Exploiting Hardware Transactional Memory to Accelerate Concurrent Spatio-Temporal Indexes

Nuno Henrique Nina Ribeiro Elvas Fangueiro
nuno.fangueiro@ist.utl.pt

Instituto Superior Técnico, Lisboa, Portugal

November 2017

Abstract

This thesis focuses on how to exploit Transactional Memory (TM) to accelerate applications that target big spatio-temporal data. TM has emerged as a promising abstraction for parallel programming, which aims at enhancing performance and simplify programming of concurrent applications. Specifically, we use Hardware Transactional Memory (HTM) as a synchronization alternative to conventional locking for main-memory spatio-temporal indexing data structures and seek an answer to the following research questions: i) what efficiency levels can be achieved by applying HTM to state of the art *single-threaded* (i.e., non-thread safe) spatio-temporal indexes algorithms? In particular, how does the performance of such HTM-based algorithms compare with state-of-the-art *concurrent* algorithms, designed from scratch to cope with the consistency issues arising in multi-threaded environments? ii) to what extent can HTM be applied to state-of-the-art concurrent indexing algorithms for spatio-temporal data, in order to enhance their efficiency?
Keywords: Spatio-Temporal Data, Spatio-Temporal Index Structures, Concurrency Control Schemes, Transactional Memory, Performance

Introduction

The problem addressed in my thesis is to study efficient ways to enable concurrent access to spatio-temporal indexes, in order to take full advantage of modern multi and many core architectures. Moreover, this thesis is framed in the context of a more general trend, which has focused on how to exploit recent hardware advances to accelerate big data processing. In particular, we plan to study how to exploit hardware implementations of Transactional Memory (TM) to allow concurrent access to spatio-temporal index in a multi-threaded environment.

TM has emerged as a promising abstraction for parallel programming, which aims at enhancing performance and simplifying programming of concurrent applications when deployed on modern parallel systems. TM represents an alternative to the traditional approach for regulating concurrency in a multi-threaded program, i.e., locking. However, the use of fine-grained locking is known to be quite complex, even for experienced programmers [12, 3], and to lack one important property that is fundamental in modern software engineering approaches: composability [8].

In contrast, TM is a much more straightforward approach to building concurrent software, since all code that has to execute atomically has simply to be wrapped within a transaction. The underlying TM implementation transparently ensures atomicity, making programmers life much easier. Further, locks use a pessimistic approach, which ensures correctness by restricting parallelism. Conversely, TM allows to fully untap the parallelism offered by modern multi-core architectures by adopting an optimistic approach that allows atomic code blocks to be executed in a speculative fashion, aborting execution only in case conflicts are actually detected.

More in detail my thesis seeks to answer two main questions:

1. what efficiency levels can be achieved by applying HTM to state of the art *single-threaded* (i.e., non-thread safe) spatio-temporal indexes algorthms? In particular, how does the performance of such HTM-based algorithms compare with state-of-the-art *concurrent* algorithms, which rely on complex, and carefully optimized, fine grained locking schemes?
2. can HTM be applied to state-of-the-art concurrent lock-based indexing algorithms for spatio-temporal data, in order to enhance their efficiency?

We answer these questions by conducting an extensive experimental evaluation considering 3 different architectures: Intel Core [7] Haswell and Broadwell, and IBM POWER8 [17] CPUs. Hence, we are able to use multiple HTM interfaces, which result in multiple HTM implementations. Moreover, we consider realistic workloads, generated using standard

benchmarking tools that allow to faithfully capture the characteristic of real life workloads by simulating e.g., traffic, using a network where object can move threw and oblige to its rules.

As a preliminary step, before addressing the two aforementioned questions, we conduct a systematic study on the tuning of some key parameters and runtime libraries that are known to affect significantly HTMs performance: transactional retry logic and implementation of the dynamic memory allocator (TCMalloc [5]). We come to the conclusion the standard GNU C library (Glibc) [11] memory allocator is the best when coupled with non-HTM indexes. Moreover, its does not falter as TCMalloc with PGrid with HTM (PGridHTM) [16], hence, we conclude that in general the Glibc memory allocator is the more well suited to handle our spatio-temporal indexes. The optimal configuration for the transactional retry logic is platform depended. Nevertheless, our results show that a 20 retry value is able to satisfy performance in all machines. This study is aimed at ensuring the correct tuning of these parameters, for the considered application domain/workloads, so to ensure a fair and representative comparison with other lock-based solutions, which are conducted in the following.

In order to answer the first question, we consider Update efficient Grid (u-Grid) [15] as the target single-threaded algorithm where to apply HTM. Since this algorithm is not suited to handle concurrency, we have to wrap the main operations (update and query) with transactions, which may not be ideal for performance. The other baseline single-threaded algorithm is the Update efficient R-tree (u-R-tree) [15], which we will not be applying HTM to, however, it is a plain comparison with u-Grid. Moreover, we include algorithms that provide concurrency with the same consistency levels as single-threaded algorithms, and we include concurrent algorithms that lower the consistency level in order to provide better parallelism and fresher query results, respectively Serialized Grid (Serial) [16] and Parallel Grid (PGrid). The results of this study shows that u-Grid with HTM (u-GridHTM) is able to achieve performance comparable to state of the art concurrent algorithms that use complex and carefully engineered fine-based locking schemes, specifically Serial and PGrid.

