TÉCNICO LISBOA

Exploiting Hardware Transactional Memory to Accelerate Concurrent Spatio-Temporal Indexes

Nuno Henrique Nina Ribeiro Elvas Fangueiro

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor(s): Prof. Paolo Romano

Examination Committee

Chairperson: Prof. Alberto Manuel Rodrigues da Silva
Supervisor: Prof. Paolo Romano
Member of the Committee: Prof. João Pedro Barreto

November 2017

To my parents, thank you for everything.

Acknowledgments

First and foremost 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.

Another person who helped a lot in doing this thesis was Pedro Raminhas. Without Pedro, the dissertation would not have gone as smoothly, thank you very much. Thank you for all the time you spent explaining and debugging the code with me and for all the feedback throughout this dissertation.

To my parents, there is not enough words that can describe how thankful I am. Thank you for supporting me in everything in life and helping me achieve my goals and dreams.

Also, I want to thank all my friends that supported me through this challenge and university. I want to give a special thank you to André Santos, who has been with me through all of my academic experiences, whom I shared amazing experiences and projects throughout this six years. Another special thank you to Bernardo Rodrigues, Pedro Rosa and Miguel Costa, that stayed up with me all night in order to finish the dissertation.

Abstract

The recent proliferation of devices that are capable of sending information about their location over time (e.g., GPS-equipped smartphones), has turned big spatio-temporal data processing into a mainstream, highly relevant for a broad class of applications. Recent literature in the area of big data has focused on how to exploit recent hardware trends/mechanisms to accelerate big data processing.

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 Indexing Structures, Concurrency Control schemes, Transactional Memory, Performance.

Resumo

A recente proliferação de dispositivos capazes de enviar informação acerca da sua localização ao longo do tempo (e.g., smartphones equipados com GPS), tornou o processamento de grandes conjuntos de dados espaço-temporais numa convenção de alta relevância para uma vasta classe de aplicações. Literatura recente na área de grande data tem-se focado em como explorar recentes tendências/mecanismos de hardware para acelerar o processamento de grande conjuntos de dados.

Esta tese foca-se em como explorar Memoria Transacional (MT) para acelerar aplicações que têm como alvo grandes conjuntos de dados espaçotemporais. A MT emergiu como uma abstracção promissora para programação paralela, que tem como alvo melhorar a performance e simplificar a programação de aplicações concorrentes. Especificamente, nós usamos a Memória Transacional em Hardware (MTH) como alternativa de sincronização ao convencional bloqueio, e procuramos responder às seguintes questões de pesquisa: i) que níveis de eficiência podem ser atingidos ao aplicar MTH em índices espaçotemporais *single-threaded* (i.e não permitem concorrência). Em particular, como é que a performance de tais algoritmos baseados em MTH se compara com algoritmos concorrentes do estado da arte, desenhados desde o principio para suportar com os problemas de consistência surgindo em ambientes *multi-threaded*? ii) até onde pode a MTH ser aplicada em algoritmos de índice do estado da arte para data espaço-temporal, de modo a melhorar a sua eficiência?

Palavras-chave: Dados Espaço-temporais, Estruturas de Índices Espaço-Temporais, Esquemas de Controlo de Concurrência, Memória Transaccional, Performance.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the past decade the amount of data generated in a global scale has increased exponentially and it appears that this trend is not going to change in the near foreseeable future. “The world contains an unimaginably vast amount of digital information which is getting ever vaster ever more rapidly” [1]. The “big data” term is used to describe these large amounts of data. Big data applications range a broad number of domains, including disease prevention, crime combat and prediction of business trends.

This dissertation is focused on a specific sub-domain of big-data, namely the development of concurrent indexes for big spatio-temporal data applications. The relevance of spatio-temporal data applications, and the volume and velocity of such a type of data has dramatically increased over the last few years thanks to the proliferation of GPS equipped devices (e.g., smartphones). This has enabled a number of challenging data-intensive applications in the spatio-temporal domain, including traffic control [2] and geo-localized social networks [3].

The problem of developing indexes for spatio-temporal queries is well-known and several approaches have been proposed in literature. Existing solutions can be coarsely grouped into three classes: i) indexes that use specialized data-structures designed for the multi-dimensional nature of spatio-temporal data, such as Quad-Trees, R-Trees, K-Trees, Grids [4, 5, 6, 7], ii) solutions that first linearize spatio-temporal data into a one-dimensional space and then apply generic/non-specialized indexing solutions, such as the B+-tree [8], and iii) solutions which combine both techniques used in previous classes, using linearization techniques to speed up the mapping of spatio-temporal information to multi-dimensional data structures [9, 10].

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 [11, 12], and to lack one important property that is fundamental in modern software engineering approaches: composability [13].

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 [14] Haswell and Broadwell, and IBM POWER8 [15] 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 objects can move through and comply with 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 (TC-Malloc [16]). We come to the conclusion the standard GNU C library (Glibc) [17] memory allocator is best when coupled with non-HTM indexes. Moreover, it does not falter (see Chapter 4) as TCMalloc with PGridHTM [5], hence, we conclude that in general the Glibc memory allocator is the most well suited memory allocator to handle our spatio-temporal indexes. The optimal configuration for the transactional retry logic strongly depends on the target platform. Nevertheless, our results show that a 20 retry value is able to ensure high 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) [4] 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) [4], which we will not be applying HTM to, due to implementation constraints. 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) [5] and Parallel Grid (PGrid). The results of this study shows 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, 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. Chapter 2 provides a background on two main areas: TM and spatio-temporal index structures. Chapter 3 describes our HTM solutions for spatio-temporal indexes. Highlighting the research directions we intend to study in this thesis. Chapter 4 presents the extensive experimental study made to our solutions. Finally, Chapter 5 concludes the dissertation by summarizing the work done this thesis and discusses possible future research.

Chapter 2

Background

2.1 Transactional Memory

Transactional Memory (TM) is a concurrency mechanism that many researchers believe to be the path to follow to untap the parallelism of modern multi-core systems, while drastically simplifying parallel programming. TM has several design characteristics that achieve greater parallelism than locks, and for that reason, it can be better suited for new multi-threaded processors.

TM is a mechanism that offers the possibility for programmers to define the transactional area of code. TM will then take under account all concurrency issues on that given area of code and provide parallel transactions.

Transactions have been successfully implemented in database systems since the early 90's [18]. Transactions provide atomicity, a transaction either fully completes its read/write operations and commits or it has no effect (aborts). Transactions buffer their writes and store their reads in memory in order to detect conflicts, and abort the transactions that compromise some target consistency criterion, typically opacity [19]. Opacity ensures that a committed transaction can be serialized instantaneously in the moment in which is committed.

In the TM programming environment, programmers simply define as transactions the program regions that access shared variables. TM runtime systems achieve high concurrent performance by optimistically executing the transactions in parallel [20].

Conflicts happen when two transactions address the same memory location. Conflict detection may be at transaction start time (eager), or at the end of the transaction (lazy).

The most important advantages that transactions bring in comparison with locks are their simplicity of implementation and their optimistic conflict detection.

There are various possible implementations of TM, in Hardware (HTM), in Software (STM), or combinations thereof, also called Hybrid TM (HyTM). In the following, each of these variants will be overviewed.

2.1.1 Hardware Transactional Memory

HTM is a concurrency protocol that 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 [21]). 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.

Recently, IBM and Intel have introduced HTM implementations respectively for High-Performance Computing (HPC) and commodity processors. "This represents a significant milestone for TM, mainly due to the predictable widespread availability of Intel Haswell processors, which bring HTM support to millions of systems ranging from high-end servers to common laptops" [22].

In the rest of this section we overview four commercial processors which support HTM, comparing their architectural structure. Following, we specifically review Intel Core and POWER8, which are the processors we used to evaluate our solutions. Finally, we make a depiction of the basic fall-back paths, focusing on how they influence HTMs performance.

2.1.1.1 Overview of existing HTM implementations

Paper [20] investigates in depth the HTM implementations existent in four commercial processors from Blue Gene/Q [23], zEnterprise EX12 [24], Intel Core [14] and POWER8 [15]. From this research, various design choices of each processor are revealed. In Table 2.1 we can observe such design choices. We

Processor type	Blue Gene/Q	zEC12	Intel Core i7-4770	POWER8
Conflict-detection guranularity	8 - 128 bytes	256 bytes	64 bytes	128 bytes
Transactional-load capacity	20 MB (1.25 MB per core)	1 MB	4 MB	8 KB
Transactional-store capacity	20 MB (1.25 MB per core)	8 KB	22 KB	8 KB
L1 data cache	16 KB, 8-way	96 KB, 6-way	32 KB, 8-way	64 KB
L2 data cache	32 MB, 16-way, (shared by 16 cores)	1 MB, 8-way	256 KB	512 KB, 8-way
SMT level	4	None	2	8
Kinds of abort reasons	-	14	6	11

Table 2.1: HTM implementations of Blue Gene/Q, zEC12, Intel Core i7-4770, and POWER8 [20]

compare two specific design choices that have the most impact in the HTM environment.

The first choice is *conflict-detection granularity*. The *conflict-detection granularity* in zEC12, Intel Core, and POWER8 it is their cache-line sizes. When working with HTMs, even if two transactions address different bytes of a cache-line, this results in a conflict. This is called a *false conflict*. As shown in Table 1, zEC12 has the bigger cache granularity, which provides better *data locality*. However, in the case of HTMs, it induces the bigger abort rate due to *false conflicts*.

The second choice is *transactional capacity*. *Transactional capacity* is the maximum amount of memory data that can be accessed by a transaction [20]. This resource is scarce in HTMs since transactions use the cache to store their metadata. This metadata is composed by accessed memory locations, used for conflict detection, and buffered transactional stores, used to store the data required to commit or abort the transaction.

In general, the load capacity is larger than the store capacity because the conflict detection has to record only the accessed memory addresses, while the store buffering needs to keep the stored data. Finally, when a transaction tries to access a cache line that will exceed the capacity, it is aborted. This is called a *capacity-overflow* abort.

2.1.1.2 Intel Core

Intel Core [14] uses the L1 cache for conflict detection and store buffering [21]. Further research from [20], evaluates that the load and store capacities are 4MB and 22KB, respectively, on Core i7-4770. Moreover, they claim that the load capacity is larger due to because it uses other resources to track the cache lines that were evicted from the L1 cache. And that, the transaction capacity for the stores is within the size of the L1 cache. In comparison with POWER8, Intel's transactions have a considerable higher amount of memory to work with.

The latest Haswell processors made by Intel Core come with a new extension of the instruction set architecture (ISA) [21], 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.

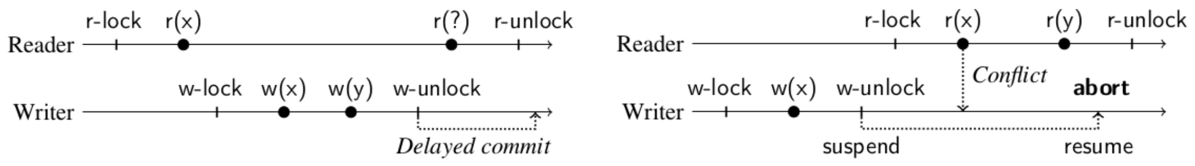
In summary HLE is used for compatibility with legacy processors. In contrast, RTM explicitly enables the programmer to define the transactional critical areas, thus bringing more flexibility. However it requires programmers to define a fall-back path.

2.1.1.3 POWER8

POWER8 [15] uses content addressable memory (CAM) linked with the L2 cache for conflict detection [25]. 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. [26], 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 2.1: 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. 2.1 (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. 2.1 (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 [26]. 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.

2.1.1.4 Single Global Lock Fall-back

As noted, HTM has become a commercial reality, however, Intel Core and POWER8 offer no progress guarantees (best-effort HTMs), implying that some form of software fall-back is needed. In the single global lock (SGL) approach, each shared data structure has an associated lock. When it starts, a hardware transaction immediately reads the lock state, an action known as eager subscription. When a repeatedly failed hardware transaction restarts in software, it acquires exclusive access to the lock, forcing any subscribed hardware transactions to abort. SGL fall-back is attractive because it is simple, requiring no memory access annotation, and no code duplication between alternative paths. Nevertheless, an inherent limitation of current SGL fall-backs schemes is that hardware and software transactions that share a global lock cannot execute concurrently. Figure 2.2 shows the four ways in which hardware and software transaction can overlap. In cases 2 and 3, the hardware transaction is aborted as soon as it checks the lock, while in cases 1 and 4 the hardware transaction is aborted when the software transaction acquires the lock. With eager subscription, it makes sense for a thread starting a hardware transaction to wait until the SGL becomes free.

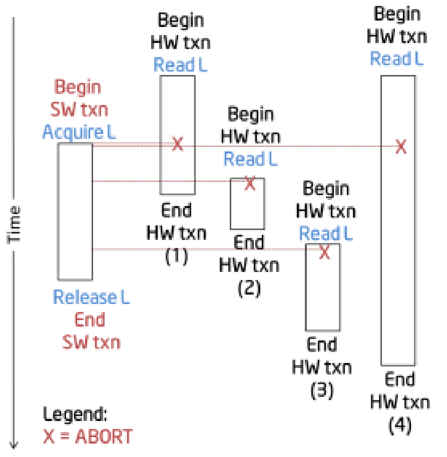


Figure 2.2: SGL fallback path (eager subscription) [27]

In a eager SGL implementation (E-SGL), a hardware transaction immediately adds the lock to its read set, ensuring the transaction will be aborted if that lock is acquired by a software transaction. Hardware and software transactions cannot overlap (Figure 2.2). Lazy subscription can improve the chances of

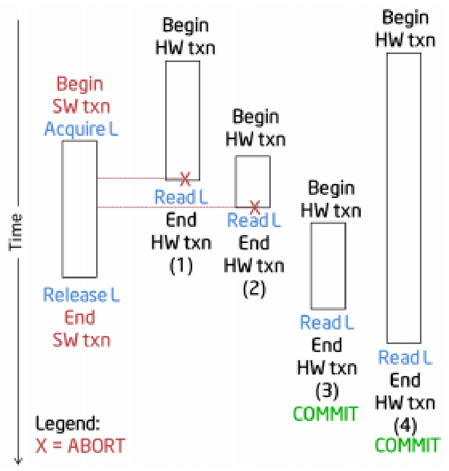


Figure 2.3: SGL fallback path (lazy subscription) [27]

success of a hardware transaction by allowing some overlap with a software transaction. In Figure 2.3, L-SGL allows transactions (3) and (4) to commit, while E-SGL would abort them.

Software and hardware transactions are treated differently in L-SGL. Each software transaction must acquire the SGL. Hardware transactions do not acquire the SGL, but they must check its status. With some exceptions, L-SGL hardware transactions read the lock only at the end, right before committing. If the lock is held by a software transaction, the hardware transaction explicitly aborts. This check is necessary because the hardware transaction may have observed an inconsistent state. If the lock is free, then no software transaction is in progress, and the hardware transaction can commit.

However, a recent paper ([28]) shows that lazy subscription is not safe for transactional lock elision (TLE) because unmodified critical sections executing before subscribing to the lock may behave incorrectly in a number of subtle ways. They also show that recently proposed compiler support for modifying transaction code to ensure subscription occurs before any incorrect behaviour could manifest, is not sufficient to avoid all of the pitfalls they identify. They further argue that extending such compiler support to avoid all pitfalls would add substantial complexity and would usually limit the extent to which subscription can be deferred, undermining the effectiveness of the optimization. Concluding, in order to avoid the pitfalls (inconsistencies) identified in [28], we avoid using the L-SGL fall-back path, and instead, use the traditional E-SGL.

2.1.2 Software Transactional Memory

STM has been focus of intense research along the past decade [22, 12]. Software is responsible to store the metadata (read/write sets) of transactions and detect conflicts. Since STMs are not cache dependent, as HTMs, they are better equipped to handle big transactions. However, STMs have bigger overheads over HTMs, due to the need of software instrumentation.

Throughout the years a number of STM designs have been explored:

1. lock-based vs lock-free - whether an STM uses locks to handle concurrency.

2. encounter-time locking vs commit-time locking - when conflict detection is performed either at commit time (lazy) or encounter time (eager).

3. word-based vs object based - granularity at which memory is accessed, usually used to validate transactions data-sets.

4. Contention Management Scheme (CMS) - Contention manager module is responsible for ensuring that the system as a whole makes progress [29].

5. visible readers vs invisible readers - whether readers actions are visible to the rest of the system.

6. weakly atomic vs strongly atomic - whether non-transactional code may access the same data as of transactions.

