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

Tomas Karnagel (TU Dresden)
Roman Dementiev (Intel Corporation)
Ravi Rajwar (Intel Corporation)
Konrad Lai (Intel Corporation)
Thomas Legler (SAP AG)
Benjamin Schlegel (TU Dresden)
Wolfgang Lehner (TU Dresden)

HPCA 2014, February 18, 2014

Challenge: Implementing Concurrency Control

High-performance in-memory databases

- Extensively use low-level synchronization mechanisms
- Synchronize access to internal in-memory data structures

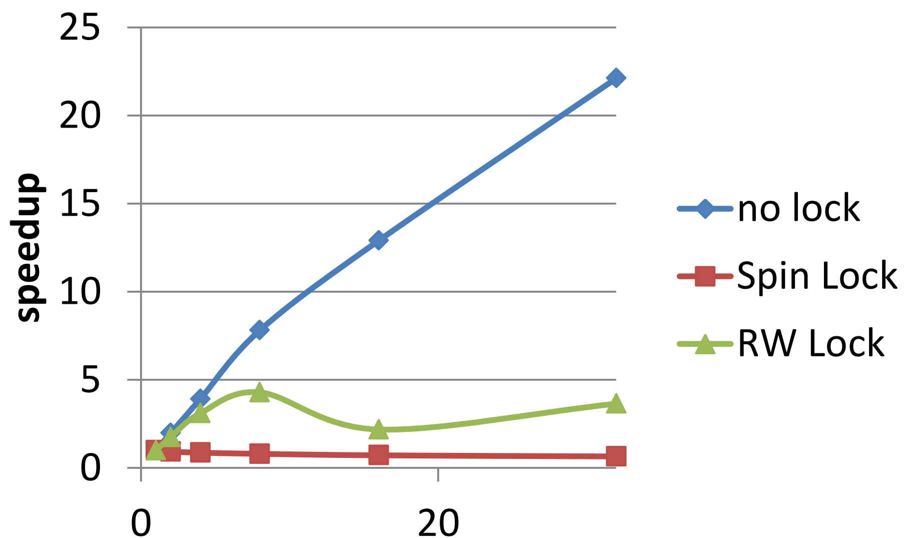
Rich history with numerous proposals and implementations

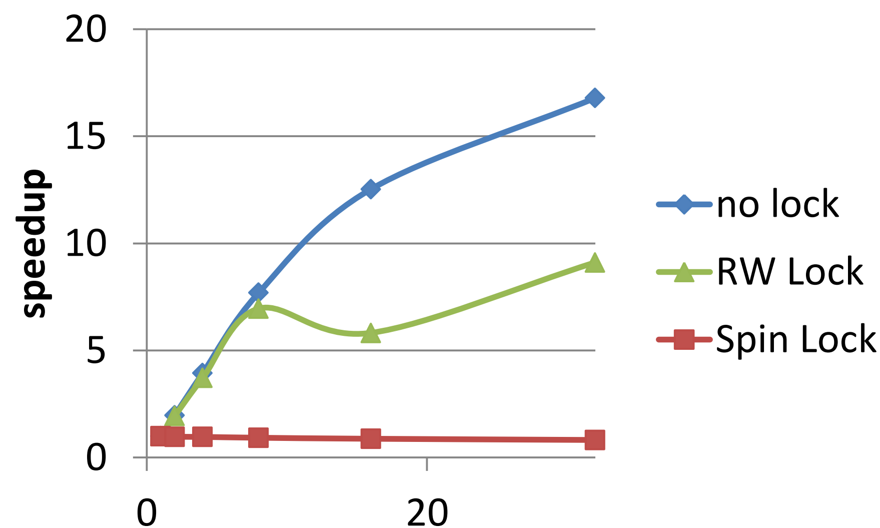
- Remains complicated and difficult to verify/deploy/productize
  - Trade-off – complexity/verifiability
  - Latches/Locks remain popular
- Consumes significant developmental effort and resources

"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."