As for the second question, we consider as baseline PGrid, which as confirmed in the first study, has very competitive performance. The main key ideas used to enhance its parallelism via HTM are the following: i) to replace the critical sections with transactions by eliding the locks. ii) to maintain its query semantics by reusing the already present atomic instructions (OLFIT). iii) to deal with the non-tx-friendly synchronization scheme (waiting for readers to become 0). iv) to partition query transactions as to avoid contention and transactional memory overflow. With these implementations we are able to make PGridHTM have better performance over PGrid in query intensive workloads and be the best performing index in update intensive workloads.

The remainder of this document structures as follows. Section 2 provides a background on HTM and its implementations on Intel Core [7] and POWER8 [17]. Section 3 describes the FGL solutions which we intend to modify to HTM. Section 4 describes our solutions. Section 5 presents the extensive experimental study made to our solutions. Finally, Section 6 concludes the dissertation by summarizing the results obtained in the previous section and discusses possible future work.

Hardware Transactional Memory

HTM is a concurrency protocol, which provides the use of atomic operations (transactions) at cache level. This occurs via ad-hoc extensions of the processor instruction set (e.g., TSX in Haswell [18]). HTM can be implemented with the cache coherency protocol of multiple CPUs, with multiple memory architectures as: Non-uniform memory access (NUMA) and uniform memory access (UMA). Thus, HTM may have an exponential performance potential if it suits such architectures.

In commercial HTM implementations, HTM uses the processors' cache to store the meta-data generated by transactional read/writes, and the cache coherence protocol to detect conflicts. Performance is increased since memory operations are made in cache and there is no need for software instrumentation, which greatly reduces overheads. The consequence of this design is the low amount of memory (cache) available to store the metadata (read/writes sets) of transactions. Workloads including big transactions may exceed hardware memory capacity, resulting in transactional aborts and inducing a big overhead. Due to these (and other) limitations, HTM transactions are never guaranteed to complete (best-effort HTMs). A fall back plan (usually resorting to locks) is required to maintain at least serial performance in case a transaction fails repeatedly (and potentially deterministically) in hardware.

Intel Core

Intel Core [7] uses the L1 cache for conflict detection and store buffering. Further research from [14], evaluate that the load and store capacities are 4MB and 22KB, respectively, on Core i7-4770. Moreover, they claim that the load capacity is larger because it uses other resources to track the cache lines that were evicted from the L1 cache. Moreover, transaction capacity for the stores is within the size of the

L1 cache.

The latest Haswell processors made by Intel Core come with a new extension of the instruction set architecture (ISA) [18], which supports Hardware Transactional Memory, called Transactional Synchronization Extensions (TSX). TSX provides two software interfaces to handle HTM, named, Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM).

HLE can be seen as subset of RTM, meant to be backward compatible with processors without TSX support. HLE is able to replace lock implementations with two provided prefixes _XAQUIRE and _XREALEASE. These prefixes are used with locks. When the software acquires the lock, the hardware has the ability to check if a thread executing the critical conflicts with other threads (speculatively). Threads that will not generate conflicts may run in parallel with others. Thus, the lock is elided and threads may run without requiring any communication with the lock. However a conflict might happen for various reasons, in that case threads get rolled back and acquire the lock.

RTM provides three new prefixes XBEGIN, XEND and XABORT to handle transactions (more prefixes are available). The programmer can start, end, or abort a transaction in any part of the program. Another difference in RTM is that programmers must define a fall-back path for an aborted transaction. RTM brings more flexibility to the programmer as he can choose what to do when a transaction aborts and explicitly start or end transactions without requiring locks.

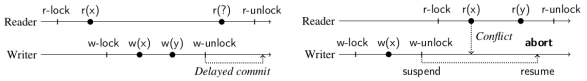
In summary HLE is used for compatibility with legacy processors. In contrast, RTM explicitly enables the programmer to define the transactional critical areas, thus bringing more flexibility. However it requires programmers to define a fall-back path.

POWER8

POWER8 [17] uses content addressable memory (CAM) linked with the L2 cache for conflict detection [9]. This CAM is called the L2 TMCAM. The L2 TMCAM records the cacheline addresses that are accessed in the transactions with bits to represent read and write. Although the transactional stored data is buffered in the L2 cache, the transaction capacity is bounded by the size of the L2 TMCAM. Since the number of the entries for the L2 TMCAM is 64, the total transaction capacity combined for loads and stores is 8 KB (=64*128 bytes), where each cache line size has 128 bytes.

POWER8s default HTM interface already allows the user to specifically set the parameters, __TM_abort and __TM_begin in the code, and define a fall-back path, as RTM. Therefore, allowing a higher level of flexibility to programmers.

Another step towards HTM progress, is a new technique proposed by Issa et al. [4], which uses a hardware-software co-design, based on HTM, to speculatively Read Write locks. Hardware Read-Write Lock Elision (HRWLE) exploits two hardware features of POWER8 processors: suspending/resuming transaction execution and rollback-only transactions (ROT). HRWLE provides two major benefits with respect to existing HLE techniques: i) eliding the read lock without resorting to the use of hardware transactions, and ii) avoiding to track read memory accesses issued in the write critical section.



(a) The write back of shared variables updated by a writer must be delayed until after all readers have completed their critical sections to preserve consistency.

(b) A new reader accessing a shared variable updated by a suspended writer will abort the suspended hardware speculation of the writer upon resume.