In the rest of this section will be presented two STMs, Swiss-TM [12] and NOrec [30]. They have different designs and are optimized for different workloads, Swiss-TM design choices focus on complex workloads and achieving good performance with large scale benchmarks, which require big transactions. NOrec however, is a much simpler approach to STM, and it is focused on applications with few concurrent writers and many concurrent readers, thus improving the performance of read only transactions.

2.1.2.1 Swiss-TM

Swiss-TM [12] focuses on large applications as games or business applications in order to use the power of multi-core processors to the full extense. These are the design choices of Swiss-TM; “Swiss-TM is lock- and word-based and uses (1) optimistic (commit-time) conflict detection for read/write conflicts and pessimistic (encounter-time) conflict detection for write/write conflicts, as well as (2) a new two-phase contention manager that ensures the progress of long transactions while inducing no overhead on short ones” [12]. Since Swiss-TM supports mixed workloads consisting of small and large transactions, as well as complex data structures, it uses these software mechanisms to achieve a greater performance.

Correctness, i.e., opacity [19], in Swiss-TM is achieved using a metadata structure where the current timestamp of each transaction is stored. The timestamps are ordered using a global counter. To validate a transaction, the timestamp of such transaction must never be greater than the timestamps of the resources it read. If such resources have a bigger timestamp, it means that another transaction concurrently changed them. Finally, a locking strategy is used to ensure that changes in memory are atomic.

Swiss-TM uses a mixed conflicting scheme. For read/write conflicts an optimistic (lazy) approach is used in order to allow more parallelism between transactions. Moreover, big transactions may address memory locations being read by other transactions. If a eager/pessimistic approach was used all other transactions would abort, leading to an overall slowing of the system. In contrast, for write/write conflicts it is used a pessimistic approach, in order to prevent transactions that are doomed to abort from running and wasting resources.

The two-phase CMS enables that short read-write or read only transactions have no overhead, while favouring the progress of transactions that have made many updates (big transactions). For short read-write or read only transactions it is used a timid CMS, which aborts transactions at the first encountered conflict. For bigger transactions, it is used the Greedy [29] CMS, which induces more overhead, but ensures completion of big transactions, thus preventing starvation.

2.1.2.2 NOrec: Streamlining STM by Abolishing Ownership Records

In order to better understand NOrec it is best to learn about Transactional Mutex Locks (TML) [31], the system where it "evolved" from. This is a particular simple system, which uses a single global sequence lock to serialize writer transactions. A single global sequence lock ensures that every invisible reader in the system must be prepared to be invalidated (restart), once one of its peers becomes a writer.

The primary advantage of a sequence lock is that readers are invisible, and need not induce the coherency overhead associated with updating the lock. In contrast, the primary disadvantage is that doomed readers may be concurrent with writers, thus readers may read inconsistent data modified by writers. TML's built in memory management system handles these dangerous memory deallocations situations. Summarizing, TML is a very low overhead STM that is highly scalable in read-mostly workloads.

However, there are two main properties that limit TML's scalability, which will be improved and result in NOrec. Firstly, the eager acquisition of the lock by writers, in a single global sequence lock, means that only one writer may be active at a time. Secondly, invisible readers must be extremely conservative and assume that they may be invalidated by any writer, even if the writers accessed data did not conflict with the readers.

To solve the first problem, NOrec uses a lazy conflict-detection and a redo log for concurrent speculative writers. Modified values are locally stored in the redo lock, if the transaction is able to commit then the values are stored in the shared memory. Furthermore, writing transactions do not attempt to acquire the lock until their commit points, which ensures that the lock is held the minimum amount of time possible, thus increasing the likelihood of read-only transactions to commit.

To handle the second problem, NOrec uses Value-Based Validation (VBV) to allow transactions, both readers and writers, to detect if they actually where invalidated by a committing writer rather than making an conservative approach. VBV allows the abolishment of ownership records (thus the name NOrec), which is used in many other STM implementations to associate transactional meta-data with each data location. In many STMs the maintenance of this table is a hard task and induces a big overhead. With VBV, the actual address and value of the memory location are logged. Thus, validation only consist on re-reading the addresses and verifying if there is a point (namely now) where the transactions' reads could have occurred atomically (to check if the value present in the memory address is equal to the one in the log). The global sequence lock provides a "consistent snapshot" where these validations can occur.

Finally, these new mechanisms enable NOrec to achieve a greater scalability, since there can be multiple transactions (either readers or writers) that are able to "survive" through a writers non-conflicting commit.

```
padded unsigned seqlock

HW_POST_BEGIN
  if (seqlock & 1)
    while (true) // await abort

HW_PRE_COMMIT
  seqlock = seqlock + 2
```

(a) Our naive algorithm

```
padded unsigned seqlock
padded unsigned counter

HW_POST_BEGIN
  if (seqlock & 1)
    while (true) // await abort

HW_PRE_COMMIT
  counter = counter + 1
```

(b) 2-Location

```
padded unsigned seqlock
padded unsigned counter[]
thread local unsigned id

HW_POST_BEGIN
  if (seqlock & 1)
    while (true) // await abort

HW_PRE_COMMIT
  counter[id] = counter[id] + 1
```

(c) P-Counter

Figure 2.4: Instrumentation of hardware transactions in hybrid NOrec algorithms [33].

2.1.3 Hybrid Transactional Memory

HyTM [32], is a mixture between both transactional paths, i.e, HTM and STM. Theoretically, in order to take advantage of the strengths of both STMs and HTMs, HyTMs seem to be a viable solution. In order to provide support for best-effort HTMs, the Hybrid solution may be used as the fall-back path of the system (instead of the usual SGL approach). Thus, this is achieved by instrumenting a hardware transaction with accesses to STM metadata, allowing it to detect conflicts with software transactions. This approach enables logical transactions (HTM transactions) to switch over to STM, per-transaction basis. However, the synchronization required to support both TM implementations within the same system can cause major overheads. Thus, recent research [22] reveals that the combination of best-effort HTMs with STMs has a lower performance than the use of the separate transactional paths.

Among the various HyTMs present in literature, hybrid NOrec [33] is probably one of the most popular solutions. Hybrid NOrec is a HyTM which combines the NOrec STM (previously presented) with a best-effort HTM. Hybrid NOrec allows concurrency between both STM and HTM, however hardware transactions must respect the single writer NOrec protocol. Thus, hardware transactions cannot commit while there are running software writeback transactions and must signal their commit, so as to trigger validation by concurrent software transactions.

Hybrid NOrec uses algorithms, as demonstrated in Figure 2.4, to ensure consistency within STM and HTM concurrent transactions. Note that these algorithms are only used for hardware transactions and are not applied for read-only transactions.

The first, most naive algorithm, is simply to read the global sequence lock (used by HTM and STM) whenever a hardware transaction wants to start. If the lock is taken, the hardware transaction spins until the lock is released. Since the lock value is incremented when this happens, and since the read of the lock by the hardware transaction is transactional, the hardware transaction aborts and retries.

The problem with this first approach is that any hardware transaction will conflict with any software transaction, due to the early read of the sequence lock in HW POST BEGIN. The second, and more worrisome problem, is that any hardware transaction will conflict with any other hardware transaction due to the increment of the sequence lock when committing. These conflicts exist for each transaction's full duration, eliminating concurrency.

Concurrency between hardware transactions is managed by the hardware itself, so there should not be any type of instrumentation at software level. To solve this second issue, algorithm 2-Location uses a new variable named counter. Counter is used by hardware transactions to signal software transactions of their commit, making the conflict between other hardware transactions only at commit time, and thus, improving concurrency between hardware transactions. The last algorithm is P-Count, it uses a different counter per thread to avoid even further conflicts with other hardware threads, however both algorithms require software instrumentation to include verification to the counter variable, inducing more overheads.

2.2 Workload Generators for Spatio-Temporal Applications

Spatio-temporal data has the specific characteristic of having multiple-dimensions, specifically, time and space. Thus, in order to create workloads with this type of data, a series of steps is required to achieve a final result. The final result, however, is always a trace. There are two different possible types of traces that benchmarks use. Recorded traces use real data made from analysis of events like traffic, providing a very realistic job stream. In contrast, synthetic traces attempt to capture the behaviour of observed workloads, and have the advantage of isolating specific behaviours that are not clearly expressed in recorded traces.

The first step is to gather the spatial data required for the workload, which is done by Geographical Data Systems (GIS) [34]. These systems are able to capture, store and manage spatial data. GIS data is stored in the popular ERSI Shapefile [35] data format, which can spatially describe vector features: points, vectors and polygons, representing for example, roads, rivers, and lakes.

The second step is to transform shapefiles into node and edge files, and finally, the third step is to take those edge and node files, and generate position and query traces. For such, we use a Moving Objects Trace generatOr (MOTO) which is based on Brinkhoff [36]. Thomas Brinkhoff's Network-based Generator of Moving Objects [36] has a special feature that usual spatio-temporal benchmarks do not consider. Moving objects, like traffic, usually have a specific path they must follow, where they interact with each other, while obliging to the paths' rules. With this in consideration Brinkhoff's generator uses a network to simulate the paths where moving objects can move through. The network is used to simulate real world and it combines real data (the network) with user-defined specifications of the properties of a real data dataset. Moreover, it has the ability to create hotspots, e.g., further populate the cities of a country. MOTO [37] is a trace generator for the moving objects application domain, a spatio-temporal data workload generator. Its main strength is scalability for large networks (from Brinkhoff [36]) and many moving objects.

2.3 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

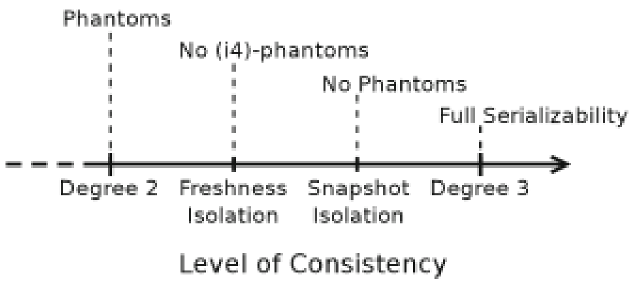


Figure 2.5: Level of Consistency [5]

and query operations. As such, they represent important building blocks for several of the distributed platforms for spatio-temporal data.

2.3.1 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. An SI implementation must respect two key properties [38]:

i) *(Snapshot Read) All operations read the most recent committed version as of the time when the transaction began.*

ii) *(No Write-Write Conflicts) The write sets of each pair of committed concurrent transactions must be disjoint.*

SI is weaker than serializability as it does not detect read-write/write-read conflicts between concurrent transactions, but provides an interesting property, called *timeslice* semantics: a query result reflects all previously executed transactions and none of the transactions executed after the query has started. *Timeslice* semantics, when applied to individual operations (insert, delete, update of an object in the index) of a (spatio-temporal) index, is, in fact, sufficient to ensure the equivalence to a serial schedule of those operations. However, it should be noted though, if the indexes are used in a more complex user level transaction, the whole application can be exposed to an anomaly that goes under the name of *write skew* (see Figure 1 for an example).

Algorithm 1: Assume two nodes N and N' are in a given range R. Any sequential execution will never remove both N and N'. Running with SI, both transactions take a snapshot of the current data base at transaction start. Concurrently, T1 and T2 respectively read N and N', before writing. Since SI does not detect read-write/write-read conflicts, it is possible that both nodes are removed. As both transactions do not write to the same node (no *Write-Write Conflict*).

```
1: Transaction 1 (T1)
2: begin
3: if( query($N$ is in range $R$) == $true$ )
4:   update (remove $N'$ from $R$)
5: commit
1: Transaction 2 (T2)
2: begin
3: if( query($N'$ is in range $R$) == $true$ )
4:   update (remove $N$ from $R$)
5: commit
```

The key benefit of SI is that it eliminates all interference between update and query transactions. Queries can access the entire database without being affected by writes made by concurrent updates. Moreover, queries are never aborted, nor they cause an abort of a update. On the other hand, typical SI implementations rely on multi-versioning schemes, which can introduce non-negligible overheads in terms of memory consumption and version management (e.g., version visibility logic and garbage collection of obsolete versions).

Finally, Freshness Isolation is a consistency criterion weaker than SI that allows various concurrency anomalies, called phantoms in the literature [39], 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 2.6, 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, black dots represent objects initial positions and white 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.

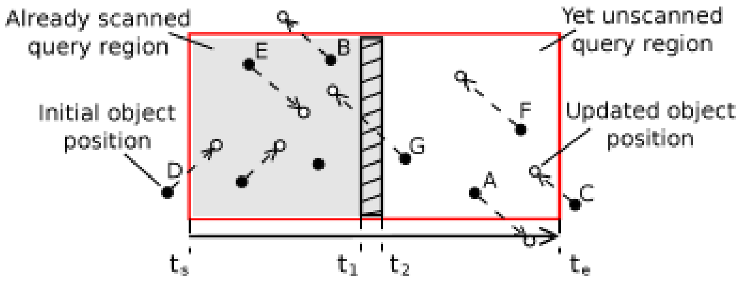


Figure 2.6: Parallel updating and querying [5]

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.

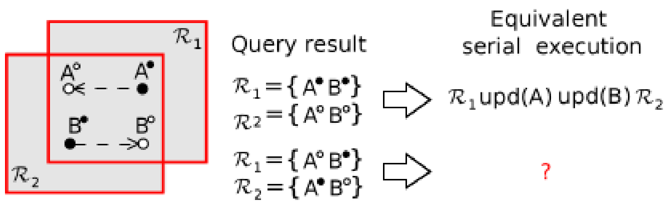
Of all of the phantoms, only i4 phantoms must be avoided to ensure *freshness* semantics. i4 phantoms happen when an object, yet to be scanned, updates its position to an already scanned area (object G). Thus, the object is still in the query range but it seems to have disappeared. It has been argued by [5] 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 2.3.2.4.

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 reports some (fresher) objects that have their last reported positions after ts (and before te) in the query range. We formally conclude *freshness semantics* applied to a range query. We use $pos^\bullet$ and pos° to denote the previous and current position of an object, respectively, and t_u denotes the last time an object was updated.

Definition 1 Given a range query $\mathbb{R}$ with processing time [t_s,t_e], its result O is said to satisfy freshness semantics if for any object o, the following hold:

1. if $o.t_u < t_s$ then $o \in O$ if and only if $o.pos^\circ \in \mathbb{R}$
2. if $t_s < o.t_u < t_e$ then
   (a) if $o.pos^\bullet \in \mathbb{R}$ and $o.pos^\circ \in \mathbb{R}$ then $o \in O$
   (b) if $o.pos^\bullet \notin \mathbb{R}$ and $o.pos^\circ \notin \mathbb{R}$ then $o \notin O$
   (c) if $o.pos^\bullet \notin \mathbb{R}$ and $o.pos^\circ \in \mathbb{R}$ then o may or may not belong to O
   (d) if $o.pos^\bullet \in \mathbb{R}$ and $o.pos^\circ \notin \mathbb{R}$ then o may or may not belong to O

The first part of Definition 1 says that if o was only updated before the query started then whether or not o is in the query result is determined by its up-to-date position. The second part deals with objects that are updated during the query processing and covers the cases discussed already. Observe that cases 2c and 2d imply that if one position is within the query range while the other is not, the decision to add o to the result is arbitrary. Such happens since updates after t_s are not necessarily seen by the query during [t_s,t_e]. For example, object A is updated after the query started, however its updated position lands outside the query range, thus it is not reported. Finally, *freshness semantics* reports the fresher updated objects and cannot guarantee serializable semantics (see Fig. 2.7) as the result of two queries in the same area may not be equivalent to a serial execution.



$\mathcal{R}_1$
$\mathcal{R}_2$
Query result
Equivalent serial execution
$\mathcal{R}_1 = \{A^\circ B^\bullet\}$
$\mathcal{R}_2 = \{A^\circ B^\circ\}$
$\mathcal{R}_1$ upd(A) upd(B) $\mathcal{R}_2$
$\mathcal{R}_1 = \{A^\circ B^\bullet\}$
$\mathcal{R}_2 = \{A^\bullet B^\circ\}$
?

Figure 2.7: *Freshness* semantics is not serializable [5]

2.3.2 Indexes Implementation

In this section we review several existing indexes for spatio-temporal data, which we list in Table 2.2. 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 Chapter 3, 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 a tree or a grid. The primary indexes are used to allow efficient retrieval of objects based on their spatial position during query operations.

A secondary index structure is also used, a hashtable, which stores objects using their id 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).