- Graefe, 2010

A Case Study: Two Index Implementations





Read-Only Queries on Dual Socket Intel® Xeon® E5-2680 Server

Hidden Scalability Impact of Atomic Read-Modify-Write Operations

Hardware Trends

Hardware support for Transactional Synchronization

- IBM System z CPU (zEC12)
- 4th Generation Core ™ Processors (codenamed Haswell)

Goal

- Enable development of simple and scalable easy-to-verify implementations
- Without requiring new programming models and paradigms

Question

- How can a commercial database exploit such hardware?
- Without requiring large scale changes?

Can Modern Databases Take Advantage of Hardware Trends?

Outline

Challenge and Trends

Intel Transactional Synchronization Extensions (Intel TSX)

Case Study: SAP HANA index implementation

Applying Intel TSX

Analysis and Transformations

Results

Lessons and Summary

What is Intel® TSX?

Hardware support to enable Lock Elision

- Focus on lock granularity optimizations
- Fine grain performance at coarse grain effort

Intel® TSX: Instruction Set Extensions for Intel Architecture

- Transactionally execute lock-protected critical sections
- Execute without acquiring lock → Expose hidden concurrency
- Hardware manages transactional updates – All or None
  - Other threads cannot observe intermediate transactional updates
  - If lock elision cannot succeed, restart execution, and acquire lock

Improves the Well-Understood Lock Based Programming Model

Intel® TSX Interfaces for Lock Elision

Hardware Lock Elision (HLE) – XACQUIRE/XRELEASE

- Software uses legacy compatible hints to identify critical section.
  - Hints ignored on hardware without TSX
- Hardware support to execute transactionally without acquiring lock
- Abort causes a re-execution without elision
- Hardware manages all architectural state

Restricted Transactional Memory (RTM) – XBEGIN/XEND

- Software uses new instructions to specify critical sections
- Similar to HLE but flexible interface for software to do lock elision
- Abort transfers control to target specified by XBEGIN operand
- Abort information returned in a general purpose register (EAX)

Outline

Challenge and Trends

Intel Transactional Synchronization Extensions

Case Study: SAP HANA index implementations

Applying Intel TSX

Analysis and Transformations

Results

Summary

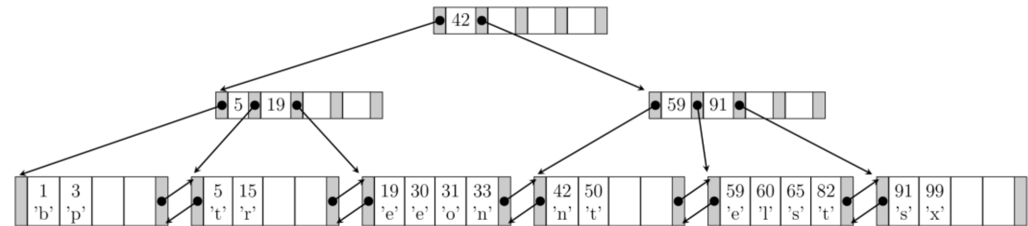
Case Study: Index Tree Implementations

SAP HANA Database

- Read optimized column store database system

Two index implementations

- B+Tree [Data Structure]
  - Common index implementation
  - Smaller foot print

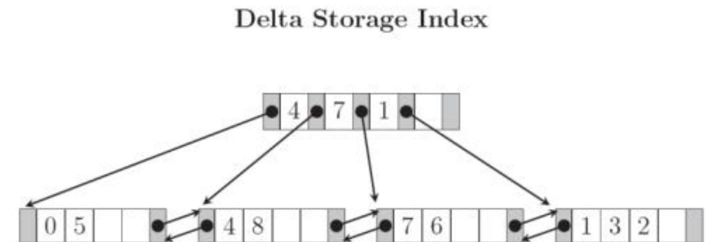


- Delta Storage Index (B+Tree with a Dictionary)
  - Complex data structure with additional support structures
  - Large foot print

Lock protect access

- **Reader-Writer**
- Spin Lock

Column Data	Dictionary ID	Dictionary Entry
1	0	Bakersfield
7	1	San Francisco
2	2	Santa Clara
3	3	San Jose
1	4	Fresno
0	5	Berkeley
8	6	Sacramento
2	7	Palo Alto
...	8	Oakland



Steps To Apply Intel TSX

Modify synchronization library to use Intel TSX for lock elision

- Use either the HLE or RTM interface
- Application changes not immediately required