Figure 1: Writers quiescence mechanism to ensure consistency

Unlike read/write locks, in HRWLE reads and writes are concurrent. In read critical sections, transactions are completely free, lacking any instrumentation (non-speculative). They only flag their state in shared memory, to indicate whenever a read critical section starts. Since writers execute within HTM transactions, they are protected from any conflict developed with other writers. However, it is still not safe to commit writers with concurrent non-speculative readers, thus, they have to wait until they can commit (Fig. 1 (a)).

To do so, writers use a quiescence call to ensure correctness with readers. The quiescence call is implemented with the suspend/resume hardware feature, which has two primary characteristics. First, it drains all current readers that may touch a writer, so that writers may commit. Second, any reader that tries any memory position updated by a concurrent suspended writer will cause the abort of this writer (Fig. 1 (b)).

Writes have several steps to follow. First, they are issued as with plain HTM (speculative), if a write cannot commit after several attempts, a ROT is issued instead. ROTs lower the semantics of the overall transaction in order to allow writers not to track read memory access. Thus, this transactions have nearly unlimited read capacity, which increases their chances of surviving big transactions. ROT's benefit from read-dominated workloads since read memory accesses usually compromise 80%-90% of the whole memory access [4].

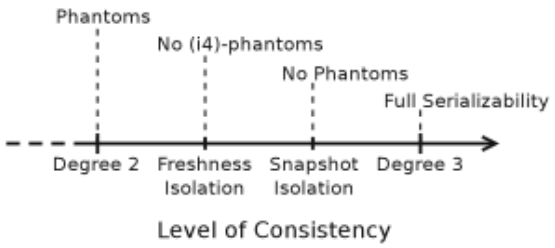


Figure 2: Level of Consistency [16]

However, ROTs themselves can not be concurrent with each other, as they do not track their reads. Nevertheless, here we are opting for a better read/write concurrency scheme, at the cost of serial writers. Finally, after several aborted attempts of a ROT transaction, a lock is grabbed as the transactions ultimate fall-back path.

Concurrent Indexing Algorithms for Spatio-Temporal Data

In this section we focus on concurrent indexing data structures for spatio-temporal data on parallel systems, which will represent the area of work of my thesis. These solutions aim at taking maximum advantage from modern multi and many core architectures, supporting concurrent execution of updates and query operations. As such, they represent important building blocks for several of the distributed platforms for spatio-temporal data.

Semantics and Parallelism

In order to ensure the correctness and efficiency of concurrent data-structures, like the spatio-temporal indexes considered in this work, it is crucial to adequately synchronize the concurrent execution of update and query operations. An ideal synchronization scheme strives to maximize the parallelism achievable when accessing the index, while preserving some target consistency criterion aimed at guaranteeing its correctness.

In literature, 4 different levels of consistency are usually considered for spatio-temporal indexes; Serializable (also called Degree 3), Snapshot Isolation (SI), Freshness Isolation and Degree 2.

Degree 3 is full serializabilty. A transaction schedule is serializable if its outcome (e.g., the resulting state of the system) is equal to the outcome of its transactions executed serially, i.e., without overlapping in time.

HTM provides degree 3 consistency. Even though transactions may be concurrent, their execution is equivalent to a serial one. The cache coherency protocol ensures conflicts are detected, while transactions ensure correctness, hence, it is possible to order transactions as an equivalent serial execution.

The next lower level of consistency is SI. However, we will not make a description of SI since it is not used it the considered indexes, and the extend abstract space is limited.

Finally, Freshness Isolation is a consistency criterion weaker than SI that allows various concurrency anomalies, called phantoms in the literature [6], which are illustrated in the following. Consider a CC scheme where updates are atomic and where queries are able to execute reads "freely", i.e., without synchronizing with any concurrent writer. Let us consider the execution illustrated in Figure 3, which depicts a query whose execution spans the [ts,te] time interval, and which is assumed to be now scanning a specific region at some time t in the interval [t1,t2]. The figure shows concurrent updates affecting objects, white dots represent objects initial positions and black dots represent their updated positions. We can identify four phantoms (concurrency anomalies):

- i1 : Object A is in the query range at ts. However, it exits the range before being seen by the query and therefore is not reported. With *timeslice* semantics, A would be reported as it could not be updated after the query started. Note that B also exits the range during [t1, t2], but is captured in both CC schemes.
- i2 : Object C is not within the query range at ts. However, it is reported because it enters the range during [t1, t2]. With *timeslice* semantics, C would not be reported. Note that D also enters the range during [t1, t2], but is not reported in either CC scheme.
- i3 : Some of the reported object positions are fresher than others. For example, objects E and F are both in the query range before and after being updated. However, only F's updated position is reported. With *timeslice* semantics both objects initial positions would be reported.
- i4 : Both of object G's positions are in the range, but the query fails to capture G because the update moves from the yet unscanned to the already scanned query region. This does not occur with *timeslice* semantics.

Of all of the phantoms, only i4 phantoms must be avoided to ensure *freshness* semantics. i4 phantoms happen when an object, yet to be scanned, updates its position to an already scanned area (object G). Thus, the object is still in the query range but it seems to have disappeared. It has been argued by [16] that this type of anomaly is strongly undesirable for typical applications. The resolution of this problem is made in the description of PGrid, in Section 3.2.2.