Name	Grid vs Tree -based	Multi-threaded	Semantics
u-R-tree	Tree	No	–
u-Grid	Grid	No	–
Serial	Grid	Yes	Timeslice
PGrid	Grid	Yes	Freshness

Table 2.2: Indexes Configuration

2.3.2.1 u-R-tree

R-tree based indexing algorithms have been investigated for decades in the literature [4, 6, 40]. The R-tree is known for its support for a large number of different query algorithms, and its ability to index spatial-extended objects (objects larger than a single point). However, the main issue with the R-Tree is its poor performance in update management — which motivated the design of the u-R-tree algorithm. Both the R-tree and u-R-tree algorithms are not designed to support concurrent manipulations, i.e., they assume a single threaded execution model. We focus on the u-R-tree algorithm and further overview it.

Figure 2.8 depicts the u-R-tree's structure. The u-R-tree is a hierarchy structure of Minimum Bounding Rectangles (MBR). An MBR is the smallest rectangle that encloses a group of objects. There are

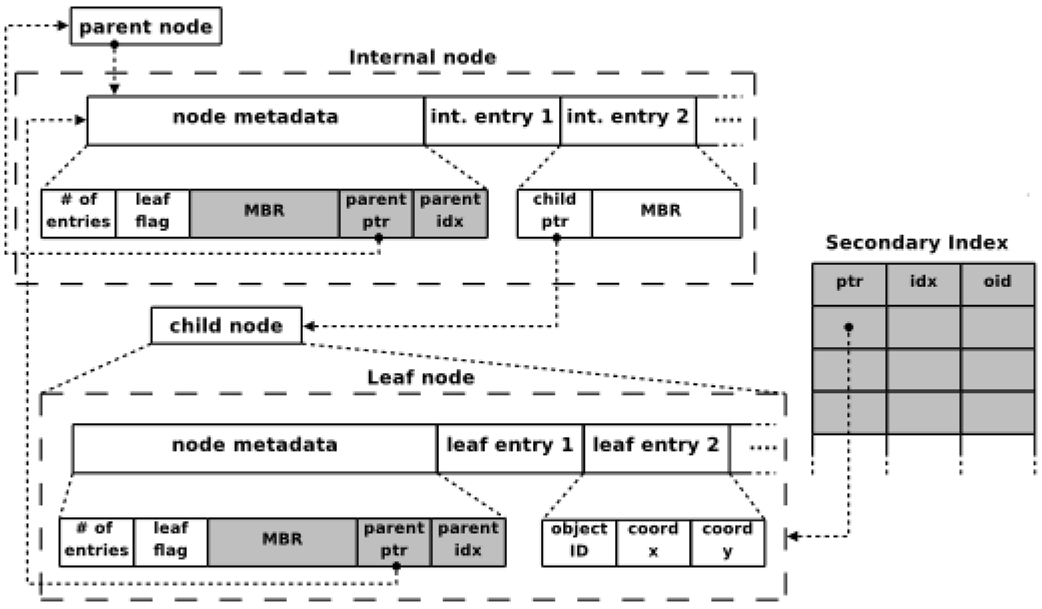


Figure 2.8: Structure of a conventional R-tree (the gray elements are present only in the u-R-tree [4].

two types of nodes in this tree. Internal nodes, which have other nodes as child, and leaf nodes, who are at the bottom of the tree and contain moving objects in them. These nodes are organized according to their MBR. Parent nodes MBR contain all child nodes MBR. Finally, the u-R-tree is defined with two parameters that are needed to adjust the tree in order to achieve a good tree balance and thus a good performance. These are node size (*ns*), expressed in cache lines and minimum children (*mc*) expressed as a fraction of the full node. In any node of the tree its entries must occupy at least *mc* percentage of *ns* otherwise the node is considered underfull and is merged with the parent node. If a leaf node is too full with objects, it is considered a node overflow, and it splits generating two leaf nodes.

Range Query: The *range query* in the u-R-tree is performed as a depth-first traversal from the root down to the leaves accessing the nodes with MBRs overlapping the range of the query. When the leaf nodes are finally reached, objects that satisfy the range query are output.

Updates: Update operations in the u-R-tree usually have worse performance when compared to queries. In the following, we describe the update operation and illustrate the reasons of this inefficiency. Updates are performed by two separate top-down operations, deletion and insertion.

The deletion operation descends the tree from the root searching the old position of the object. This is done by recursively accessing the nodes that contain the position of the object in their MBR. MBRs may overlap, so more than one path can be visited while searching for the object. Once the required leaf node is located, the appropriated entry is deleted. Finally, ancestors (all nodes above the current) MBRs, may become not minimum (removal of the object may shrink the MBR), thus requiring traversing the tree back to the root adjusting the MBRs. Furthermore, the nodes may become underfull (as earlier explained), requiring an expensive reinsertion of their entries.

The second part of the update operation uses the insertion algorithm. Insertion begins by traversing

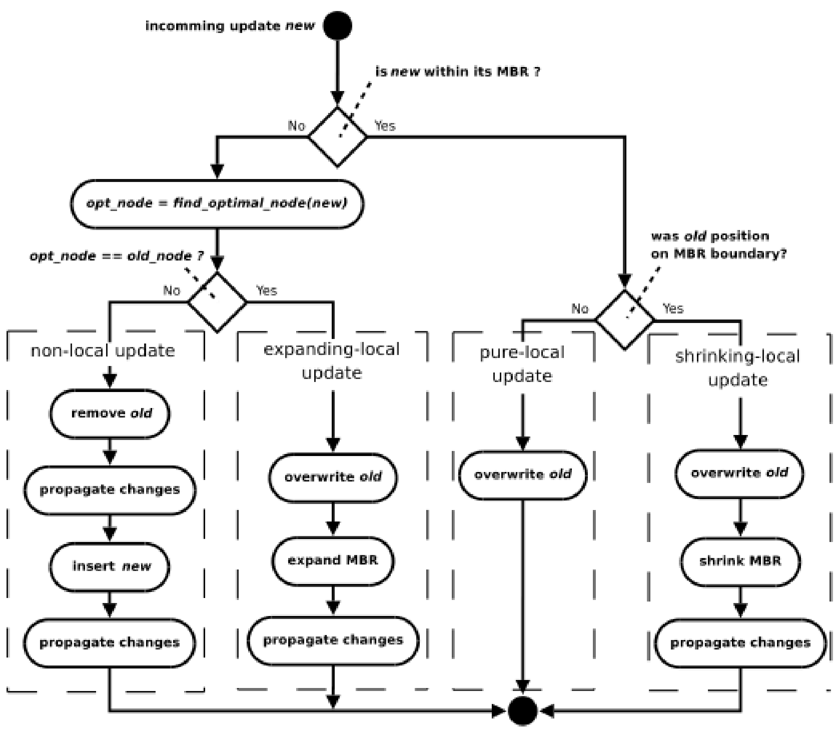


Figure 2.9: Bottom-up update in the u-R-tree [4].

the tree from the root to the leaf node as well. At each node a heuristic function is called to choose the most suitable path to further descent the tree. When a suitable leaf node is located, the object is inserted there. Once again, by inserting a new object, ancestors MBRs may become invalid and have to be adjusted traversing the tree to the top. Also when inserting a new object on the leaf node, we might be causing a node overflow (earlier explained), which originates a node split on that leaf node. By doing this we are adding another entry to the parent node, which itself may exceed node capacity, and split. The split may propagate up the tree. Summarizing, a single update results in at least 3 tree transverses, thus this being the main reason why updates in the R-Tree are very poor, performance wise.

Bottom-up Updates: In order to improve updates performance, u-R-tree makes updates in a bottom-up fashion. To do so, an extra secondary index (hashtable) (Figure 2.8) is used, which facilitates updates to the tree providing direct access to objects inside the tree. However, bottom-up updates do not solve problems as the leaf node overflow, which split may propagate up the tree. Nevertheless, there is no need to make a costly traverse to the tree searching for the object and near constant $O(1)$ updating time can be achieved. This is a simplified version of the algorithms proposed by Lee et al [41] in order not to complicate too much the already CPU-heavy algorithms (updates and queries).

Figure 2.9 demonstrates how updates are classified depending on their new position. If the new position is within its MBR, two options follow. First, if the old position is within the MBR boundary, a pure-local update occurs. Second if the old position is in the limit of the MBR then a shrinking-local update occurs. Conversely, if the new position is not within its MBR, two options follow. First, if its optimal node is the same, a expanding-local update occurs. Second if its optimal node is a different one,

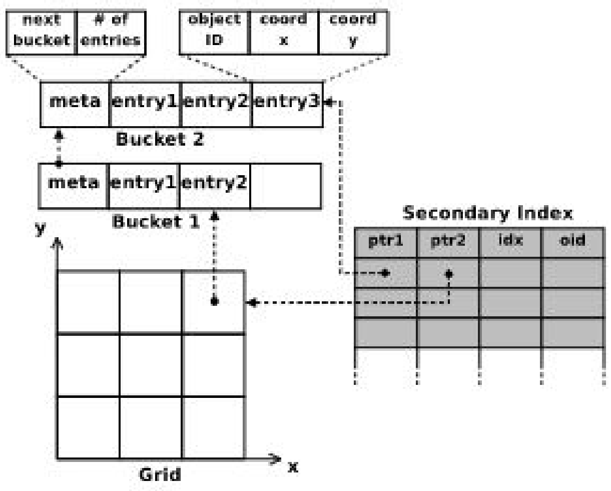


Figure 2.10: u-Grid Structure [4]

a non-local update occurs.

2.3.2.2 u-Grid

u-Grid is a single-threaded index structure that offers high performance for traffic monitoring applications in single-threaded settings [4]. Queries are serviced using a uniform grid [42], while updates are facilitated via a secondary index in bottom-up fashion. A uniform grid (Figure 2.10) 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, as it occurs with the u-R-tree. Objects within a particular grid cell belong to that cell. Grid cells are stored as a two-dimensional array. Each grid cell stores a pointer to a linked list of buckets, these 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).

Updates are performed in a bottom-up fashion and queries are made by searching/scanning the grid (primary index). The pseudo code for the update algorithm (Algo. 2) is defined as follows. A new ObjectTulpe is to be updated, its coordinates are extracted to calculate its corresponding cell (Line 1). The secondary index entry (sie) is extracted with the new ObjectTuple object id (oid), which is the secondary index (hashtable) key. However, the object in the hashtable is the current version of object, not the new version. Hence, OldCell and current object, obj, are extracted from the sie. Both objects cells are used to check whether the update is local (same cell), or non-local (Line 5). If the former is the case, obj is overwritten with new's coordinates in the grid. The sie is not changed since the object remains in the same location in the grid. If the latter occurs, the current object has to be physically deleted (nullify hashtable entries and remove from grid) and the new object is inserted (added in hashtable and grid). These procedures are managed by the insert and delete algorithms.

Algorithm 3 is the delete function. The sie of the object to deleted (Obj) is used to extract its location in the grid. Obj is to be overwritten by the last object of the first bucket (Cell points to first bucket). The object is overwritten in Line 4. Once the object is overwritten, the number of objects in the first bucket

Algorithm 2: update(ObjectTuple new)

```
1  newCell = computeCell(new.x, new.y) ;
2  sie = SecondaryIndex.lookup(new.oid);
3  oldCell = sie.cell;
4  obj = getObj(sie.bckt, sie.idx) ;                          // object tuple
5  if newCell == oldCell then
       // local update
6      obj.x = new.x;
7      obj.y = new.y;
8  else
       // Nonlocal update
9      delete(sie) ;                                      // physical deletion
10     insert(new, newCell, sie) ;                       // physical insertion
11 end
```

is decremented. Once reached 0 objects, the first bucket is to be deleted, and the next bucket is to be set first (Line 6-8). Finally, the deleted objects references are nullified from the hashtable. In contrast, the sie of lastObj (object used to overwrite) is updated with its new position in the grid, concluding the delete algorithm.

Algorithm 3: delete(SIEntry sie)

```
1  Obj = sie.Bckt.entries[sie.Idx];
2  firstBckt = *sie.Cell ;                       // cell refers to the 1st bucket
3  lastObj = firstBckt.entries[firstBckt.nO - 1];
4  write(lastObj, Obj) ;                               // lastObj copied over Obj
5  firstBckt.nO– ;                                // decrement number of objects
6  if firstBckt == 0 then
7      *sie.Cell = firstBckt.nxt ;          // No more queries can enter firstBckt
8      free(firstBckt);
9  end
10 Nullify all Obj references in sie;
11 Lookup for lastObj's sie and update it;
```

The insert algorithm inserts the new object in the new cell. If the first bucket is full a new one must be allocated (Line 2,3). The object is inserted in a free position of the first bucket (Line 5,6). The sie is updated with the new object position in the grid (Lines 7-9). Finally, the first bucket number is incremented, concluding the inset algorithm.

Algorithm 4: insert(ObjectTuple new, Cell cell, SIEntry sie)

```
1  firstBckt = *cell ;                           // cell refers to the 1st bucket
2  if $isFull$(firstBckt) then
3      Allocate new bucket and make it first;
4  end
5  freePos = firstBckt.entries[firstBckt.nO];
6  write(new, freePos) ;                                // new written to freePos
7  sie.cell = cell;
8  sie.bckt = firstBckt;
9  sie.idx = firstBckt.nO;
10 firstBckt.nO++ ;                               // increment number of objects
```

The range query retrieves all objects form a rectangle area. Algorithm 5 depicts the pseudo-code. The cells covered by the query range are computed (Line 2). Each cell is scanned (CellScan) for their objects, which are then retrieved and added to the result (Line 4,5). Similar processing is done to partially covered cells.

Algorithm 5: rangeQuery(Rect q, int ts)

```
1 res = 0 ;                                   // container for storing the result
2 cells = computeCoveredCells(q);
3 for cell ∈ cells do
4     objects = CellScan(cell);
5     res.add(objects);
6 end
7 Similar processing is performed for partially covered cells;
8 return res;
```

Finally, we review the how cells are scanned in the CellScan algorithm (Algorithm 6). Each bucket is read, starting from the first bucket until there are no remaining buckets (Line 2). Buckets are read from the beginning to the end (Line 3). Consequently, objects are retrieved and added to the result (Line 4,5). Finally, the next bucket is set to be read (Line 7), concluding the CellScan algorithm.

Algorithm 6: CellScan(Cell cell)

```
1 objects = 0 ;                               // container for storing the result
2 while bckt != null do
3     for idx := 0; To bckt.nO - 1; Step +1 do
4         obj = readObj(bckt.entries[idx]);
5         objects.add(obj);
6     end
7     bckt = bckt.nxt;
8 end
9 return objects;
```

2.3.2.3 Serial

Serial is a multi-threaded version of u-Grid, with a fine-grained (non-strict) 2 phase read/write locking [43] scheme, which ensures *timeslice* semantics. *Timeslice* semantics are guaranteed by ensuring that there are no concurrent updates and queries in the same cells, however, there can be concurrent operations in different cells. Moreover, as queries are read-only, they acquire a read lock, making it possible to have concurrent queries on the same cells. In contrast, updates to an object preclude any concurrent access to the cell, either update or query — which represents one of the main limitations of this relatively simple algorithm. Synchrony between the primary and secondary index is held with cell locks, as an updater locks the primary index cell and the secondary index entry with the same cell lock.

Range Query: Range queries (readers) acquire (read) cell locks in shared mode. A growing phase starts until all cells from the grid, within the query range, are locked. No updates are made to locked cells, thus, queries are now able to scan cells and read their buckets. The next phase is the decrementing phase. Once a cell is read, its locks are immediately released, ensuring that queries acquire locks the minimum possible time. The locks can be released safely before all the cells are inspected since range queries are read-only and never aborted.

Bottom-up Updates: Before any object modification, each updater acquires a (write) lock on the current object's cell. If it is a local update, the objects' entries in the grid are overwritten with the new ones, and the lock is released. If the update is non-local, the updater tries to lock the new (destination) cell as well. In case the lock is obtained successfully, the old object is deleted from the old cell and inserted into the new cell. Finally, both cell locks are released. If the lock on the new cell is not acquired, the update is aborted and restarted to avoid deadlocks.

Overview: Overall, locks are expected to be held for a very short duration of time as there is no grid refinement or re-balancing. At most two cells are locked during an update, and multiple queries can access the same cells. Results from [5] indicate that Serial has a limited scalability, since it has a high sensitivity to update-update and update-query contention. Hence, Serial suffers the most with hotspots, where an above average number of objects are scanned/updated in neighbour cells.

