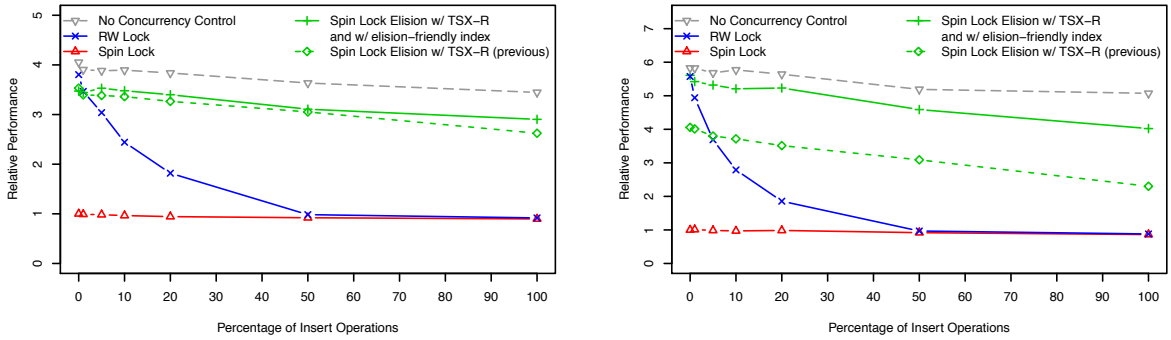
To address this, we use one memory allocator per core [6] and avoid global memory variables. Each allocator has its own memory pool. The core affinity of a running thread assigns an allocator, similar to the dictionary per core approach.

5.3. Tuned Delta Storage Index Results

Figure 9 shows the relative performance of the Delta Storage Index with various lock implementations (See Section 4.1) for different insert percentages. The performance is normalized to the runtime of the spin lock with 0% insert operations. Figure 9a shows a 4-thread configuration (HT disabled) and Figure 9b shows the 8-thread configuration (HT enabled). The plot corresponding to the label "Spin Lock Elision w/ TSX-R and w/ elision-friendly index" uses a Delta Storage Index implementation with the transformations described in Sections 5.1 and 5.2. The remaining plots continue to use the original Delta Storage Index implementation without these transformations. That configuration represents the best performance for the RW Lock and Spin Lock for reasons discussed in Section 5.1.

In Figure 9a we can see that TSX-R with the elision-friendly Delta Storage Index performs slightly better than the original index version, especially in the 100% insert case. This is due to a reduction in conflicts through changes in the dictionary and memory allocation implementations. In the TSX-R configuration for an insert rate of 100%, the percentage of traversals that eventually resulted in a lock acquire is now 1.7% as compared to 3.6% prior to the transformations. Recall that TSX-R on an abort retries elision multiple times prior to a threshold after which it acquires the lock.

In Figure 9b we see a significant improvement to TSX-R with an elision-friendly Delta Storage Index. The improvement is similar for all percentages of insert operations, demonstrating the effectiveness of the data footprint reduction. With the elision-friendly index, we achieve significant performance gains for all insert percentages. For an insert rate of 100%, the elision-friendly Delta Storage Index achieves a speedup of 4.6x compared to the best performing RW lock implementation. The relative gain from these improvements is higher for the 100% insert than for the 0% insert, demonstrating the benefit of conflict reductions from the insert operation. For example, in the 100% insert case with TSX-R, the percentage of traversals that eventually resulted in a lock acquire (and hence serialization) is now only 1.3% as compared to nearly 75% prior to the transformations (Section 4.3).



(a) 4 Threads (HT disabled)

(b) 8 Threads (HT enabled)

Figure 9: Elision-Friendly Delta Storage Index Performance

As can be seen, applying the transformations has brought the TSX-R line close to the upper bound version without any concurrency control. Even though the serialization rate due to lock acquires has been significantly reduced, aborts still occur. Further opportunities for reducing aborts and bridging the remaining gap is future work.