With *freshness* semantics, a query processed from ts to te returns all objects that have their last reported positions before ts in the query range, and it

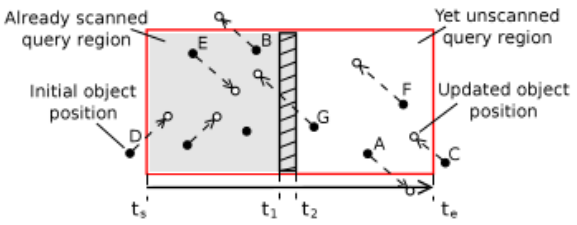


Figure 3: Parallel updating and querying [16]

reports some (fresher) objects that have their last reported positions after ts (and before te) in the query range.

Indexes Implementation

In this section we review several existing indexes for spatio-temporal data. The considered algorithms all target in-memory data structures, but differ across multiple dimensions. While presenting the various algorithms, we shall focus in particular on u-Grid and PGrid, as, in Section 4, we will use them as starting point to derive HTM-enhanced versions. The remaining algorithms will instead be overviewed at a higher level of abstraction.

All indexes presented in the following use two different structures. The primary structure is the grid, Figure 4. The primary index is used to allow efficient retrieval of objects based on their spatial position during query operations.

A secondary index structure is also used, an hashtable, which stores objects using their id's as key. This index is used by update operations, to efficiently locate the corresponding entry to be updated in the primary index. This technique, which aims to accelerate access to objects in the primary index, is called bottom-up fashion update.

All the indexes reviewed in the following support the same set of operations, namely the Update (Object_id, old_x, old_y, x, y), and the RangeQuery (Rectangle q).

u-Grid

u-Grid is a single-threaded index structure that offers high performance for traffic monitoring applications in single-threaded settings [15]. Queries are serviced using a uniform grid [1], while updates are facilitated via a secondary index in bottom-up fashion. A uniform grid (Figure 4) is a space-partitioning index where a defined area is divided into cells (grid), whose resolution can be statically set based on the expected data density. However, no grid refinement or re-balancing is made during the execution of the system. Objects within a particular cell grid belong to that cell. Grid cells are stored as a two-dimensional array. Each grid cell within the array stores a pointer to the linked list of buckets that contain the object data. Objects are incrementally stored in buckets following no specific order. Thus, the grid is defined by three parameters: grid_area, grid_cell_size (gcs), and bucket_size (bs).

PGrid

PGrid (Fig. 4) is a multi-threaded version of u-Grid with a fine-grained 2 phase locking [2] scheme. PGrid avoids acquiring locks on the entire set of cells to be accessed by a query before starting scanning such cells. Conversely, queries that need to scan multiple cells acquire at most a cell lock at a time, and only to enlist the reader into the cell, i.e. for a very short duration. This allows updates to manipulate the position of objects in the cell concurrently with queries. In order to ensure freshness semantics, PGrid uses two additional ideas/mechanism:

1. whenever an update alters the position of an object it always preserves the object's previous position. This allows queries to detect and fix situations where the i4 phantom may arise (which would cause objects to be missed due to concurrent updates moving the object from a cell yet to be scanned to an already scanned cell, see Fig 3) by letting queries return the position of objects not reflecting the updates issued by concurrent updates.
2. in order to ensure that concurrent execution of updates does not jeopardize the correctness of queries, the atomicity of each object read is carried out using lock-free mechanisms (possibly exploiting specialized hardware support).

With the execution of multiple concurrent updates, both the primary (Grid directory) and secondary index (hashtable) are going to be manipulated concurrently. The changes made in one have to be reflected in the other. To guarantee consistency between the two, PGrid's Concurrency Control (CC) scheme includes two types of locks: object locks and cell locks.

The main purpose of object locks is to provide synchronized, single-object updates between the two structures. After an object lock is acquired, the updater is sure that the object-related data is not changed in either index by concurrent updates. Since an object lock blocks write accesses just to one particular object, it has only a modest effect on the potential parallelism. As mentioned before, concurrent updates to the same object are rare if ever encountered.

The main purpose of a cell lock is to prevent concurrent cell modifications, i.e., physical deletion/insertion of new objects in a cell and, consequently, deletion/insertion of new buckets in the cell. For example, when a bucket becomes full, the cell lock guarantees that only one thread at a time allocates a new bucket and modifies the pointers so that the new bucket becomes the first (bucket pointed by cell).

Accelerated Spatio-Temporal Data Indexes

In this chapter we present several HTM-based spatio-temporal indexes. The starting point to derived such algorithms are two state of the art solutions, namely u-Grid and PGrid (see Section 3). For u-Grid we consider implementations based both on plain HTM interface, as well as on HRWLE (see Section 2). For PGrid, we consider how to integrate HTM in the OLFIT-based version, and consider the possibility of ensuring atomicity of the ob-

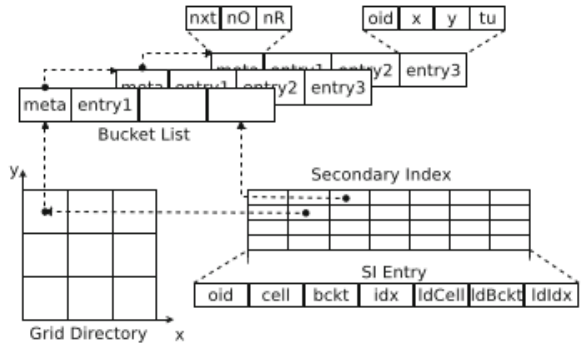


Figure 4: PGrid Structure [16]