2.3.2.4 PGrid

PGrid (Fig. 2.11) is a multi-threaded version of u-Grid with a fine-grained 2 phase locking [43] scheme. Unlike Serial, 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 promotes further parallelism between updates and queries, unlike with Serial. In order to ensure freshness semantics, PGrid uses two additional ideas/mechanisms:

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 2.6) 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

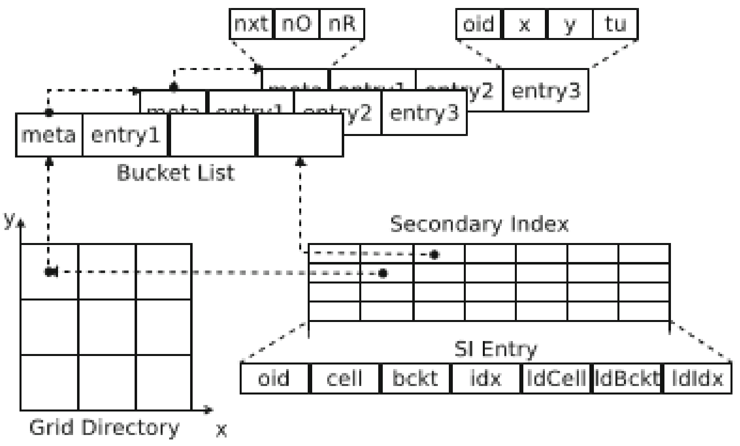


Figure 2.11: PGrid Structure [5]

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).

PGrid's updates are also made in bottom-up fashion (see Fig. 2.11). Local updates are treated the same way as previous implementations, since no explicit object insertion/deletion is required, and since objects stay in the same cell. Then, it is only required to overwrite the object. Non-local updates, however, do not follow the same pattern. Since i4 phantoms must be avoided, there is the need to maintain the old position of the object in case the range query misses the new updated position. Thus, PGrid introduces a new notion of *logical deletion*. When a non-local update occurs, the old position of the object is kept (logically deleted), thus its entries on the secondary index are moved from its current entries (cell, bckt, idx) to the logically deleted entries (ldcell, ldbckt, ldidx). Moreover, in the primary index, a deletion flag is set to the object's timestamp. This informs the queries that this is the old position of the object. The next update to the same object will delete the object's old position (logically deleted) from both indexes, performing a physical deletion.

An assumption is made by [5] that the processing time of a query [ts,te] is shorter than the time between two consecutive updates. Therefore, case i4 can be handled by keeping one previous object position in the index, which will be deleted in the next update. Doing so guarantees that a query always encounters at least one object version and does not miss any objects. In Figure 2.6, the query then reports G's previous position (black dot). However, in cases where a query is longer than two consecutive updates, a simple timer may be added, where if the query surpasses the timer's limit then it is aborted and redone.

PGrid allows concurrent updates and queries in the same cells, hence, it is required that read/write operations made to the primary index (Grid) are made atomically. This ensures that the execution of updates does not jeopardize the correctness of queries. To do so, PGrid considers two lock-free mechanisms, specifically, Optimistic Lock-Free Index Traversal (OLFIT) and Single Instruction Multiple Data (SIMD).

OLFIT was designed as a cache-conscious CC scheme for main-memory index structures on shared-memory multiprocessor platforms [44]. This approach maintains a latch and a version number for each object. Updates' always obtain a latch first, to ensure no concurrent writes to the same data item. In addition, before a latch is released, an updater increments the version number. A reader starts by copying the version number and optimistically reads the data without latching. Then, if the latch is free and the current version number is equal to the copied one, the read is consistent; otherwise, the reader starts over. Thus, OLFIT ensures optimistic reads which are favourable for multi-core architectures since they avoid the memory write required by latching. Since, with latching, even if the actual object data does not change, the entire cache line where the memory write was made will be invalidated, forcing all other cores to also invalidate the cache lines with that memory address.

SIMD is a technique currently supported by commodity processors. This technique makes possible the transportation of multiple data items via vector operations. Such processors come with instructions for loading and storing data into SIMD registers. Thus, SIMD can be employed in PGrid in order to make writes/reads to objects atomic. Micro-benchmarks made by [5] confirm that loading/storing a double quad-word (128-bit value) into/from SIMD registers (xmm) from/to a memory location aligned on a 16-byte boundary is atomic [14].

We proceed with the detailed description of PGrid's algorithm, which we will use as starting point to derive an HTM-based variant in Chapter 3.

The update algorithm, Algorithm 7, starts by computing the cell where the object is to be updated (Line 1). Immediately after, it locks the object since the sie (secondary index entry) of this object will have to be accessed (Line 3). With the sie, we can check if the object has a logically deleted (ld) position (Line 6), in this case, it is deleted. As previously mentioned, an assumption is made indicating that the processing time of a query is shorter than the time between two consecutive updates. Therefore, it is safe for updates to delete their ld positions, this is called a physical deletion. If deletion fails, the object lock is unlocked and the whole update operation restarts, in order to avoid deadlocks (Line 8,9). Following, we must check if the update is local or non-local. This is done comparing the current object's cell (oldCell) and the new updated objects' cell (newCell) (Line 10). If it is a local update, we overwrite the current object with the new object in the grid index (primary). Notice that this write (Line 11) must be atomic, as there could be concurrent queries reading the same cell. However, if it is a non-local update, we first need to logically delete the current object. This is done by moving the object's entries in its sie from the new to the ld entries (Lines 13-15). Moreover, the timestamp of the object in the grid index must be negated so queries distinguish current and ld objects (Line 17). Finally the new object is inserted in the new cell (Line 16).

Algorithm 8 contains the pseudo code for the physical delete operation (Line 7 of Algorithm 7). Physical deletion of objects is secured using a cell lock, so that only a single thread is able to perform

Algorithm 7: Update(ObjectTuple new) [5]

```
1  newCell = computeCell(new.x, new.y);
2  lockObj(new.oid);
3  sie = SecondaryIndex.lookup(new.oid);
4  oldCell = sie.cell ;
5  obj = getObj(sie.bckt, sie.idx);                // object tuple
6  if hasLD(sie) then
7      if !delete(sie) then                        // physical deletion
8          unlockObj(new.oid);
9          go to 2 ;                               // try again
10 if newCell == oldCell then
       /* Local update */
11     writeObj(new, obj);                         // new copied over obj
12 else
       /* Nonlocal update */
13     sie.ldCell = sie.cell ;
14     sie.ldBckt = sie.bckt ;
15     sie.ldIdx = sie.idx ;
16     insert(new, newCell, sie);                  // physical insertion
17     obj.tu = -new.tu;                           // mark as logically deleted
18 unlockObj(new.oid);
```

Algorithm 8: bool delete(SIEntry sie) [5]

```
1  lockCell(sie.ldCell);
2  ldObj = sie.ldBckt.entries[sie.ldIdx];
3  firstBckt = *sie.ldCell;                        // cell refers to the 1st bucket
4  lastObj = firstBckt.entries[firstBckt.nO - 1];
5  if tryLockObj(lastObj) then
6      writeObj(lastObj, ldObj);                   // lastObj copied over ldObj
7      firstBckt.nO- -;                            // decrement
8      if firstBckt.nO == 0 then                   // is empty?
9          *sie.ldCell = firstBckt.nxt;
                                   // No more queries can enter firstBckt
           waitUntilNoReaders();
10         free(firstBckt);
11     Nullify all ld references in sie ;
12     Lookup for lastObj's sie and update it;
13     unlockObj(lastObj); unlockCell(sie.ldCell);
14     return true;
15 else
16     unlockCell(sie.ldCell);
17     return false;
```

deletion at a time, in the same cell. Deletion starts by using the objects' sie to lock the cell where its ld position resides (Line 1). This is necessary in case the first bucket of that cell becomes empty and has to be deleted. The ld object is extracted using the new objects sie, received as argument (Line 2). In the grid index, to perform a deletion of an object, the last object of the first bucket is always used to overwrite the object to be deleted. Hence, we extract the last object of the first bucket (Line 3,4). We try to lock *lastObj* in order to perform the deletion, however, if the lock is already acquired by another thread, the entire operation is restarted falling back to the update function (Lines 16,17). This costly path is necessary to avoid dead locks, which could occur in very specific situations.

Nevertheless, if the lock is successfully acquired, the ld object is overwritten atomically by the last object of the first bucket, and the number of objects in the first bucket is decremented (Lines 6,7). If the first bucket becomes empty then the next bucket is assigned first (Line 9) and the first is freed (Line 10). However, there can still be queries reading from this respective cell. Hence, deletion must wait for all queries to conclude their scanning in order to free the first bucket, which occurs in the *waitUntilNoReaders* function. Finally, all ld references are nullified in the sie (sie of ldObj) and the lastObj's sie is updated since this object was moved in the grid to overwrite ldObj (Lines 11,12). Eventually deletion concludes by unlocking both locks.

Algorithm 9: insert(ObjectTuple new, Cell cell, SIEntry sie)) [5]

```
1  lockCell(cell);
2  firstBckt = *cell ;                    // cell refers to the 1st bucket
3  if isFull(firstBckt) then
4      Allocate new bucket and make it first;
5  freePos = firstBckt.entries[firstBckt.nO];
6  writeObj(new, freePos);               // new copied over freePos
7  sie.cell = cell ;
8  sie.bckt = firstBckt ;
9  sie.idx = firstBckt.nO ;
10 firstBckt.nO++;                                      // increment
11 unlockCell(cell);
```

Object insertion is relatively simple (Algorithm 9). A new object is always inserted in the first bucket, which is pointed by the cell (Line 2). In case the bucket is full, a new bucket is allocated and the necessary pointers are updated, so that the new bucket becomes the first (Line 4). The first free position at the end of the first bucket is determined (Line 5), and the new tuple is written (Line 6). The fields *cell*, *bckt*, and *idx* in the sie are also updated accordingly (Lines 7–9). Finally, the number of elements in the bucket is incremented. This process is secured by the target cell lock, other threads cannot concurrently perform any deletion or insertion of objects in the specific cell.

Algorithm 10 represents the range query algorithm, which is structured as follows. The algorithm takes two inputs: a two-dimensional rectangle specifying the query range (*q*) and an integer value specifying the query's timestamp (*ts*). The algorithm returns the objects covered by the query range. For simplicity, Algorithm 10 shows only the processing of fully covered cells (computed in Line 2). Thus,

Algorithm 10: rangeQuery(Rect q, int ts) [5]

```
1  res = ∅;                                // container for storing the result[3]
2  cells = computeCoveredCells(q);
3  foreach cell ∈ cells do
4      objects = pCellScan(cell);
5      foreach obj ∈ objects do
6          if obj.tu > 0 then
7              res.addAndOverwrite(obj);
8          else if abs(obj.tu) > ts then
9              res.addIfNoSuch(obj);
10 Similar processing is performed for partially covered cells;
11 return res;
```

the extra check (whether an object is within the range), done for partially covered cells, does not appear in the algorithm.

All cells buckets are scanned in order to retrieve the objects tuple (Line 4). A filtering mechanism is applied to objects in order to gather only the most up-to-date objects and assure *freshness* semantics (Line 6-9). If a current version of the object is read (objects' timestamp is higher than 0), the object is immediately added, overwriting any previous object in the result.

In contrast, a negative timestamp signals that the tuple contains a ld object. A ld object is added to the result if two conditions hold. First, the absolute value of its timestamp exceeds that of the queries'. This implies the object was updated after the query started, hence, it must be considered to guarantee the i4 phantom does not occur (Fig. 2.6, Line 8). The second condition is realized via the container (Line 9): The tuple is only added if the result does not contain an object with the same *oid*. This as well as add-and-overwrite functionality can be efficiently supported using an unordered_map from the Standard C++ library.

Before reading a bucket, queries use a compare and swap (CAS) instruction to atomically increment the number of readers (nR) of that bucket, which signals updates that a query is reading that bucket. The opposite occurs when queries leave buckets. To note, the CAS instruction is only used to allow concurrent query threads to modify the nR of a specific bucket.

Nevertheless, queries always scan cells from the first bucket to the last. To ensure a safe read of the first bucket, a cell lock is briefly acquired as concurrent updates may delete the first bucket (Lines 2-8). The cell lock blocks any insertion/deletions from concurrent updates, hence, ensuring the first bucket will not be deleted or modified. Solely incrementing the nR of the first bucket is not enough as the bucket may be deleted/modified before the query increments its nR. For the subsequent buckets, the counter can be accessed safely, as updates only insert/delete objects using the first bucket (Lines 13–16).

Recall that an updater can move the last object of the first bucket (to overwrite an object to be deleted). If a query scans a bucket from its beginning, the moved object could be missed if it is moved from the as-yet-unscanned part of the bucket to the already scanned part. To eliminate this problem, objects within a bucket are examined starting from the last entry of the bucket (Line 10).

Finally, PGrid relies on two alternative mechanisms to guarantee that queries do not observe partial object updates performed by concurrent update operations (which modify several metadata of the object).

Algorithm 11: pCellScan(Cell cell) [5]

```
 1 objects = $\emptyset$;                        // container for storing the result
 2 lockCell(cell);
 3 if isEmpty(cell) then
 4     unlockCell(cell);
 5     return objects ;
 6 bckt = *cell;                                // cell refers to the 1st bucket
 7 aInc(bckt.nR);                               // atomic increment
 8 unlockCell(cell);
 9 while bckt ! = null do
10     for idx = bckt.nO - 1 downto 0 do
11         obj = readObj(bckt.entries[idx]);
12         objects.add(obj);
13     if bckt.nxt ! = null then
14         aInc(bckt.nxt$\rightarrow$nR);              // atomic increment
15     aDec(bckt.nR);                           // atomic decrement
16     bckt = bckt.nxt ;
17 return objects ;
```

Chapter 3

Hardware Accelerated Big Spatio-Temporal Data Indexes

In this chapter we present HTM-based spatio-temporal indexes. The starting point to derive such algorithms are two state of the art solutions, namely u-Grid and PGrid (see Section 2.3). For u-Grid we consider implementations based both on plain HTM interface, as well as on HRWLE (see Section 2.1.1). For PGrid, we consider how to integrate HTM in the OLFIT-based version, and consider the possibility of ensuring atomicity of the objects scanned during the query operations via HTM transactions spanning a tunable number of objects. Table 3.1 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 4.

3.1 Analysis of HTM Spatio-Temporal Indexes

The HTM-based spatio-temporal indexes that we developed differ by the number of transactions that are used to support queries. We do so to solve two HTM drawbacks: contention and HTM capacity.

Contention happens when concurrent transactions access the same cache line and at least one of the two transactions writes to it. Contention may occur between updates and between updates and queries. However, algorithms such as PGrid's range query (Alg. 10), increment/decrement the variable (nR) in their cells' buckets as to alert updates that there are queries currently scanning those buckets. Hence, PGrid's queries are not read-only and can therefore, potentially, contend with each other. However, contention generated by this is negligible in practice.

Updates are short and access a small amount of shared variables. Indeed, the use of the bottom-up

Name	Base Algorithm	HTM Interface	# Update Transactions	# Query Transactions
u-GridHTM	u-Grid	HTM	1	1
PGridOHTM	PGridOLFIT	HTM	1	Partitioned
u-GridHRWLE	u-Grid	HRWLE	1	0

Table 3.1: Spatio-temporal indexes HTM solutions

update scheme allows direct access to the grid (primary index) using an hashtable (secondary index). The hashtable is also TM-friendly as objects are stored by their object id, hence, only a single hashtable entry is accessed to fetch a specific object, avoiding further conflicts with other updates. Hence, contention between updates is not a major concern for HTM.

The major cause of contention are concurrent updates and queries. The grid, being composed by a linked list of buckets is not TM-friendly, as queries are forced to read the entire list to return their result. Any modification (update) to a previously read bucket/cell will force an abort of one of the transactions (update or query), depending on the architecture. Hence, it is imperative to maintain contention between updates and queries low to achieve good performance.

The other main issue of HTM relates with its limited capacity. In the considered spatio-temporal indexes, the capacity issue is particularly relevant for queries, as updates access a small number of memory locations. In contrast, queries may have to scan multiple cells. All memory addresses of the objects inside the cells buckets have to be stored, which consumes the cache's store capacity (reviewed in Section 2.1.1.1). Hence, it is imperative to maintain small query transaction sizes as so not to exceed transactional capacity.

This is precisely the first approach we took when designing PGridHTM, in which queries scan cells using multiple transactions. Hence, contention between updates and query transactions drastically lowers, and memory footprint is not exceeded as we pre-define the size of query transactions. In contrast, the second approach does not rely in transactions to perform the cell scanning. Instead, we use the atomic instructions (i.e., OLFIT), present in PGrid, to perform the cell scanning. Doing so reduces contention as conflicts will only occur directly between objects being updated/read concurrently. Moreover, memory consumed by queries will be negligible as OLFIT is used to perform the query scan.