RTM: Start with good “fallback handler” design

- Avoid “the lemming effect”
- Retry appropriately for both conflict and capacity aborts

Analyze early results

- Extensive hardware profiling infrastructure

Apply transaction-friendly transformations if needed

- To reduce conflict and capacity aborts

Lemming Effect: Persistent convoy of non-transactional execution

```
XA      xbegin; test; xabort; (retry loop when lock is busy)

L—U     Lock; critical section; Unlock (non-transactional execution)

T1 --AL

T2 ---A

T3 ---A
```

- Happens when a thread aborts and acquires the lock
  - And aborts other threads

Lemming Effect: Persistent convoy of non-transactional execution

```
XA       xbegin; test; xabort; (retry loop when lock is busy)

L—U      Lock; critical section; Unlock (non-transactional execution)

T1 --AL--------

T2 ---AXAXAXAXA

T3 ---AXAXAXAXA
```

- Happens when a thread aborts and acquires the lock
  - And aborts other threads
- And other threads immediately retry transactional execution
  - Find the lock held and abort
    - And immediately retry transactional execution

Lemming Effect: Persistent convoy of non-transactional execution

```
XA       xbegin; test; xabort; (retry loop when lock is busy)

L—U      Lock; critical section; Unlock (non-transactional execution)

T1 --AL------------UXAXAXAXAXAsssssss

T2 ---AXAXAXAXAXAsssL------------UXAX

T3 ---AXAXAXAXAXAsssssssssssssssssL--
```

- Happens when a thread aborts and acquires the lock
  - And aborts other threads
- And other threads immediately retry transactional execution
  - Find the lock held and abort
    - And immediately retry transactional execution
  - Reach their retry limit
    - And enter non-transactional fall-back execution, trying to acquire lock

Lemming Effect: Persistent convoy of non-transactional execution

```
XA       xbegin; test; xabort; (retry loop when lock is busy)

L—U      Lock; critical section; Unlock (non-transactional execution)

T1 --AL------------UXAXAXAXAXAssssssssssssssssL----------UXAXA

T2 ---AXAXAXAXAXAsssL------------UXAXAXAXAXAXAssssssssssssL---

T3 ---AXAXAXAXAXAsssssssssssssssssL----------UXAXAXAXAXAssssss
```

- Happens when a thread aborts and acquires the lock
  - And aborts other threads
- And other threads immediately retry transactional execution
  - Find the lock held and abort
    - And immediately retry transactional execution
  - Reach their retry limit
    - And enter non-transactional fall-back execution, trying to acquire lock
  - And never actually had any opportunity to elide the lock
- Elision is effectively disabled until all threads have serially released the lock
  - Disabled forever if at least 1 thread is holding the lock