Name	Base Alg.	HTM Inter.	# U Txs	# Q Txs
u-GridHTM	u-Grid	HTM	1	1
PGridOHTM	PGridOLFIT	HTM	1	Partitioned
u-GridHRWLE	u-Grid	HRWLE	1	0

Table 1: Spatio-temporal indexes HTM solutions

jects scanned during the query operations via HTM transactions spanning a tunable number of objects. Table 1 depicts all of our solutions distinguished by base algorithms, interfaces used and number of transactions used in their main operations. These solutions will be experimentally evaluated in Chapter 5.

u-GridHTM and u-GridHRWLE

Our first approach was to straightforwardly apply HTM to a state of the art single-threaded index, u-Grid [15], by wrapping all of its operations with HTM transactions. The modifications made to the original u-Grid's algorithms are marked with a yellow shading. Note that these algorithms use HTM with a SGL fall-back path.

However, wrapping the two main operations in a single transaction, respectively, may not be ideal in terms of performance. If on the one hand, update transactions are small and unlike to conflict with other update operations. On the other hand, queries access a much larger number of memory locations. Wrapping them within a single hardware transaction makes them prone both to suffer from capacity exceptions and to contention with concurrent update transactions.

A similar approach was used to derive u-Grid with Hardware HRWLE (u-GridHRWLE). In this case, query transactions are mapped to a read-only transaction/critical section, whereas updates are mapped to a write critical section. HRWLE is expected to make HTM capacity issues negligible, by allowing queries to run uninstrumented and, hence, without any capacity limitation. Nevertheless, contention between updates and queries still exists. Further, HRWLE induces extra costs to update transactions, by forcing them to undergo a quiescence phase to wait for any active reader. As such, HRWLE is expected to be better fitted for query (read) intensive workload scenarios.

In terms of consistency, u-GridHTM and u-GridHRWLE deliver Degree 3 consistency level, which provides serializability and *timeslice* query semantics.

PGridHTM

PGridHTM uses a complex and carefully optimized algorithm, based on the joint use of fine-grained locking, atomic operations and non-blocking synchronization techniques (e.g. OLFIT). Further, in order to boost performance, PGridHTM adopts a non-serializable consistency criterion, namely *freshness* semantics (see section 3.1). In the following we depict the main changes brought by PGridHTM:

1. HTM is used to elide objects and cells locks, Which provides a higher degree of parallelism. Specifically, in object insert and delete operations, and in the query subscription to a cell. These operations were previously protected by a cell lock, which forced the serialization of these operations.

2. Since PGrid relies on a fine-grained locking scheme, in the moment in which HTM is used to elide its locks, one is left with the decision on how to regulate the fall-back path of HTM. The simplest method is to use a single global lock (SGL), however, in doing so we are restricting the original parallelism offered in PGrid. Hence in order to provide a fall-back path, which ensured the same parallelism as PGrid, we used the fine-grained locking mechanism of PGrid as our fall-back path.

3. We maintain *freshness* semantics by reusing the already present atomic instructions (OLFIT), which allow atomicity between updates and queries, whether running transactionally or in the fall-back path. In order to allow concurrent updates and queries, PGrid resorts to atomic operations (e.g. Optimistic Lock-Free Index Traversal (OLFIT)) to ensure queries do not read corrupted object data, concurrently modified by updates. In order to allow the same mechanism with lock elision, the fall-back path uses these operations to ensure atomic operations. Conversely, transactions provide atomicity, hence, they do not require any instrumentation. In fact, PGridHTM allows concurrency between transactions and OLFIT.

4. We provide the possibility of performing query scans in multiple transactions, which we call

transactional partitioned queries. In PGrid, these were made using atomic operations (e.g OLFIT) without resorting to locks. Nevertheless, we want to evaluate if issuing transactions in query scans (instead of using OLFIT), can improve performance.

We implement PGridHTM with two different fall-back paths. The first version, uses the SGL as its fall-back path, whereas the second version, uses the FGL system of PGrid as its fall-back path. They are respectively named PGridHTM-SGL and PGridHTM-FGL.

Evaluation

Our evaluation is split among 3 different platforms, which were used to make an extensive study of HTMs behaviour in different architectures. Each platform is described as follows:

1. **Broadwell**: Intel(R) Xeon(R) CPU E5-2648L v4 @ 1.80GHz processor with 2 sockets, connected with UMA, 14 physical cores per socket, where each core can execute 2 hardware threads (hyper-threading [13]). The operative system (OS) is Ubuntu 16.04.4 and uses GCC version 5.4.0.
2. **Haswell**: Intel(R) Xeon(R) CPU E3-1275 v3 @ 3.50GHz processor with 1 socket, 4 physical cores, where each core can execute 2 hardware threads (hyper-threading [13]). The OS is Ubuntu 12.04.2 LTS and uses GCC version 4.8.1.
3. **POWER8**: 80-way IBM POWER8 8247-21L @ 3.42 GHz processor with 2 sockets, connected with NUMA, 5 physical cores per socket, where each core can execute 8 hardware threads. The OS is Fedora 24 with Linux 4.7.4 and uses GCC version 6.2.1.

Finally, we consider the following algorithms in our evaluation:

1. Single-threaded algorithms which we only report performance achieved in single-threaded configurations, namely u-Grid and u-R-tree.
2. Multi-threaded algorithms with a fine-grained locking concurrency scheme, namely Serial and PGrid.
3. Multi-threaded algorithms with an HTM-based concurrency scheme, namely u-GridHTM, u-GridHRWLE and PGridHTM.