Finally, workloads have a big impact on HTMs performance. Query intensive workloads allow queries to run with larger transactions due to the lower update count, which generates less contention with queries. In contrast, queries read sets quickly fill HTMs capacity, hence, transaction sizes cannot overly increase.

On the other hand, update intensive workloads may force queries to run with a smaller transaction size, as the sheer amount of concurrent updates greatly increases contention between updates and queries. Capacity aborts are a minor issue in these workloads due to the low amount of queries.

After having discussed the main challenges/performance pathologies that we expect to encounter when using HTM to build concurrent spatio-temporal indexes, we can now describe the proposed solutions.

3.2 u-GridHTM and u-GridHRWLE

Our first approach was to straightforwardly apply HTM to a state of the art single-threaded index, u-Grid [4], by wrapping all of its operations with HTM transactions (see Algs. 12 and 13). 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, which is detailed in Appendix A.1.

When applying HTM this way to indexes, the resulting consistency level is Degree 3 (Fig. 2.5), providing full serializability. 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.

Algorithm 12: update(ObjectTuple new)

```
1  newCell = computeCell(new.x, new.y) ;
2  TM_BEGIN_SGL( ) ;                                  // begin transaction
3  sie = SecondaryIndex.lookup(new.oid);
4  oldCell = sie.cell;
5  obj = getObj(sie.bckt, sie.idx) ;                       // object tuple
6  if newCell == oldCell then
       // local update
7      obj.x = new.x;
8      obj.y = new.y;
9  else
       // Nonlocal update
10     delete(sie) ;                                  // physical deletion
11     insert(new, newCell, sie) ;                   // physical insertion
12 end
13 TM_END_SGL( ) ;                                      // end transaction
```

Algorithm 13: rangeQuery(Rect q, int ts)

```
1  res = 0 ;                              // container for storing the result
2  cells = computeCoveredCells(q);
3  TM_BEGIN_SGL( ) ;                                  // begin transaction
4  for cell ∈ cells do
5      objects = pCellScan(cell);
6      res.add(objects);
7  end
8  Similar processing is performed for partially covered cells;
9  TM_END_SGL( ) ;                                      // end transaction
10 return res;
```

A similar approach was used to derive u-Grid with 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.

3.3 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 2.3.1), already present in PGrid. In the following we depict the main changes brought by PGridHTM:

1. HTM is used to elide object and cell locks. As reviewed in section 2.3.2.4, when a non-local update occurs, a deletion or insertion of an object may incur in the deletion or creation of a new bucket (see Algs. 8 and 9). These operations have to be performed by a single thread in order to allow correct modifications to the first bucket (pointed by the cell). Therefore, these operations are guarded with cell locks, preventing any concurrent deletion and insertion of objects in the cell. The other operation guarded by cell locks is the reader subscription into a cell (see Alg. 11), where queries are required to ensure the nR of the first bucket is incremented before performing a scan. This way, concurrent updates know that the query is currently reading the first bucket of that cell, and hence, they are prevented from deleting the first bucket. Finally, object locks used to serialize concurrent single object updates are also elided.

   By eliding cell and object locks we allow optimistic concurrency in these operations, further increasing parallelism. Even though these operations access shared data that would result in a conflict between concurrent transactions to the same operation (e.g., nR, or the increment/decrement of the number of objects of the first bucket), there can still be coexistent transactions. The conflict window (time span where conflicts occur) does not last the entire duration of the operation. Hence, there can be concurrent transactions to the same operation as long as they do not coexist during each others conflict window. In contrast, the conflicting window with locks spans during the entire duration of operations, limiting parallelism.

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 solution consists in falling back to a SGL, which is subscribed by every transaction (reader or writer). This has the effect of ensuring that whenever a transaction activates the fall-back path, every other concurrent transaction, even if accessing a different object/cell, is immediately aborted and blocked. In case of fall-back activation, thus, a SGL-based approach restricts parallelism with respect to the original PGrid algorithm, where operations accessing different objects and cells can always run concurrently.

   This observation led us to develop a second variant that exploits the original PGrid locking-scheme in order to derive a fine-grained locking fall-back mechanism for HTM transactions. This is relatively simple to achieve, as it only requires to replace each request to acquire a lock L with a transaction begin followed by a read to L. This is sufficient to guarantee that if a transaction activates the fall-back path and acquires a set of locks S (as prescribed by PGrid), it will cause the abort only of transactions that are attempting to elide a lock in S.

This, on the one hand, preserves the original consistency guarantees of PGrid, while, on the other hand, enabling a higher degree of concurrency with respect to the SGL-based approach. Nevertheless, the fine-grained locking fall-back path has a concerning disadvantage. The use of multiple locks adds extra instrumentation, forcing searches in index structures (hashtables, arrays or trees) for the specific locks and consuming additional transactional capacity.

Since we wanted to optimize the acquisition of locks by the HTM, we use padded locks with a size corresponding to each architecture cache line size. This way, we ensure that no false conflicts occur in the acquisition of the locks. We further tested this solution using the hashtable that PGrid uses to store its locks (same hashtable used in the secondary index). We found out that the hashtable has very poor performance with the padded locks, hence, we had to use a different structure to store the locks, we opted by an array. The array is compatible with padded locks, moreover, we know the exact position of the lock in the array before searching it. Hence, searches are made directly to the position where the lock resides, optimizing performance.

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. 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, hence, we explain how we ensure correctness between reads and writes.

   Concurrent read/write and write/read access to the same data (object) by OLFIT and HTM must be properly synchronized to ensure correctness. In case there is a concurrent HTM writer and OLFIT reader, the HTM writer will not detect the OLFIT reader. In contrast, the OLFIT reader reads the version (timestamp) of the object at the beginning and end of the operation. If the latter version is not equal to the former, it means a concurrent update to the object was performed, hence, the reader aborts ensuring consistency. The other possible case occurs with a concurrent HTM reader and an OLFIT writer. In this case the OLFIT writer does not abort, in contrast, the HTM reader is aborted since the accessed data is concurrently modified, naturally triggering a conflict (even without OLFIT). Hence, transactions can be safely concurrent with 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. To do so, we had to modify the delete operation (see Alg. 8) to make the “waitUntilNoReaders” function cope with the transactional query scanning.

In PGrid, a cell lock is used to perform the delete operation, ensuring further queries are blocked from scanning the cell. This way, if the number of elements in the first bucket reaches 0, the "waitUntilNoReaders" function loops over the nR of the first bucket until all queries have left it, and then the bucket can be freed.

However, the delete operation does not cope with the transactional query scanning. Queries are no longer blocked by the cell lock, instead, they are blocked by the HTM fall-back path. The fall-back path mechanism blocks all concurrent transactions, in case the SGL approach is used, or it blocks concurrent transactions on the same cell/object, in case the FGL approach is used. Specifically, the problem surges when the delete operation acquires the fall-back path, and remains in the "waitUntilNoReaders" waiting for queries to leave the first bucket. Since the fall-back path blocks all other transactions (SGL approach) or blocks transactions in the specific cell (FGL approach), a deadlock is created, where queries scanning the first bucket are blocked from leaving it, hence, no progress is made as deletion is waiting for queries to leave the first bucket.

Therefore, we had to resort to a non-blockable fall-back path (see Algs. 30 and 31), where query scanning resorts to the atomic operations found in PGrid (e.g. OLFIT) to complete the scan in case the fall-back path is acquired. This way, the deadlock is resolved since scanning transactions are no longer blocked by the fall-back path, instead, they resort to OLFIT continue the scan as in PGrid.

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 fine-grained locking system of PGrid as its fall-back path. They are respectively named PGridHTM-SGL and PGridHTM-FGL. The full description of the fall-back path algorithms can be found in Appendix A.

The following algorithms are based on the ones in Section 2.3.2.4, which we modify in order to implement an HTM version of these. Hence, these algorithms were previously explained in detail and we now focus on the differences which we introduced to take advantage of HTM. Note that we identify the new modifications to the algorithms with a yellow shading.

3.3.1 PGridHTM-SGL

In this version of PGridHTM we use the SGL as the fall-back path to HTM, all previous locks (cell and object) were removed, and all synchronization relies on HTM or in the SGL fall-back path. Following, we detail the algorithms used in PGridHTM-SGL.

One of the two interface operations, the update algorithm (Alg. 14), is wrapped within a transaction (Lines 2-22). Insert and delete algorithms are a part of the update, hence, they are already inside the transaction (Lines 7, 19). Updates are concurrent with other updates and queries. Hence, they use the SGL fall-back path in order to synchronize them when the available HTM attempts expire (Algorithms 23, 24). The major difference in this algorithm is in the way writes are performed to the primary index. Recall that writes to the primary index must be atomic to ensure that concurrent queries may read the primary index correctly.

When running transactionally, writes to the primary index do not need to use the OLFIT mechanism originally used in PGrid, since HTM already provides atomicity in transactions. Hence, we use a mix of instructions depending whether if we are running on the fall-back path or running transactionally. When running with transactionally, plain writes are performed (Line 13). In contrast, when running in the fall-back path, writes use the original PGrid's OLFIT logic (based on CAS operations, Line 11).

The delete operation (Alg. 15), also distinguishes writes depending on whether they are made transactionally or requiring to the fall-back path (Lines 5, 7). Recall that if the deletion of an object includes the deletion/modification of the first bucket of a specific cell, deletion must wait until no queries are present in the first bucket ("waitUntilNoReaders"), and no more queries may enter the cell during this time. Previously in PGrid (Section 2.3.2.4), this is ensured with a cell lock, which blocks further queries from entering the specific cell. When running with the fall-back path we use the SGL to achieve same effect. In contrast, running transactionally, this operation is optimistic and does not rely no locks. However, we want to replicate the same behaviour as with PGrid, and not waist transactional attempts on trying to delete the first bucket when queries may be constantly entering. Hence, we lower the number of attempts to 1 whenever a deletion of the first bucket is required (Line 13). This ensures that if transaction fails, the fall-back path is acquired and any further readers are blocked from entering the cell, allowing for the fast removal of the first bucket and avoiding wasting further transactional attempts.

Moreover, we make the distinction between the transactional path and the fall-back path in the "waitUntilNoReaders" function. The "waitUntilNoReaders" function loops over the nR of the first bucket until it becomes 0. When running transactionally, any modification to this variable (by a concurrent transaction) will deterministically force an abort of the current transaction, hence, we avoid this by immediately aborting the transaction. This way, we avoid the time spent waiting for an abort. In contrast, when running with the SGL, the thread may wait until the nR becomes 0 and finally free the first bucket.

The insert algorithm (Algorithm 16) is equal to the PGrid's one. The only difference is again in the way writes are made to the primary index, as previously explained.

The range query algorithm (Algorithm 17) is the other interface operation allowed by the indexes. The pseudo code for this algorithm maintains the same as PGrid's. However, changes are performed in the *PpCellScan(cell)* function contained in it.

Algorithm 14: update(ObjectTuple new)

```
newCell = computeCell(new.x, new.y) ;
TM_BEGIN_SGL( ) ;                                    // begin transaction
sie = SecondaryIndex.lookup(new.oid);
oldCell = sie.cell;
obj = getObj(sie.bckt, sie.idx) ;                    // object tuple
if hasLD(sie) then
    delete(sie) ;                                    // physical deletion
end
if newCell == oldCell then
    // local update
    if fallback then
        OLFIT_write(new, obj) ;                      // atomic OLFIT write
    else
        *obj = new ;                                 // new copied over object
    end
else
    // Nonlocal update
    sie.ldCell = sie.cell;
    sie.ldBckt = sie.bckt;
    sie.ldIdx = sie.idx;
    insert(new, newCell, sie) ;                      // physical insertion
    obj.tu = -new.tu ;                               // mark as logically deleted
end
TM_END_SGL( ) ;                                      // end transaction
```

Algorithm 18 introduces the changes made to the range query algorithm. The critical section used in PGrid to obtain a reference to the first bucket of the cell and increment the nR is replaced with an HTM transaction (Lines 2-12). We recall that this increment ensures the first bucket of the cell is safe for reading, also in this case, performed via simple accesses to memory from a transactional context and using atomic operations from within the fall-back path. Recall that an updater may need to stop queries from entering a cell, in order to perform a deletion of the first bucket. It is in this critical area where queries are stopped until the updater is finished. In order to achieve this synchronization between update and query operations, this transaction subscribes the SGL. This guarantees that if some update operation detects the need to garbage collect a bucket, by activating the fall-back it will block any concurrent query (by aborting it and forcing to wait for the SGL to become free).

As for the scanning of the bucket, in PGrid this is performed in a lock-free fashion, by resorting to OLFIT to ensure correct reads. In fact, since a bucket is scanned by a query only after the nR has been incremented, that bucket is guaranteed not to be garbage-collected while the reader is scanning. Also, the freshness semantics (along with the use of logically deleted versions) allows to observe any object stored in the bucket as long as it reflects atomically the updates of concurrent update operations.

As mentioned, we have here an opportunity for replacing the OLFIT mechanism with transactions, to perform a consistent scanning of the bucket. There is one important observation that can be done here, though: the transaction used during the scanning phase can avoid subscribing the SGL, as it can safely run with any concurrent update operation executing in the fall-back patch, which by using OLFIT writes would cause the corresponding transaction to abort anyway. Also, if the transaction encompassing

Algorithm 15: delete(SIEntry sie)

```
ldObj = sie.ldBckt.entries[sie.ldIdx];
firstBckt = *sie.ldCell ;                                  // cell refers to the 1st bucket
lastObj = firstBckt.entries[firstBckt.nO - 1];
if fallback then
    OLFIT_write(lastObj, ldObj) ;                          // atomic OLFIT write
else
    *ldObj = *lastObj ;                                    // lastObj copied over ldObj
end
firstBckt.nO– ;                                            // decrement number of objects
if firstBckt == 0 then
    *sie.ldCell = firstBckt.nxt ;                          // No more queries can enter firstBckt
    if attempts $>$ 1 then
        attempts = 1 ;       // ensures no further deterministic failure attempts occur
    end
    if fallback then
        waitUntilNoReadersLock() ;                         // wait and proceed
    else
        waitUntilNoReadersTM() ;                           // if nR $>$ 0; abort transaction
    end
    free(firstBckt);
end
Nullify all ld references in sie;
Lookup for lastObj's sie and update it;
```

Algorithm 16: insert(ObjectTuple new, Cell cell, SIEntry sie)

```
firstBckt = *cell ;                                        // cell refers to the 1st bucket
if $isFull$(firstBckt) then
    Allocate new bucket and make it first;
end
freePos = firstBckt.entries[firstBckt.nO];
if fallback then
    OLFIT_write(new, freePos) ;                            // atomic OLFIT write
else
    *freePos = new ;                                       // new written to freePos
end
sie.cell = cell;
sie.bckt = firstBckt;
sie.idx = firstBckt.nO;
firstBckt.nO++ ;                                           // increment number of objects
```

```
Algorithm 17: rangeQuery(Rect q, int ts)
res = 0 ;                                       // container for storing the result
cells = computeCoveredCells(q);
for cell ∈ cells do
    objects = pCellScan(cell);
    for obj ∈ objects do
        if obj.tu > 0 then
            res.addAndOverwrite(obj);
        else if abs(obj.tu) > ts then
            res.addIfNoSuch(obj);
    end
end
Similar processing is performed for partially covered cells;
return res;
```

```
Algorithm 18: pCellScan(Cell cell)
objects = 0 ;                                   // container for storing the result
TM_BEGIN_SGL() ;                                // being transaction
if isEmpty(cell) then
    return objects;
end
bckt = *cell ;                                  // cell refers to the 1st bucket
if fallback then
    aInc(bckt.nR) ;                             // CAS increment of number of readers
else
    bckt.nR++ ;                                 // increment number of readers
end
TM_END_SGL() ;                                  // end transaction
while bckt != null do
    TM_BEGIN_NB_TX() ;                          // begin transaction without subscribing the SGL
    if fallback then
        for idx := bckt.nO - 1 ; To 0 ; Step -1 do
            obj = readObj(bckt.entries[idx]) ;  // atomic OLFIT read
            objects.add(obj);
        end
        if bckt.nxt ! = null then
            aInc(bckt.nxt→nR) ;                 // atomic CAS increment
        end
        aDec(bckt.nR) ;                         // atomic CAS decrement
        bckt = bckt.nxt;
    else
        for idx := bckt.nO - 1 ; To 0 ; Step -1 do
            obj = bkct.entries[idx] ;           // plain memory read
            objects.add(obj);
        end
        if bckt.nxt ! = null then
            bckt.nxt→nR++;                      // non-atomic increment
        end
        bckt.nR– ;                              // non-atomic decrement
        bckt = bckt.nxt;
    end
    TM_END_NB_TX() ;                            // end transaction
end
return objects;
```

the scanning phase exhausts its available attempts and has to use the fall-back path, it can use the original OLFIT read mechanism, which is guaranteed to ensure atomic reads of objects in presence of concurrent updates executing either in the fallback path (i.e., using OLFIT writes) or in transactions.