Lemming Effect

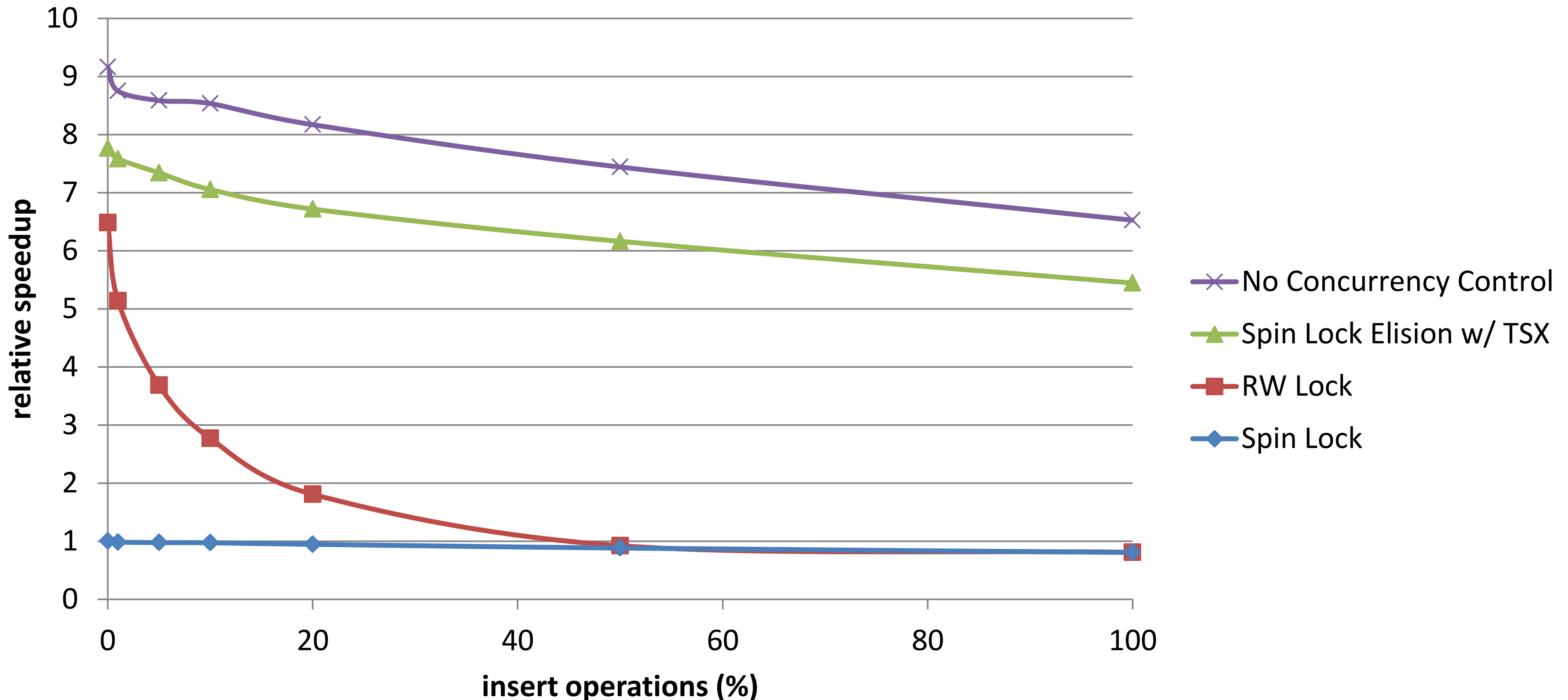
Fix: Don’t retry until the lock is free (test-and-test-&-set)

```
T1 --AL------------UX-------

T2 ---AsssssssssssssX-------

T3 ---AsssssssssssssX-------
```

Initial Results: B+Tree

Intel TSX provides significant gains with no application changes

- Outperforms RW lock on read-only queries
- Significant gains with increasing inserts (6x for 50%)



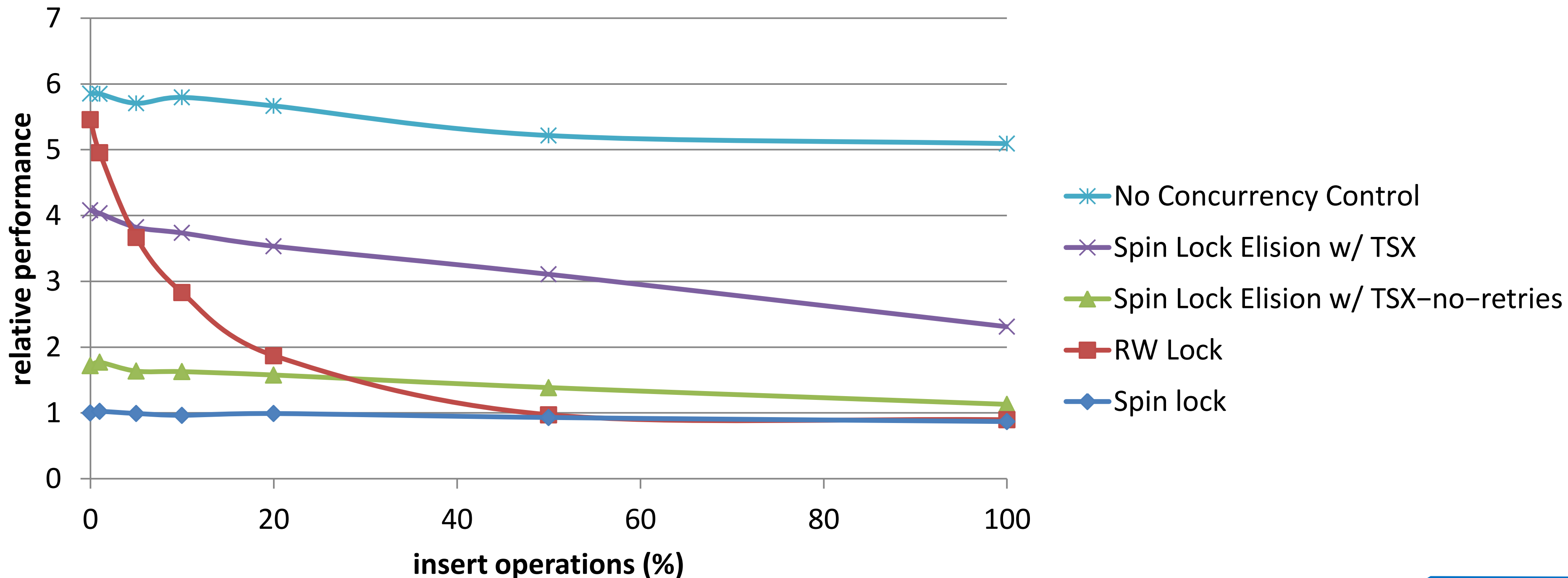
Intel® Core™ i7 processor with 4 physical cores / 8 logical cores (HT)