Haswell

In query intensive workloads (top plots of Fig. 5), Serial is clearly the best performing index. Mainly due to the relatively low frequency of update operations, which contribute to the rare existence of hotspots. Moreover, recall that Serial relies on read locks to protect queries' execution, while ensuring that these can be processed concurrently.

The u-R-tree has a competitive performance even though it is a single-threaded index. This occurs due to the efficient queries performed in it. In contrast the u-Grid is better suited for update intensive workloads, hence, it has a considerably lower performance than the u-R-tree. Finally, u-GridHTMs performance is very poor, as previously explained, capacity and conflict aborts are abundant due to

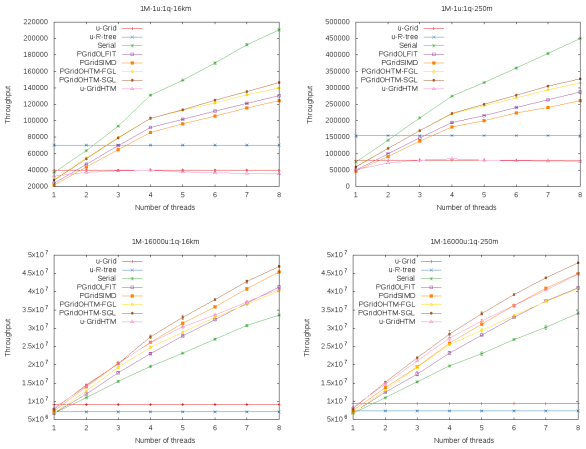


Figure 5: Haswell - All indexes performance evaluation

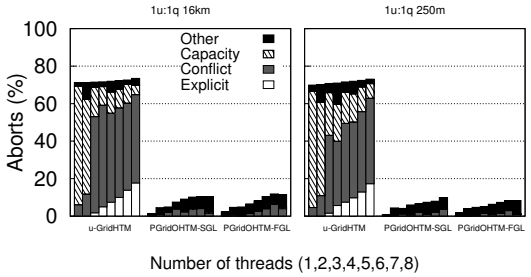


Figure 6: Haswell - Abort Breakdown of HTM indexes

the need of wrapping the entire update and query functions in single transactions.

Update intensive workloads originate completely different results (bottom plots of Fig. 6). Serial's performance drastically diminishes due to the hotspots generated by updates. Hence, it becomes the worst performing multi-threaded index. In contrast, u-GridHTM performance drastically improves, becoming competitive with PGrid and PGridHTM indexes. It is interesting to observe that in the 16000u:1q-250m workload u-GridHTM, which is obtained by straightforwardly applying HTM to a single threaded index, achieves performance on par (or even better) than the lock-based PGrid variants, which rely on complex and highly optimized synchronization strategies.

The best performing index is PGridOHTM-SGL, achieving 25% speedups in comparison with PGrid-SIMD. Again, PGridOHTM-FGL extra instrumentation hiders performance since the fall-back path is rarely acquired. u-Grid's throughput is able to surpass the u-R-tree's, due to the update efficient grid structure.

The obtained results answer the fundamental questions this thesis proposes. First, we are able to achieve on par (or even better) with a sin-

gle threaded index made concurrent with HTM, in comparison with a state-of-the art lock-based multi-threaded indexes, specifically designed to support multi-threading. Second, we achieve better performance using HTM against lock-based multi-threaded indexes. Using HTM in ways that improves parallelism and performance of these indexes. Finally, there seems to be potential to improve by using HTM, but the degree of parallelism is too small to emphasize the gains, in Haswell. Therefore, we will be answering these question when considering Broadwell and P8 architectures, which do support a higher degree of parallelism.

Broadwell

Figure 7 depicts the performance of all indexes in the Broadwell architecture. In query intensive workloads, Serial is again the index with the best performance. We attribute this performance due to the low frequency of hotspots generated in these workload scenarios, and due to the lightweight instrumentation imposed to queries in Serial. Nevertheless, PGridHTM is able to outperform PGrid. In the 16km query range scenario, PGridHTM-FGL is the index with the higher throughput during the entire thread spectrum. At maximum thread count, the two highest throughput indexes are PGridHTM-FGL and PGridSIMD. In contrast, in the 250m query range scenario PGridHTMs (SGL/FGL) achieves 5% speedups over PGrid (OLFIT/SIMD). Figure 8 depicts the abort breakdown, where we can see that the FGL version has less aborts than the SGL version. Finally, u-GridHTM throughput is lower than the single threaded indexes, due the earlier explained contention and capacity overflow. The u-R-tree is again able to have better performance over u-Grid in query intensive workloads. In update intensive workload scenarios the best solutions are both