In the light of this considerations, the transactions that we use to replace OLFIT reads do not subscribe any lock. This allows these transactions to avoid blocking/aborting in case the SGL fallback path is activate. Because of this, we call these transactions, non-blockable transactions (nb-transactions). The fallback path of nb-transactions consists of the original PGrid's OLFIT read logic, and continues to be executed in a lock-free way.

In this pseudo code we partition the transaction with the length of a while loop, which is equivalent to an entire bucket (64 objects, 1024 bytes). However, we are able to partition transactions with any desired length, from a single object per transaction up to x buckets per transaction. This provides us various transactional sizes to experiment, which influence the contention and the memory footprint of transactions.

The while is divided between the fall-back and the transactional path (respectively Lines 15-25, 25-35). When in the fall-back path we use atomic operations to read the bucket entries (Line 17) and to increment/decrement the nR (Lines 21, 23). In contrast, a transaction is already atomic, hence, we simply read the bucket entries (Line 27) and increment/decrement nR (Lines 31, 33).

3.3.2 PGridHTM-FGL

The HLE mechanism of TSX can already be used to transparently elide the locks of a program and substitute them with transactions. When transactional attempts expire, transactions acquire the elided locks. However, HLE is an automatic process, which disregards some important optimizations, which we are able to perform with RTM, by explicitly controlling lock elision via the RTM's interface of TSX. For example, PGrid's algorithms require acquiring multiple locks to support operations' execution. In an HLE approach, each level of locks would be substituted by a begin_htm_tx(), which would result in several nested transactions. Even though HTM disregards nesting, in the sense that all transactions are treated as a flat transaction, where if a inner transaction aborts it is rolled back to the most outer one. There is a cost in opening/closing transactions, which we are able to avoid using RTM. Moreover, since our goal is to achieve the best throughput, every optimization is essential.

In the following, we describe the changes brought by the fine-grained locking fall-back system. To note, all previous locks used in PGrid are maintained, transactional handlers now read the specific locks needed and use them as fall-back path, in case HTM attempts expire. Note that transactions are safe to proceed if the lock is unlocked, since any modification to the state of the lock by a concurrent thread will result in the abort of the transaction.

Moreover, there are two types of transactional handlers: the outer (Algo. 25 and 26) and the inner handlers (Algo. 27, 28, and 29). The outer handlers are the only ones which actually begin/end a transaction and initialize the variables used to control the code flow, e.g., the *fallback* variable. In contrast, the inner handlers are not required to do so as they are already inside the critical section. Hence, we

Algorithm 19: update(ObjectTuple new)

```
1  newCell = computeCell(new.x, new.y);
2  objLock = getObjLock(new.oid);
3  TM_BEGIN_OUTER_FGL(objLock) ;                           // begin transaction
4  sie = SecondaryIndex.lookup(new.oid);
5  oldCell = sie.cell;
6  obj = getObj(sie.bckt, sie.idx) ;                           // object tuple
7  if hasLD(sie) then
       // physical deletion
8      if !delete(sie) then
9          TM_END_INNER_FGL(objLock);
10         go to 3;
11     end
12 end
13 if newCell == oldCell then
       // local update
14     if fallback then
15         OLFIT_write(new, obj) ;                          // atomic OLFIT write
16     else
17         *obj = new ;                                 // new copied over object
18     end
19 else
       // Nonlocal update
20     sie.ldCell = sie.cell;
21     sie.ldBckt = sie.bckt;
22     sie.ldIdx = sie.idx;
23     insert(new, newCell, sie) ;                         // physical insertion
24     obj.tu = -new.tu ;                               // mark as logically deleted
25 end
26 TM_END_OUTER_FGL(objLock) ;                               // end transaction
```

use them to read the locks, and to decide whether to follow the HTM or the fine-grained locking path, in regards to the control flow variables.

The update algorithm, Algorithm 19, reads the lock of the object to be updated in order to properly react to its state (Line 3). In order to ensure no deadlocks, as it occurs in PGrid, if the deletion of a logically deleted object returns false (Line 8), then we must unlock all previous locks and restart the update operation. Hence, we are respecting the fine-grained locking system found in PGrid. This The update algorithm ends with the commit of the transaction or the release of the updated objects' lock (Line 26).

Algorithm 20: delete(SIEntry sie)

```
 1 cellLock = getCellLock(sie.ldCell);
 2 TM_BEGIN_INNER_FGL(cellLock) ;                                // begin transaction
 3 oldObj = sie.ldBckt.entries[sie.ldIdx];
 4 firstBckt = *sie.ldCell ;                         // cell refers to the 1st bucket
 5 lastObj = firstBckt.entries[firstBckt.nO - 1];
 6 lastObjLock = getObjLock(lastObj.oid);
 7 if TM_BEGIN_INNER_DEADLOCK_FGL(lastObjLock) then
 8     if fallback then
 9         OLFIT_write(lastObj, ldObj) ;                          // atomic OLFIT write
10     else
11         *ldObj = *lastObj ;                             // lastObj copied over ldObj
12     end
13     firstBckt.nO– ;                                   // decrement number of objects
14     if firstBckt == 0 then
15         *sie.ldCell = firstBckt.nxt ;             // No more queries can enter firstBckt
16         if attempts > 1 then
17             attempts = 1 ;   // ensures no further deterministic failure attempts occur
18         end
19         if fallback then
20             waitUntilNoReadersLock() ;                              // wait and proceed
21         else
22             waitUntilNoReadersTM() ;                     // if nR > 0; abort transaction
23         end
24         free(firstBckt);
25     end
26     Nullify all ld references in sie;
27     Lookup for lastObj's sie and update it;
28     TM_END_INNER_FGL(lastObjLock);
29     TM_END_INNER_FGL(cellLock);
30     return true;
31 else
32     TM_END_INNER_FGL(cellLock);
33     return false;
34 end
```

The delete algorithm, Algorithm 20, also reads the locks in order to properly react to their state. However, it also ensures there are no dead locks when in the fall-back path, by returning false in case the lastObj lock is not able to be acquired (Line 7, 32-33). Recall that the lastObj is used to overwrite the logically deleted object, performing its physically deletion. The major advantage of using the FGL version in the delete algorithm is that, in the fall back path, we only lock the specific cell where the logically object object resides, hence, only the queries that want to read this cell are blocked. Moreover, only the last object of the first bucket is locked, which is needed to perform the overwrite operation (Line 7). At the end of the algorithm, depending whether we are running transactionally or in the fall-back path, we respectively proceed without committing, as we are inside a transaction already, or we unlock the cell and lastObj locks (Line 28,29).

Algorithm 21, is the insert algorithm, which is protected by a inner handler. Therefore, the specific cellLock is read and no transactions are started or committed. In case we are running in the fall-back path, the cellLock is locked and unlocked (Line 2, 17).

Finally, we describe the changes made to the the pCellScan algorithm, Algorithm 22. Recall that the scanning of cells must ensure the first bucket is not deleted/modified before the scan starts (Line 3-13). With PGridHTM-SGL version, when we were running non-transactionally, we used the SGL to perform the block of concurrent updates. In contrast, the FGL version only uses the specific cellLock to to block concurrent updates which target the same cell, hence, facilitating parallelism. Finally, the scan of the cells is still made with nb-transactions.

Algorithm 21: insert(ObjectTuple new, Cell cell, SIEntry sie)

```
1  cellLock = getCellLock(cell);
2  TM_BEGIN_INNER_FGL(cellLock);
3  firstBckt = *cell ;                                  // cell refers to the 1st bucket
4  if $isFull$(firstBckt) then
5      Allocate new bucket and make it first;
6  end
7  freePos = firstBckt.entries[firstBckt.nO];
8  if fallback then
9      OLFIT_write(new, freePos) ;                      // atomic OLFIT write
10 else
11     *freePos = new ;                                 // new written to freePos
12 end
13 sie.cell = cell;
14 sie.bckt = firstBckt;
15 sie.idx = firstBckt.nO;
16 firstBckt.nO++ ;                                     // increment number of objects
17 TM_END_INNER_FGL(cellLock);
```

Algorithm 22: pCellScan(Cell cell)

```
objects = 0 ;                                        // container for storing the result
cellLock = getCellLock(cell);
TM_BEGIN_OUTER_FGL(cellLock) ;                       // being transaction
if isEmpty(cell) then
    return objects;
end
bckt = *cell ;                                       // cell refers to the 1st bucket
if fallback then
    aInc(bckt.nR) ;                                  // CAS increment of number of readers
else
    bckt.nR++ ;                                      // plain increment of number of readers
end
TM_END_OUTER_FGL(cellLock) ;                         // end transaction
while bckt != null do
    TM_BEGIN_NB_TX() ;                               // begin non-blockable transaction
    if fallback then
        for idx := bckt.nO - 1 ; To 0 ; Step -1 do
            obj = readObj(bckt.entries[idx]) ;       // atomic OLFIT read
            objects.add(obj);
        end
        if bckt.nxt ! = null then
            aInc(bckt.nxt→nR) ;                      // atomic CAS increment
        end
        aDec(bckt.nR) ;                              // atomic CAS decrement
        bckt = bckt.nxt;
    else
        for idx := bckt.nO - 1 ; To 0 ; Step -1 do
            obj = bckt.entries[idx] ;                // plain memory read
            objects.add(obj);
        end
        if bckt.nxt ! = null then
            bckt.nxt→nR++;                           // non-atomic increment
        end
        bckt.nR– ;                                   // non-atomic decrement
        bckt = bckt.nxt;
    end
    TM_END_NB_TX() ;                                 // end transaction
end
return objects;
```

Chapter 4

Evaluation

In this chapter we conduct an extensive study aimed to quantitatively assess the performance of the proposed HTM-based algorithms and compare them with state of the art single-threaded and multi-threaded (lock-based) solutions. In our study we will evaluate the HTM implementations available in three different architectures (Haswell and Broadwell by Intel and IBM Power8) using a test bed composed by 4 primary workloads. We conduct a preliminary study aimed at identifying the optimal tuning of several relevant parameters affecting HTMs performance, in particular, retry count, choice of the memory allocator, size of query transactions and thread pinning. Next, equipped with the knowledge on how to tune the proposed HTM-based indexing algorithms, we seek to identify the workloads where HTM implementations are advantageous in comparison with the state of the art indexes.

4.1 Description of the workloads

Update and query traces are used to create workloads with different configurations. Table 4.1 shows the configuration setup of our workload. We enforce these workload specifications used in [5] in order to have some fixed point of view in comparison with their results. Our workloads are composed of 1 million objects with 10 timestamps, resulting in 10 million updates (Fig. 4.1). Workload density (# objects / km^2) is enough to create sufficient contention and transactions to provide a proper evaluation to the indexes. We use Germany as the monitored area, which was constructed grouping all shapefiles of German cities. We specifically use all German roads as our (Brinkhoff [36]) network where objects may move through. Our workload generator (MOTO) has the ability of creating hotspots in cities, simulating reality.

The workload parameters we target to evaluate are the range query size and the update/query ratio. The update/query ratio is the fundamental balance of the workload when targeting HTM study. The other parameter we target is range query size. As we can see in Figure 4.2, range query size has a big impact on the area covered by queries. This is an image of two query traces each with 40k range queries, representing the query ranges of, respectively, 4km and 250m. The difference in area coverage is considerable, despite only changing the range query size. As reviewed in section 2.1.1.1,

Parameter	Values
Objects (x 10^6)	1
Updates (x 10^6)	10
Monitored Region (km^2)	Germany, 641 x 864
# road network segments	32,750,494
# road network nodes	28,933,679
Speed (km/h)	20, 30, 40, 50, 60, 90
Update/query ratio	**1:1**, ...1:10, ...250:1, ...1000:1, ...**16000:1**
Range query size (km^2)	**0.25**, 1, 4, **16**

Table 4.1: Workload Configuration

HTM struggles with exceeding memory capacity. By increasing/decreasing the range query size we are also increasing/decreasing the number of objects that are swiped in a range query. Thus, we aim to see how this affects HTM in terms of conflict and capacity aborts, due to the higher number of objects needed to be transactionally read.

Finally, the configurations of the base workloads used to tune HTM (Section 4.3) and to perform the final evaluation (Section 4.5) are represented in bold (see Table 4.1). We specifically use opposing parameters to ensure our study encompasses a larger variety of workload configurations, and hence, our experiments are suited to a broader range of workloads.

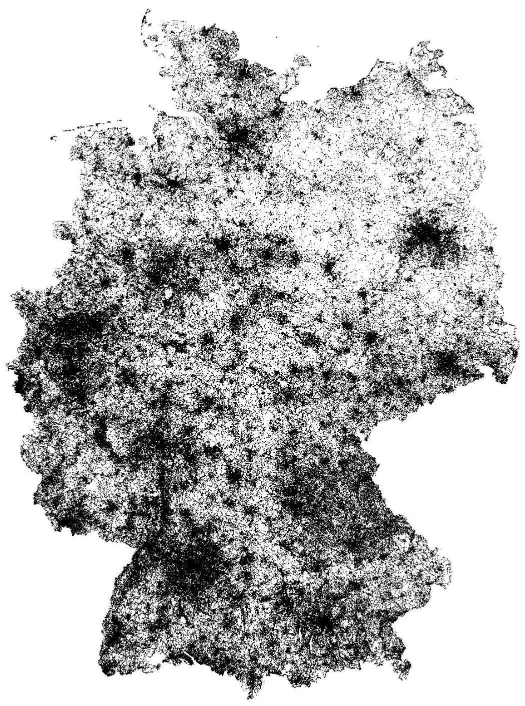
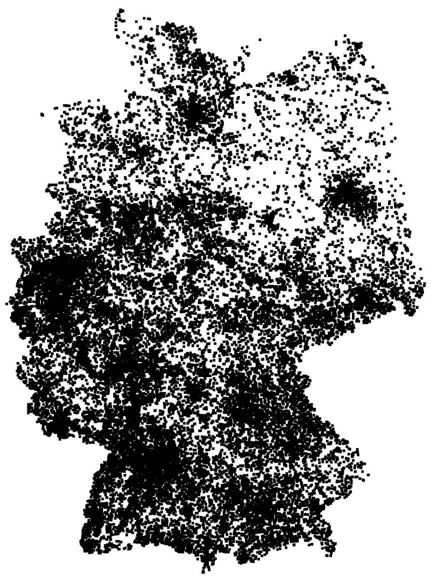
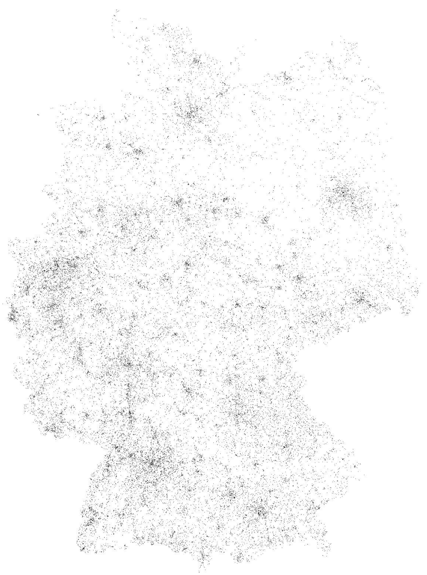


Figure 4.1: MOTO Germany update trace file with 1 million objects and 10 timestamps, total of 10M updates



(a) Query range 4km



(b) Query range 250m

Figure 4.2: MOTO Germany query trace files with 40k queries and variable query ranges

4.2 Platforms and Metrics

Our evaluation is split among 3 different platforms:

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 [45]). 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 [45]). 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.

Both Haswell and Broadwell share similar (Intel) TSX implementations, specifically regarding their transactional capacity and cache line granularity (see Table 3.1). Hence, we expect similar behaviours since HTM environments are also similar. In contrast, POWER8 has a 128 bytes cache line granularity, whereas Intel only has 64 bytes. This makes POWER8 more subjective to false conflicts, as more memory is loaded into cache, further increasing the possibility of invalidating other cache lines. Moreover, POWER8s transactional capacity is very small (8KB). Hence, we expect POWER8 to be the platform which delivers the worst HTM environment.

Haswell is a single-socket 4 core CPU with hyper-threading technology [45], which allows each core to execute 2 hardware threads. Hence, we evenly distribute threads until all cores have at least one thread. Then, we activate hyper-threading distributing the remaining threads evenly by all cores. In the end, each core is running 2 hardware threads, hence, reaching 8 threads in total.