Initial Results: Delta Storage Index

Intel TSX provides gains with no application changes

- Different profile as compared to B+Tree
- Spin lock w/ Intel TSX better than RW Lock when > 5% insert
  - Significant gap as compared to no concurrency control

Baseline should implement good retry policy on aborts

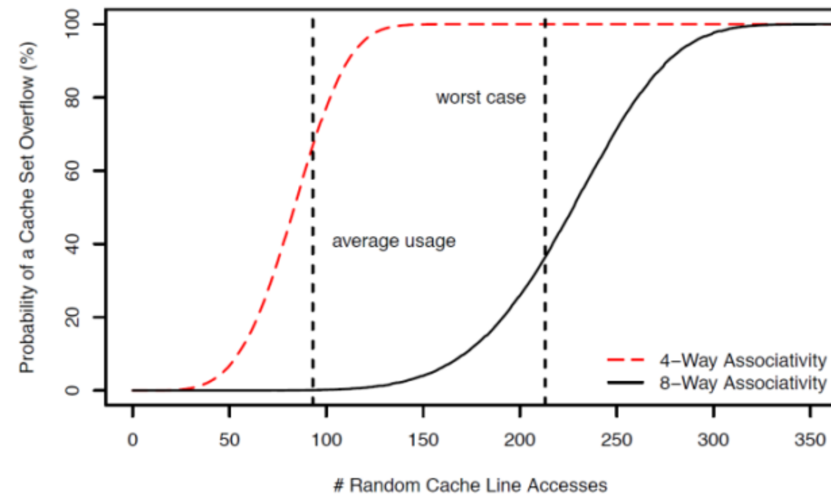


Intel® Core™ i7 processor with 4 physical cores / 8 logical cores (HT)

Analysis: Delta Storage Index

Capacity Aborts

- Algorithmic level
  - Node/Leaf Search Scan
  - Causes O(n) random lookups
- Cache Associativity Limits
  - Aborts typically before cache size limits
  - Hyper-threads share the L1 cache
- Dictionary contributes to larger footprint



Data Conflicts

- Single dictionary
- Global memory allocator

Well Known Causes of Transactional Aborts

Software Transformations

Capacity Aborts

- Node/Leaf Search Scan
  - Causes O(n) random lookups
- Transformation – Binary Search
  - Causes O(log(n)) random lookups