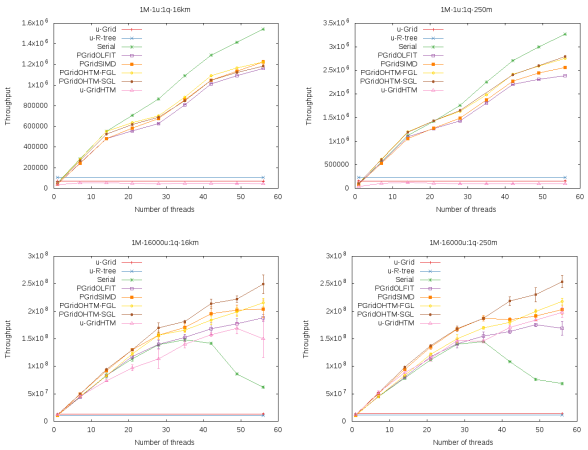


Figure 7: Broadwell - All indexes performance evaluation

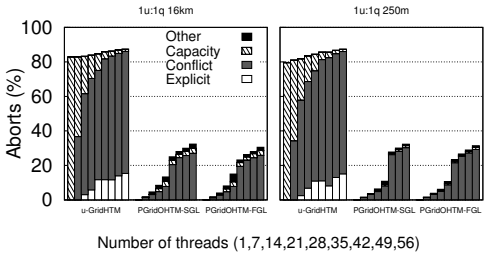


Figure 8: Braodwell - Abort Breakdown

PGridHTM's versions. However, the SGL version achieves a higher throughput over the FGL version. As previously explained, the extra instrumentation of FGL, plus, the lack of lock acquisition in update intensive workloads, hinder its performance. Most importantly, PGridOHTM is able to achieve 25% speedups over PGridSIMD and 40% over PGridOLFIT.

Serial's performance is again drastically affected due to the contention generated by updates. In contrast, u-GridHTM drastically improves due to the much lower frequency of queries, which generate contention and capacity issues. In the 250m scenario, it is even capable of out-performing PGridHTM (OLFIT version), due to its lack of instrumentation. Finally, u-Grid is able to outperform u-R-tree, due to the more conflict prone update processing of the u-R-tree.

u-GridHTM's results confirm the first question proposed in this thesis, thanks to HTM, it is possible to achieve performance competitive to complex lock-based algorithms at a fraction of the complexity. This is only true, though, if the workload characteristics fit the architectural restrictions (e.g., transactional capacity). The architectural restrictions imposed to current HTM implementations restrict their usage. In this sense, these results suggest that ad-hoc locking strategies, despite more complex and error prone, represent currently a more robust and general solution that HTM.

As for the second question, PGridHTM is able to achieve 25% speed-ups over PGrid. The fact that HTM can further improve over complex locking scheme, in realistic workloads, confirms the potential and relevance of HTM.

POWER8

In query intensive workloads (top plots of Fig.9), Serial is the best performing index. As previously explained, due to the low contention between updates, a few amount of hotspots is generated, which are the source of bad performance in Serial.

PGridHTM-FGL is able to achieve performance equal or better than PGridOLFIT. Being able to achieve 25% speedups in the 1u:1q-16km workload scenario. As we can see in Figure 10, both ver-

sions of PGridHTM (SGL and FGL) have a negligible abort count, this relates to the OLFIT mechanism used to perform query scans. We find this a great discovery for architectures with small transactional capacities, which have issues with the amount of memory used in query scans. This occurs by exploiting concurrency between non-bockable synchronization techniques (i.e., OLFIT) and atomic operations (transactions). Interestingly, the FGL version is able to achieve a higher throughput over the SGL version, hence, the overhead of using multiple locks (FGL) is favourable to the overhead of blocking all transactions when an abort occurs (SGL).

Next, u-GridHTM performance is still worse than the single-threaded indexes. Due to contention and capacity issues, it is even more explicit in POWER8s environment. In contrast, u-GridHRWLE is able to begin with a competitive performance until the 16 thread mark, since it does not exceed capacity with its non-speculative queries. Nevertheless, after the 16 thread mark contention starts to rise, conflict aborts become excessive and throughput drops below the single-threaded indexes throughput. Finally, as previously mentioned, the u-R-tree has a faster query scanning algorithm, which allows it to have a superior throughput over u-GridHTM.

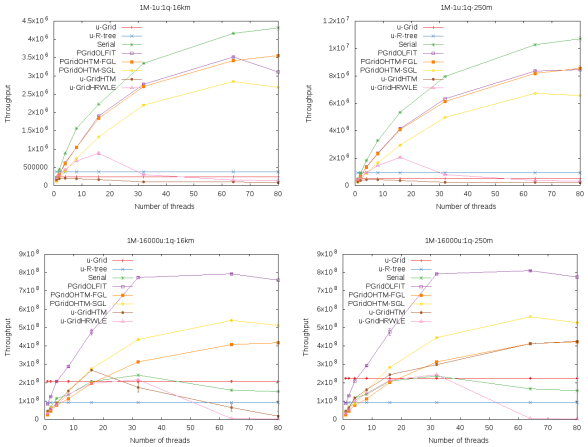


Figure 9: POWER8 - All indexes performance evaluation

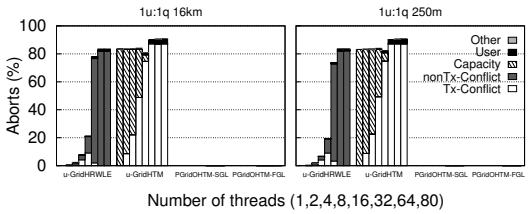


Figure 10: POWER8 - Abort Breakdown

Update intensive workload scenarios force a completely different behaviour on indexes. In these scenarios, PGrid is the best performing index. In contrast, Serial's performance is drastically affected, mostly due to the contention between updates. Serial only allows serializable updates in the same cell, which does not favour update intensive scenarios. u-Grid is able to outperform the u-R-tree in this scenario. Mostly, due to the already discussed inefficiency of u-R-tree in handling update operations.