Figure 10 shows scaling of TSX-R with increasing thread count for an elision-friendly Delta Storage Index. We compare TSX-R (using a spin lock) and a RW lock at two ends of the insert spectrum: read-only (0% inserts) and write-only (100% inserts). The relative performance is scaled over the performance for the single thread RW lock. As we can see, TSX-R performance is slightly lower than RW locks for the read-only case (for reasons discussed in Section 4.3) but significantly outperforms the RW lock for the write-only case. In the 4-thread data point, each thread is assigned to a separate core (HT disabled) whereas in the 8-thread data point, two threads are assigned to each core (HT enabled). Even with 8 threads running with HT enabled, we see good scaling with respect to 4 threads running with HT disabled.

The study above was for up to 8 threads as that was the largest available configuration. The performance gap to the RW Lock is expected to be even larger with increasing cores and additional sockets, as is typical in server platforms. For multi-socket systems we expect TSX-R to outperform the RW lock in read-only scenarios as well, because TSX-R avoids moving the cache line with the lock variable around in the system and thus it will have a positive effect on system performance (as shown in the Figures 3c and 4c).

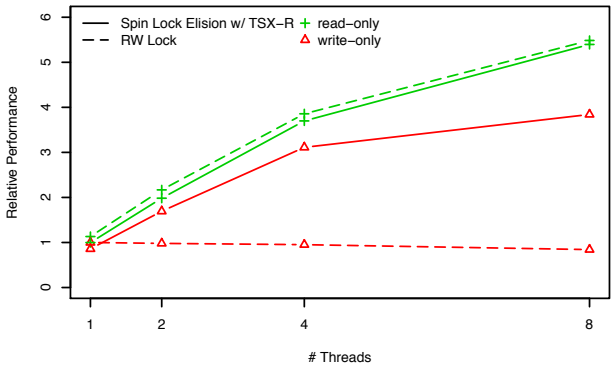


Figure 10: Delta Storage Index Scaling with Intel TSX

6. Related Work

Database indexes [4] have been extensively studied and over the years numerous concurrency control implementations for these indexes have been proposed. These proposals, both research and commercial, vary greatly in implementation. But nearly all implementations rely on some form of locking or latching protocols, whether to coordinate concurrent transactions accessing a database or to coordinate concurrent threads accessing the underlying data structures. An index traversal often requires following pointers or links, and the implementation must ensure such a pointer does not become inconsistent in the middle of a traversal by one thread due to the actions of another thread. If multiple locks are used to protect various nodes, an implementation must follow proper rules for deadlock avoidance or support detection and recovery mechanisms. Some operations on trees, such as inserts, can eventually result in a change in the tree's structure, and dealing with this in a high concurrency environment is fairly subtle. Graefe [10] provides an excellent survey of different approaches.

Index implementations that use simple concurrency control, such as a reader-writer lock, while easier to verify, don't perform well under heavy writes, since writers serialize accesses to the index. Adding levels of granularity by employing multiple locks helps improve concurrency but increases complexity. Example of finer granularity implementations include Lock Coupling [5] where a parent node must be kept locked until a child node is locked, and Lock Splitting where a set of locks protect different nodes of the index. Lock free implementations, while avoiding high-level transaction locks when accessing indexes, often still rely on low level synchronization operations and, as we saw earlier, such operations can also hurt scalability in modern server systems. Other approaches,

instead of protecting the actual structure, focus on higher-level mechanisms such as key range locking.

Alternative approaches include B-Link trees [16] where the traversal algorithm for reads is made tolerant to changes in the underlying links by any writers. This avoids synchronization for readers at the cost of additional constraints on the underlying data structure (e.g., the size of the data element), and complexity in dealing with an optimistic traversal in software. Numerous extensions to B-Link trees have also been proposed [21, 11]. A versioning approach shows similar properties as the B-Link tree [8]. PALM [22] relies on batching queries following a bulk synchronous parallel algorithm approach. Kuszmaul et al. use a Software Transactional Memory (STM) implementation to allow concurrent accesses to a cache-oblivious B-Tree [15]. Performance is limited through the computational overhead of the STM system.

Herlihy and Moss [13] introduced Transactional Memory and proposed hardware support to simplify the implementation of lock-free algorithms. Because Transactional memory requires a new lock-free programming model and relies on mechanisms other than locks to provide concurrency, integrating it into complex software systems remains a significant challenge [12]. Rajwar and Goodman [19, 20] introduced lock elision as a way to execute lock-based programs in a lock-free manner. Similar hardware support has been investigated for other domains and data structures [24, 9].