Simultaneous multi-threading (SMT), also know as hyper-threading in Intel, is a technology which may significantly improve systems performance, however, it also has some drawbacks. The main idea of SMT is to have parallel threads in the same core, increasing performance with instruction level scheduling, which grants the possibility of having more instructions in each cpu cycle. Instructions come from the several threads the core has attached to it, and hence, each cpu cycle may have several instructions from parallel threads. Nevertheless, there are drawbacks in this technique. Each thread has to share the cores' resources, in regards to HTM, the main concern is whether there is enough cache to fit each threads' transactional meta-data. Moreover, as resources are shared (including bandwidth) it means that threads run slower than they would do if they had exclusive use of the processors' core. However, in most cases, the combined throughput of the threads is greater than the throughput of either one of them running exclusively [46].

The Broadwell and POWER8 machines are instead dual socket machines. In these machines, one has the possibility to opt between two main thread-pinning [1] alternatives.

[1]Thread pinning allows the programmer to choose how threads are attributed to the different CPU cores.

In both alternatives, we start by first assigning threads to different physical cores of the same socket. Once we have one thread per core of the first socket, we are faced with a choice: the next thread can be assigned to a different socket or to the same socket. We call the former multi-socket-first (MS-first) or SMT-first.

In order to conduct a proper evaluation of our experiments we use 3 types of metrics/parameters:

Performance Metrics:

1. **Throughput** - Measures the overall performance of the system. It measures the number of transactions committed per second.
2. **Breakdown of commits** - Percentage of committed transactions running HTM or that have used the fall-back path mechanism. In case of HRWLE, we also consider which fraction of transactions committed/aborted using ROTs, and read only transactions.
3. **Breakdown of aborts** - Percentage of aborted transactions, split by reason why they aborted.

Configuration parameters:

1. **Transactional Retries** - Number of times a transaction may abort until it acquires the fall-back path.
2. **Thread Pinning** - Whether threads are pinned with MS-first or SMT-first.
3. **Memory Allocation** - Experiments running with either TCMalloc [16] or the Glibc memory allocator.

Index parameters

1. **Query Transaction Size** - Size of transactions in query cell scanning.

Workload parameters:

1. **Update/query ratio** - Ratio between read and write transactions.
2. **Query range** - Area covered by a range query.

All experiments where made with -O3 optimization level and 64 bits instruction size. The reported results represent the average of 8 runs.

We summarize our findings in order to gain insights on the advantages and drawbacks of each architecture. In Section 4.5, we make a final comparison between all the architectures.

4.3 Tuning of HTM

We tune HTM in regards with four different parameters: Number of transactional retries, number of objects read in query transactions (only for PGridHTM), memory allocators and thread pinning. For each parameter we search empirically for the optimal value, which as, we will see, is often architecture and index dependent. These parameters have dependencies between each other, hence, they are not trivial to tune. To make a proper tuning we follow this specific order: First, we tune the number of transactional retries until we spot the default optimal number. Next, we use the best performing configuration of the transactional retries and assess the choice of the memory allocator. Second, we evaluate which pinning technique has best performance. Third, we tune the number of objects read in a query (only for PGridHTM), which we found to have a non-negligible impact on the memory allocation performance for the case of PGridHTM (but not for u-GridHTM). As such, for PGridHTM we evaluate the tuning of these two parameters in conjunction.

Note that the experiments performed with PGridHTM, where performed with both versions of PGridHTM (SGL and FGL) and with the OLFIT non-blockable fall-back path. However, neither fall-back path affects the results, hence, we refer to the index solely by its first name, PGridHTM.

4.3.1 Transactional Retries

Transactional retries are the number of attempts a transaction can try to commit before recurring to the fall-back path. A higher value of transactional retries may allow HTM to provide more robust performance in contention prone scenarios, which trigger HTM to abort with higher probability. However, this can also have repercussions as we are delaying HTM to acquire the fall-back path. Hence, deterministic aborts take longer to follow the fall-back path, increasing the time they are exposed, where they may be conflicting with other transactions.

4.3.1.1 Haswell

Figure 4.3 reports the plots used to set the transactional retry number for both u-GridHTM and PGridHTM. We only use the query intensive workload scenarios as these are where the major performance differences, due to the retry number, occur.

Starting with u-GridHTM, it is clear that the best performance is achieved using the minimum transactional retries value, 5. Unfortunately, u-GridHTM has serious contention and capacity struggles and, as you can see from the abort breakdown of these plots in Figure 4.6, the abort percentage rounds about 70% to 90%. Due to these issues, it is actually best to reduce the transaction retry value and allow the fall-back path to be acquired more often. Most of the aborts are capacity aborts, which are deterministic and will only generate further contention.

In contrast, PGridHTM is only slightly affected by the different values of the transactional retry number. The breakdown of aborts illustrates a lack of HTM contention and memory capacity overflow. Instead, most aborts are related with other causes as system calls or operating system interrupts. Due to the low abort count, transactional retries have negligible affect on performance. Hence, also for PGridHTM we will use the same retries count value, namely 5.

With the optimal transactional retry value defined, we may focus on the different memory allocators for u-GridHTM (no dependency with other parameters). As shown in Figure 4.5, u-GridHTMs performance is greatly improved when using TCMalloc in query intensive workload scenarios (1u:1q). In contrast, update intensive workload scenarios show similar performance between the two memory allocators. The abort breakdown (Fig. 4.6), shows that the abort percentage is similar between both memory allocators. However, a difference can be spotted in the abort reasons, TCMalloc has a higher conflict abort count, whereas Glibc has a higher capacity abort count. One can suggest that due to the deterministic nature of the higher count of capacity aborts found in Glibc, these will never commit and will force the acquisition of the fall-back path.

4.3.1.2 Broadwell

Figure 4.7 contains the plots used to define the optimal transactional retry value for u-GridHTM and PGridHTM. As previously discussed, u-GridHTM is not scalable in query intensive workload scenarios. Hence, its best performance is at the 14 thread count, a full socket working with hyper-threading. At this point, 10 is the transactional retry value which achieve grater throughput. However, in terms of scalability, 20 is the best transactional retry value (56 thread mark). Moreover, we further tested u-GridHTMs retry value in update intensive workload scenarios (plots not displayed). In these experiments, the best retry value was clearly 20, in terms of scalability and throughput. u-GridHTM is not competitive in query intensive workload scenarios, hence, we further value performance at update intensive workloads. Concluding, the transactional value chosen was 20, as it has the greater scalability in both workloads and the higher throughput in the favoured, update intensive workloads scenarios.

Next, we evaluate the optimal retry value for PGridHTM. A low amount of retries (5) hinders scalability in query intensive workload scenarios. Adding additional retries reduces the amount of aborts, as we can see in Figure 4.8, which contrasts with u-GridHTM. Therefore, the optimal retry value for PGridHTM is 20.

Finally, we perform an evaluation to the different memory allocators in u-GridHTM. The results in Figure 4.9 show us that the different memory allocators have similar performances. In query intensive workload scenarios, the traditional Glibc memory allocator reaches further performance, nevertheless, at the maximum thread mark its performance is similar to TCMalloc's. The breakdown of aborts (Fig. 4.10) can help us select the better allocator. As we may conclude from the figure, there is a higher count of conflict aborts with TCMalloc in the query intensive workloads, which also occurred in Haswell (Fig. 4.6). Previously, TCMalloc was able to achieve better throughput, now however, performance is not improved.

In update intensive workload scenarios, the performance of the two allocators is very similar. In the following we will consider Glibc for Broadwell, since TCMalloc was not able to improve performance and raised the conflict abort count.

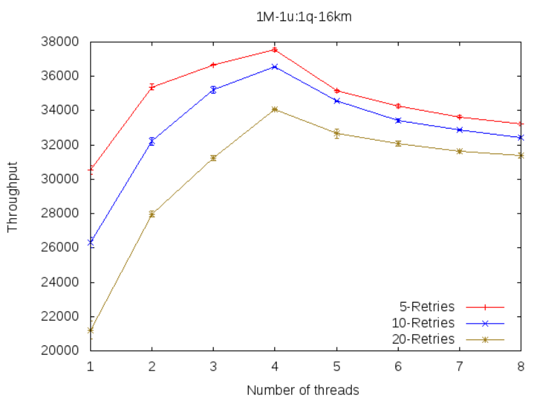
4.3.1.3 POWER8

With POWER8, we choose to evaluate the transactional retry number with update intensive workload scenarios, where we most clearly can see the differences imposed by the transactional retry number.

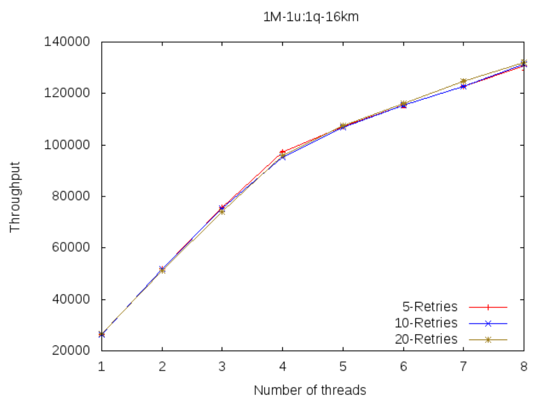
Figure 4.11 has the plots used to evaluate both indexes (u-GridHTM and PGridHTM). A low retry value in POWER8 drastically hinders performance. A loop is created where eventually almost all updates acquire the fall-back path. As we can denote from Figure 4.12, the minimum amount of retries for u-GridHTM and PGridHTM is respectively, 20 and 10. By increasing the retry value, we allow transactions to fail more times, before acquiring the fall-back path. Hence, the best performing retry number for both indexes is 20.

In Figure 4.13 we perform an evaluation to u-GridHTMs performance when applying the different memory allocators. Respectively, the abort breakdown can be found in Figure 4.14.

In query intensive workload scenarios, TCMalloc is clearly the best memory allocator, maintaining always a slightly superior performance over Glibc. Similarly, in update intensive scenarios, TCMalloc can maintain better throughput throughout most of the thread spectrum. However, at the maximum thread count, Glibc seems to achieve a slightly better performance. Nevertheless, the variance between both indexes indicates that throughputs can be quite similar. Finally, we opt for TCMalloc as the best memory allocator for u-GridHTM since it has the best average performance in both workloads.



(a) u-GridHTM



(b) PGridHTM

Figure 4.3: Haswell: Defining the optimal transactional retry number

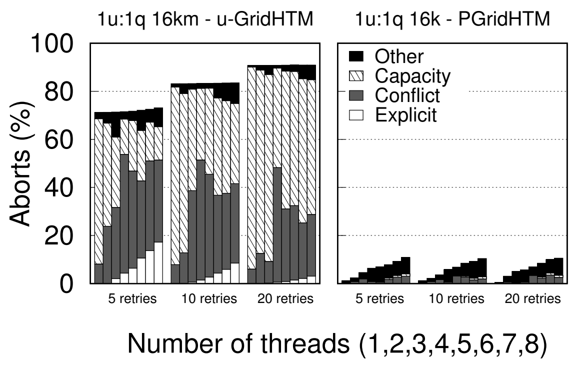


Figure 4.4: Haswell: Transactional retries abort breakdown

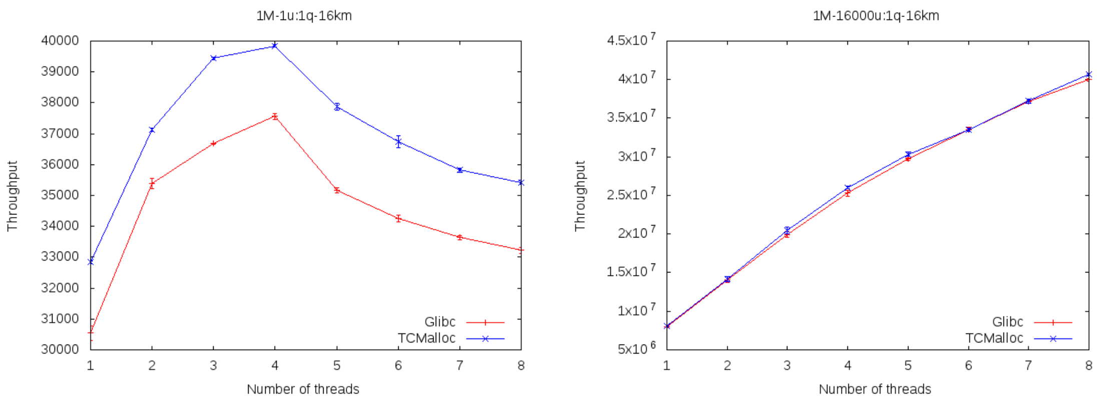


Figure 4.5: Haswell: Defining the best memory allocator for u-GridHTM

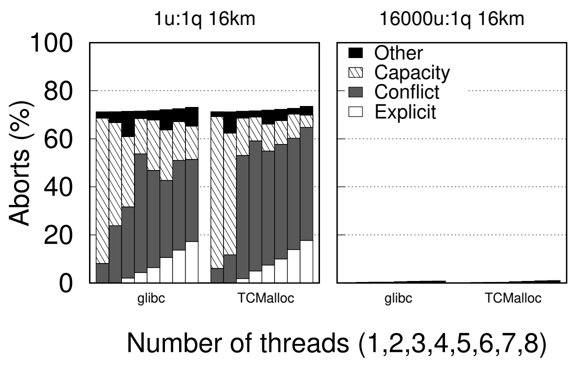
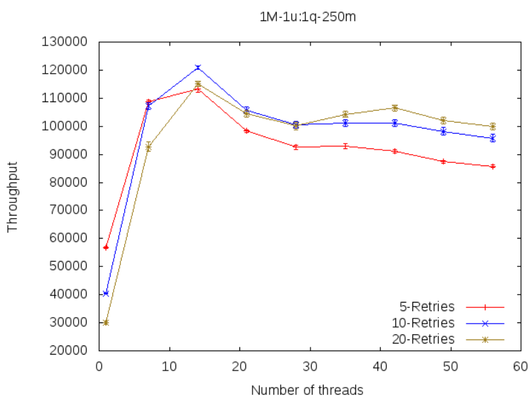
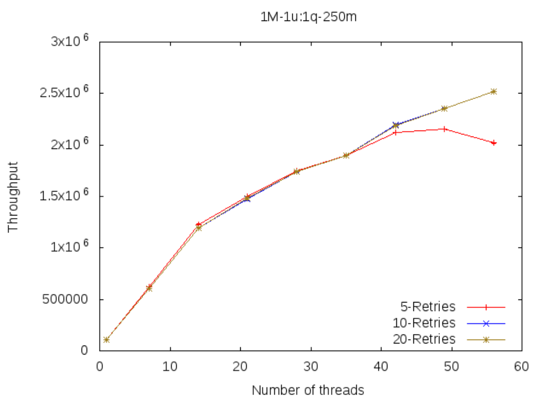


Figure 4.6: Haswell: u-GridHTM memory allocators abort breakdown



(a) u-GridHTM



(b) PGridHTM

Figure 4.7: Broadwell: Defining the optimal transactional retry number

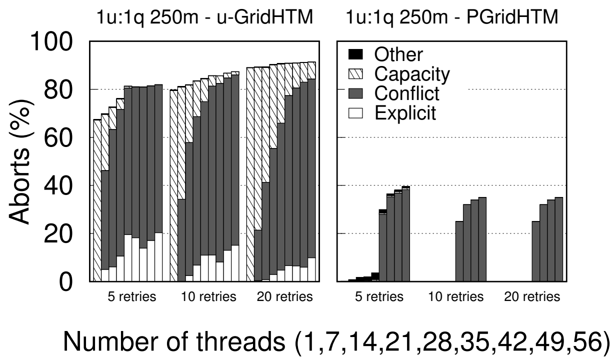
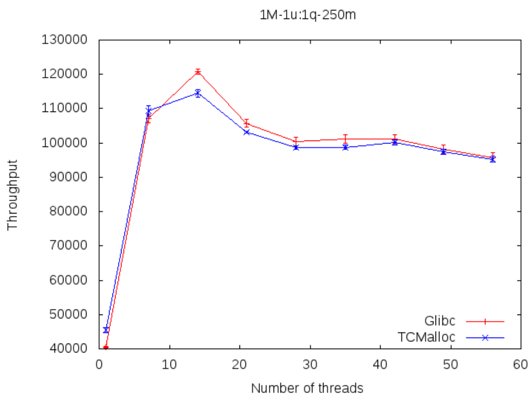