Data Conflicts

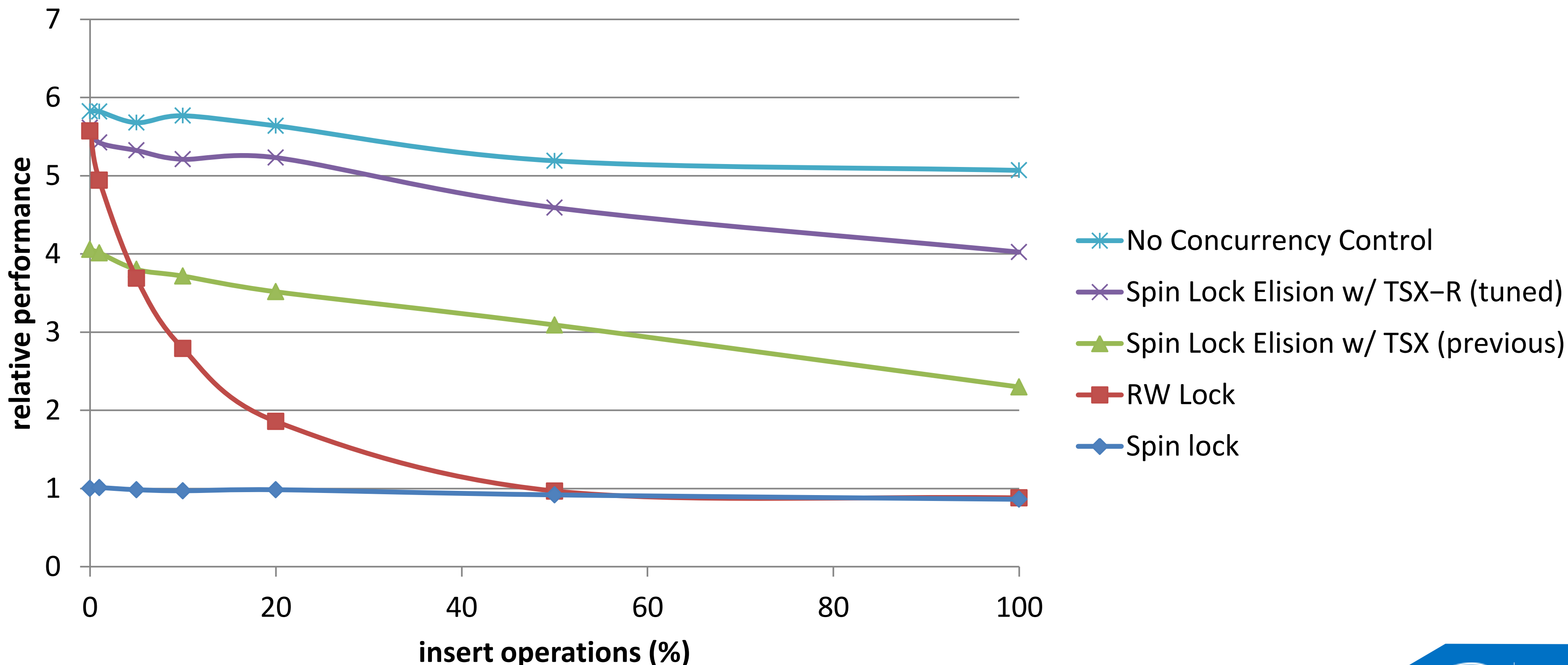
- Single dictionary
- Global memory allocator
- Transformation – Multiple Dictionaries, per-thread/core allocators

Well Known Transformations

Tuned Results: Delta Storage Index

Intel TSX provides significant gains with transformations

- Restores read-only query performance
- Spin lock w/ Intel TSX significantly outperforms RW lock (5x for 50% inserts)
- Close to ‘No Concurrency Control”



Intel® Core™ i7 processor with 4 physical cores / 8 logical cores (HT)

Lessons

Lemming Effect

- A common but deadly mistake
- Simple fixes

Capacity Limits

- Best determined by running actual workloads
- Transient aborts due to associativity

Use Analysis Tools and Iterate Optimizations

- Extensive infrastructure
- Can pinpoint areas of interest effectively

Retry on Aborts

- Limited retries are effective – even on capacity aborts
- Abort rates can be misleading

Summary

Atomic read-modify-write operations can limit scalability

- Even for read only transactions using reader-writer locks
- An often overlooked performance issue

Database data structures compelling candidates for Intel TSX

- SAP HANA Delta Storage Index
- B+Tree

Intel TSX-based lock elision can provide excellent scalability

- Provides gains even for unmodified applications
- Additional gains with simple well-known software transformations

Great MT performance - Delta Storage Index

- **2x** for 10% insert to **~5x** for >50% insert