PGridHTM is only able to achieve the second best throughput. Moreover, the SGL version is capable to achieve a higher throughput due to its lightweight instrumentation, which is especially noticeable in high throughput workloads. u-GridHTM has a bad performance on the 16km range query workload, which forces capacity and conflict aborts. Conversely, in the 250m scenario it has competitive performance with PGridHTM. u-GridHRWLE has competitive throughput until the 32 thread mark, however, due to the few queries present in theses scenarios, contention between updates and queries still occurs, and performance drops lower than the single-threaded indexes.

Conclusions

The relevance of spatio-temporal data applications and the volume and velocity of such type of data has dramatically increased, over the last few years, thanks to the proliferation of GPS equipped devices. The problems of developing indexes for spatio-temporal queries is well-known and several have been proposed in literature [16, 10]. In this thesis, we study efficient ways to enable concurrent access to spatio-temporal data indexes, in order to take advantage of modern multi and many core architectures.

TM has emerged as a promising abstraction for parallel programming. Specifically, we use HTM as a synchronization alternative to conventional locking for main-memory spatio-temporal indexing data structures and seek an answer to the following research questions: i) what efficiency levels can be achieved by applying HTM to state of the art *single-threaded* (i.e., non-thread safe) spatio-temporal indexes algorithms? In particular, how does the performance of such HTM-based algorithms compare with state-of-the-art *concurrent* algorithms, designed from scratch to cope with the consistency issues arising in multi-threaded environments? ii) to what extent can HTM be applied to state-of-the-art concurrent indexing algorithms for spatio-temporal data, in order to enhance their efficiency?

To answer the first question we apply HTM to u-Grid, a state of the art single threaded index that is well known for its high efficiency in dealing with update operations. Our results show that

u-GridHTM is able to achieve performance comparable to state of the art concurrent algorithms that use complex and carefully engineered fine-based locking schemes.

To answer the second question we study how HTM can be used to execute in a speculative fashion the critical sections used in the PGrid concurrent algorithm. Our results demonstrate that PGridHTM is able to achieve 25% speedups over PGrid in query intensive scenarios, and up to 40% speedups over its rival, PGridOLFIT, in update intensive workload scenarios.

Evaluation was performed considering 3 different parallel machines, equipped with processors that adopt different architectures and HTM implementations. In particular the Intel Haswell and Broadwell and IBM POWER8 CPUs. Moreover, we considered a data set of 4 different realistic workloads, generated using Moving Objects Trace generatOr (MOTO).

Acknowledgements

I would like to thank my supervisor, Prof. Paolo Romano, for all the support in the realization of this dissertation. For all the hours spent discussing possible solutions and interpreting results, for the patience and the time that allowed me to write this dissertation.

References

[1] V. Akman, W. R. Franklin, M. Kankanhalli, and C. Narayanaswami. Geometric computing and uniform grid technique. *Computer-Aided Design*, 21(7):410–420, 1989.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *RRENCY CONTROL AND RECOVERY IN DATABASE SYSTEMS*. Addison- Wesley, 1987.

[3] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *ACM sigplan notices*, volume 44, pages 155–165. ACM, 2009.

[4] P. Felber, S. Issa, A. Matveev, and P. Romano. Hardware read-write lock elision. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 34. ACM, 2016.

[5] S. Ghemawat and P. Menage. Tcmalloc: Thread-caching malloc, 2009.

[6] J. Gray and A. Reuter. *Transaction processing: concepts and techniques*. Elsevier, 1992.

[7] P. Guide. Intel® 64 and ia-32 architectures software developer's manual, 2011.

[8] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60. ACM, 2005.

[9] H. Q. Le, G. Guthrie, D. Williams, M. M. Michael, B. Frey, W. J. Starke, C. May, R. Odaira, and T. Nakaike. Transactional memory support in the ibm power8 processor. *IBM Journal of Research and Development*, 59(1):8–1, 2015.

[10] S. Li, S. Hu, R. Ganti, M. Srivatsa, and T. Abdelzaher. Pyro: a spatial-temporal big-data storage system. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 97–109, 2015.

[11] S. Loosemore, R. M. Stallman, A. Oram, and R. McGrath. The gnu c library reference manual. 1993.

[12] D. Makreshanski, J. Levandoski, and R. Stutsman. To lock, swap, or elide: on the interplay of hardware transactional memory and lock-free indexing. *Proceedings of the VLDB Endowment*, 8(11):1298–1309, 2015.

[13] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-threading technology architecture and microarchitecture. In *Intel Technology Journal Q1*. Intel Corporation, 2002.

[14] T. Nakaike, R. Odaira, M. Gaudet, M. M. Michael, and H. Tomari. Quantitative comparison of hardware transactional memory for blue gene/q, zenterprise ec12, intel core, and power8. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 144–157. ACM, 2015.

[15] D. Šidlauskas, S. Šaltenis, C. W. Christiansen, J. M. Johansen, and D. Šaulys. Trees or grids?: indexing moving objects in main memory. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 236–245. ACM, 2009.

[16] D. Šidlauskas, S. Šaltenis, and C. S. Jensen. Processing of extreme moving-object update and query workloads in main memory. *The VLDB Journal*, 23(5):817–841, 2014.

[17] I. P. I. Version. 2.07, 2013.

[18] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of intel® transactional synchronization extensions for high-performance computing. In *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*, pages 1–11. IEEE, 2013.