7. Concluding Remarks

In this paper we addressed the question of whether modern database implementations can take advantage of hardware support for lock elision. We evaluated the effectiveness of such hardware support using the Intel TSX extensions available in the Intel 4th Generation Core™ Processors. We focused on database index implementations, a common B+Tree and the SAP HANA Delta Storage Index.

We demonstrated that a simple spin lock with lock elision can provide good scalability across a wide range of access configurations. While the gains can be observed without any changes to the index implementations, making elision friendly changes can further improve performance. We give examples of some of these transformations. The resulting index concurrency control implementation is simple and scalable with lock elision, while performance is close to an index tree without any concurrency control implemented.

The results are quite encouraging and suggest compelling benefits to internal database data structures from Intel TSX-based lock elision. Applying Intel TSX to other internal database data structures is an area of future work.

8. Acknowledgments

This work is partly supported by the German Research Foundation (DFG) within the Cluster of Excellence “Center for Advancing Electronics Dresden" and by the EU and the state Saxony through the ESF young researcher group “IMData".

References

[1] Intel 64 and IA-32 Architectures Optimization Reference Manual. 2013. Intel Corporation.

[2] Intel 64 and IA-32 Architectures Software Developer Manual. 2013. Intel Corporation.

[3] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced Allocations. *SIAM J. Comput.*, 29(1):180–200, Sept. 1999.

[4] R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indices. In *Proceedings of the 1970 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*, 1970.

[5] R. Bayer and M. Schkolnick. Readings in Database Systems. chapter Concurrency of Operations on B-Trees. 1988.

[6] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[7] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance Pathologies in Hardware Transactional Memory. In *Proceedings of the 34th Annual International symposium on Computer Architecture*, 2007.

[8] S. K. Cha, S. Hwang, K. Kim, and K. Kwon. Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems. In *Proceedings of the 27th International Conference on Very Large Data Bases*, 2001.

[9] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early Experience with a Commercial Hardware Transactional Memory Implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[10] G. Graefe. A Survey of B-Tree Locking Techniques. *ACM Trans. Database Syst.*, 35(3), 2010.

[11] G. Graefe, H. Kimura, and H. Kuno. Foster B-trees. *ACM Trans. Database Syst.*, 37(3):17:1–17:29, Sept. 2012.

[12] T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[13] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Freed Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 1993.

[14] M. D. Hill. Is Transactional Memory an Oxymoron? *Proceedings of the VLDB Endowment*, 1(1):1–1, Aug. 2008.

[15] B. C. Kuszmaul and J. Sukha. Concurrent Cache-Oblivious B-trees using Transactional Memory. *Workshop on Transactional Memory Workloads*, 2006.

[16] P. L. Lehman and S. B. Yao. Efficient Locking for Concurrent Operations on B-trees. *ACM Trans. Database Syst.*, 1981.

[17] R. Rajwar and P. A. Bernstein. Atomic Transactional Execution in Hardware: A New High-Performance Abstraction for Databases? In *The 10th International Workshop on High Performance Transaction Systems*, 2003.

[18] R. Rajwar and M. G. Dixon. Intel Transactional Synchronization Extensions. *Intel Developer Forum San Francisco 2012*, 2012.

[19] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, 2001.

[20] R. Rajwar and J. R. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[21] Y. Sagiv. Concurrent Operations on B-Trees with Overtaking. In *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 1985.

[22] J. Sewall, J. Chhugani, C. Kim, N. R. Satish, and P. Dubey. PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors. *Proceedings of the VLDB Endowment*, 4(11):795–806, 2011.

[23] N. Shavit and D. Touitou. Software Transactional Memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, 1995.

[24] R. Yoo, C. Hughes, K. Lai, and R. Rajwar. Performance evaluation of intel transactional synchronization extensions for high performance computing. In *Proceedings of the 2013 ACM/IEEE Conference on Supercomputing*, Supercomputing '13, 2013.