Figure 4.8: Broadwell: Transactional retries abort breakdown



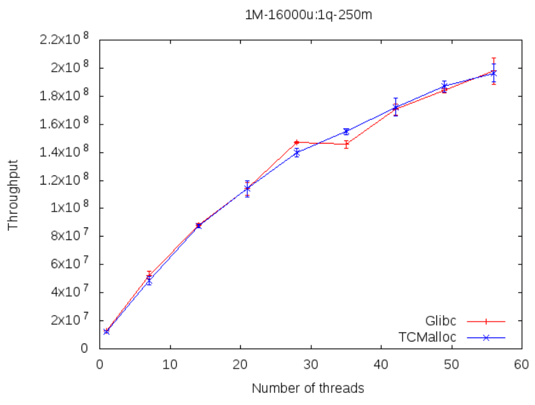


Figure 4.9: Broadwell: Defining the best memory allocator for u-GridHTM

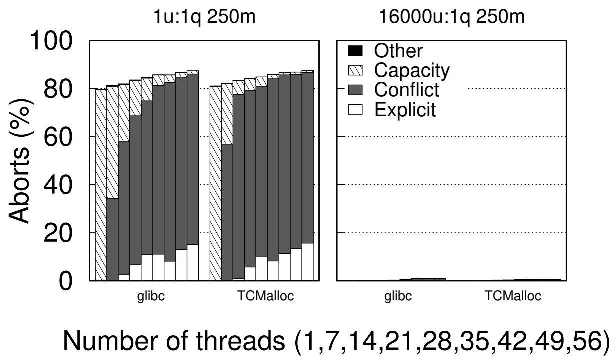
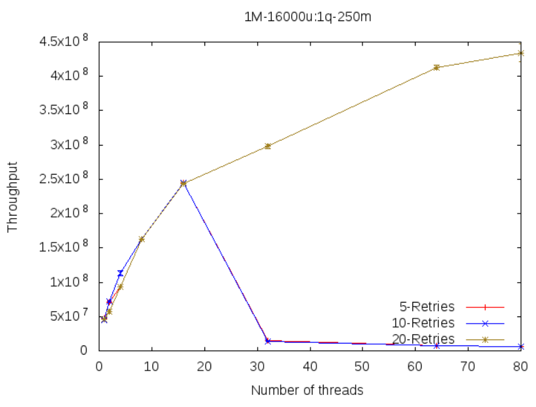
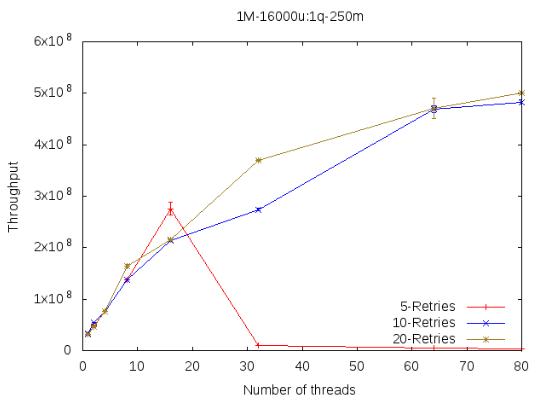


Figure 4.10: Broadwell: u-GridHTM memory allocators abort breakdown



(a) u-GridHTM



(b) PGridHTM

Figure 4.11: POWER8: Defining the optimal transactional retry number

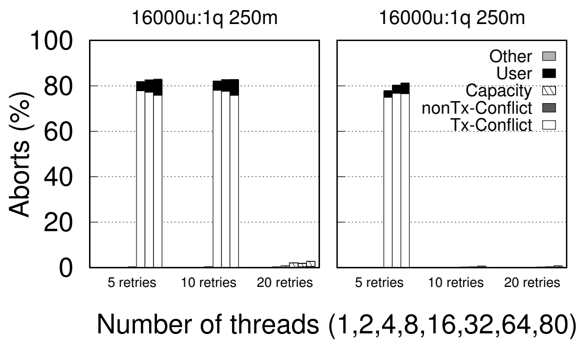


Figure 4.12: POWER8: Transactional retries abort breakdown

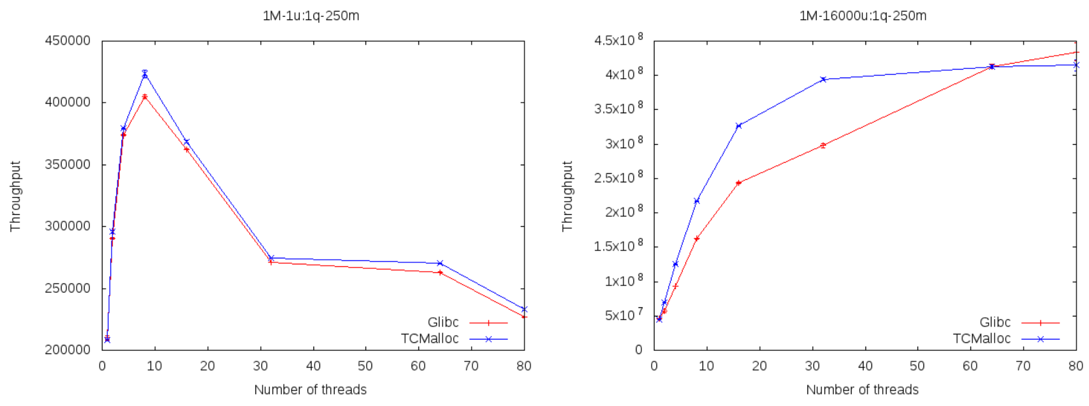


Figure 4.13: POWER8: Defining the best memory allocator for u-GridHTM

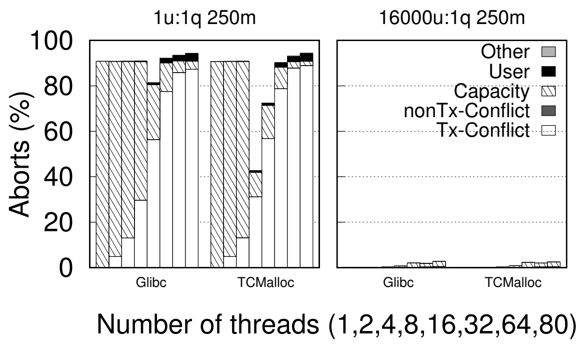


Figure 4.14: POWER8: u-GridHTM memory allocators abort breakdown

4.3.2 Thread pinning

In this section we evaluate the different pinning techniques in regards to their performance. As previously mentioned, these techniques are SMT-first and MS-first. Recall that the Haswell architecture does not contain a second socket, hence, it is not possible to perform this experiment with it.

4.3.2.1 Broadwell

Broadwell activates hyper-threading past the 14 and 28 thread mark, respectively with the SMT-first and MS-first pinning techniques. Figures 4.15 and 4.16 include the plots and respective abort breakdown for PGridHTM.

In update intensive workload scenarios, the MS-first pinning technique achieves the overall best throughput. Despite having a higher abort rate at lower thread counts, MS-first can still maintain a higher throughput over SMT-first. Note that a loss on throughput can be clearly seen after the 14 thread mark (hyper-threading activated) and again at the 28 thread mark (second socket hyper-threading). We conclude that the extra capacity aborts cause by using hyper-threading, is the cause for the SMT-first drop on performance after the hyper-threading thread marks. Indeed, as already discussed, to share the same physical core increases the cache of incurring capacity aborts.

In contrast, in query intensive workload scenarios, the SMT-first pinning technique achieves the overall best throughput. The differences, anyhow, do not appear to be dramatic, and, therefore, in the following we will opt for STM-first as the default thread pinning strategy when using Broadwell.

4.3.2.2 POWER8

POWER8 activates SMT past the 5 and 10 thread mark, respectively with the SMT-first and MS-first pinning techniques. Figures 4.17 and 4.18 include the plots and respective abort breakdown for PGridHTM.

In both workload scenarios, the MS-first pinning technique is the pinning technique, which achieves better throughput throughout the entire spectrum of the thread count. In the query intensive workload, SMT-first encounters the same capacity related issues seen in Broadwell (Fig. 4.15). POWER8 ships with a much smaller capacity for HTM transactions compared with the Intel processors, and supports a much higher SMT-level (8x vs 2x). The net result is that, when SMT is fully enable in POWER8, the available capacity for each hardware threads narrows down significantly. This causes a sharp increase in the probability of aborts and, eventually, in the frequency of acquisition of the pessimistic fall-back path. In contrast, update intensive workloads only suffer slightly from exceeding memory capacity. Due to the obtained results, we choose MS-first as the default pinning technique in POWER8.

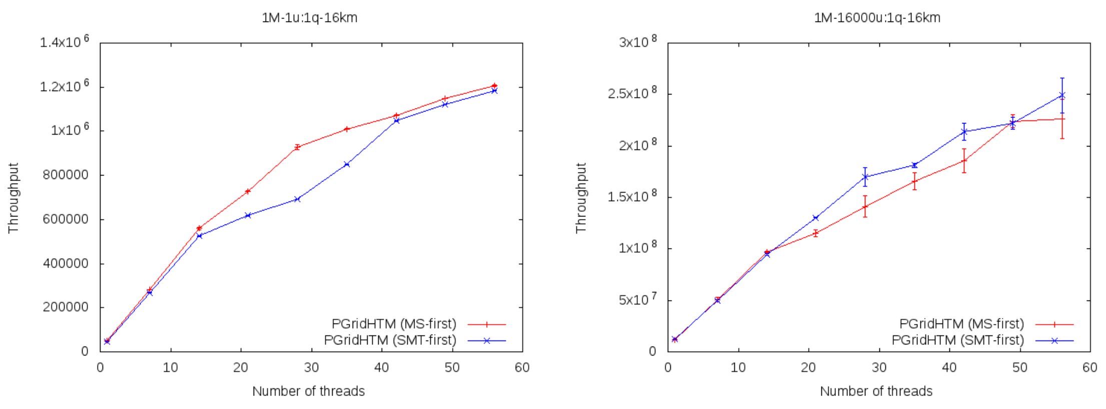


Figure 4.15: Broadwell: Pinning techniques comparison

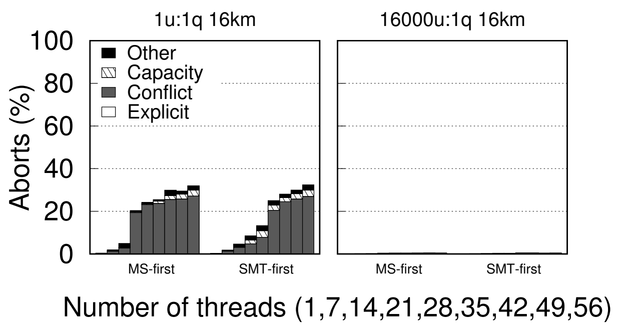


Figure 4.16: Broadwell: Pinning techniques abort breakdown

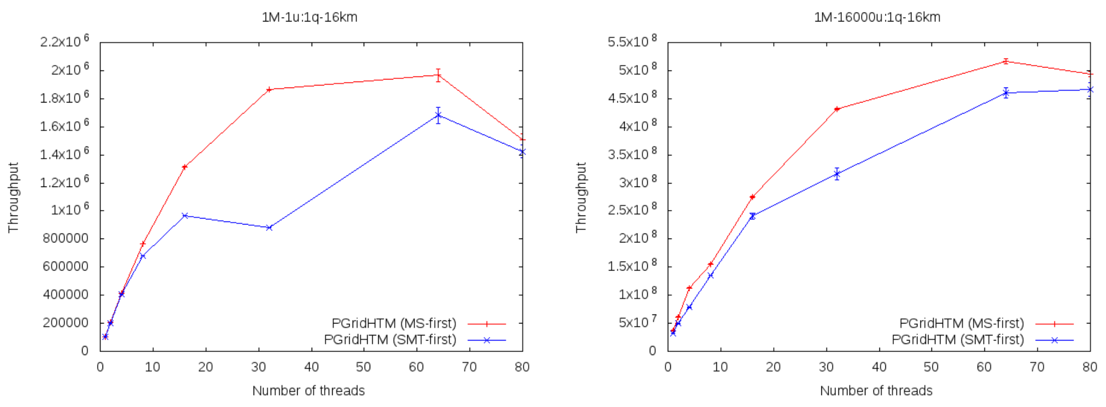


Figure 4.17: POWER8: Pinning techniques Comparison

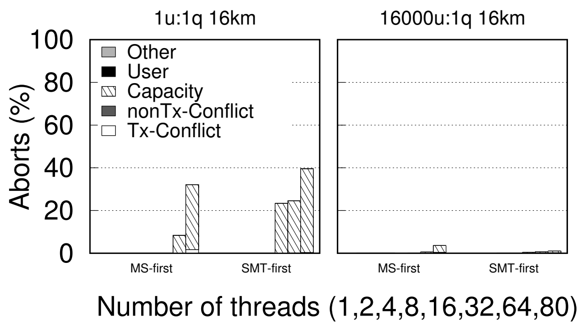


Figure 4.18: POWER8: Pinning techniques abort breakdown

4.3.3 Transactional Partitioned Queries

Transactional partitioned queries are only used in PGridHTM, which partitions queries by multiple transactions in order to reduce contention between updates and queries, and to avoid exceeding transactional capacity (see Section 3.1). Transactional partitioned queries acquire the non-blockable fall-back path (i.e., OLFIT) in case they are not able to commit a respective number of times. The non-blockable fall-back path may also be permanently used, in case its performance is better than the use of transactions.

In this section we vary the number of objects read within transactions in the bucket scanning phase and empirically assess the corresponding impact on performance. Recall that the maximum amount of objects in a bucket is 64 (1024 bytes).

4.3.3.1 Haswell

Figure 4.19 reports the experimental results obtained while varying the transaction's size when using, on the left, the Glibc memory allocator and, on the right, TCMalloc. Let us start by analysing the results using Glibc.

Our studies reveal that the optimal transaction size is workload dependent. In query intensive workloads the best performance is achieved with 1 and 2 buckets transaction size. With range queries of 250 meters, the 2 bucket transaction size is the best performing. However, increasing the range query size to 16km makes so that the 2 bucket transaction size encounters further conflicts and capacity aborts (Fig. 4.21). Hence, the smaller 1 bucket sized transaction is better suited for large query sizes.

In contrast, update intensive workloads have best performance with smaller transaction sizes, even though performance differences are smaller, due to the low amount of transactions in these workloads scenarios. This is imputable to the speed at which updates are processed with the lack of query interference. Hence, when a query transaction is performed, there are an abundant number of concurrent update transactions, which easily conflict with the query transaction.

For both update intensive workloads, the optimal transaction size is 32 elements per bucket or the usage of atomic operations as OLFIT, instead of transactions. This occurs since the higher number of concurrent updates forces a higher contention with queries. Hence, a lower transaction size avoids further conflicts with the high number of updates. Abort breakdown is negligible as there are too few queries.

The use of TCMalloc usually improves the overall performance, however, it restricts the query transaction size. This is better visualized in query intensive workloads, see the 16km query size plot. The overall performance is increased by TCMalloc, with respect to Glibc. However, the 2 bucket size transactions suffer a huge drop on performance. Similarly, the 250m query size plot improves its overall performance. However, the best transaction size is no longer 2 buckets, instead, the single bucket transaction is the only one which prevails.

The same happens for update intensive workloads, however, performance changes are almost negligible due to the already fast updating speed. Transactional query size, however, is still affected by

TCMalloc. In both workloads, the 1 and 2 bucket size transactions are slightly affected.

4.3.3.2 Broadwell

The plots are again organized depending on the memory allocator (Fig. 4.20), and we first do the analysis of the plots with Glibc and only then with TCMalloc. With Broadwell we expanded the limit of the transaction size to 3 buckets since this architecture had promising results with only 2 buckets. The abort breakdown of these plots is in Figure 4.22.

Starting with query intensive workloads, 2 buckets is the ideal transaction size for both 250m and 16km range query sizes. Interestingly, the abort rates descend from 1 bucket to 2 bucket transaction sizes. This occurs due to the fact that this is a query intensive workload, where contention between updates and queries is is relatively modest. We also tested the 3 bucket sized transactions in order to see if there was even further potential. However, transactions this big start originating capacity issues and slow down the overall throughput.

In update intensive workloads the results are too similar to distinguish the best transaction size. Despite that, we define again 2 buckets as the transaction size. What we can retrieve is that HTM is generally faster than using OLFIT.

When adding TCMalloc to query intensive workloads, there is a surge of conflicts when threads start to be active on both sockets ($>$28 threads), and throughput lowers drastically. To avoid this, the transaction size must be set as low as possible. In the 250m example, the 32 and 16 elements transaction sizes avoid such a sharp drop of performance and attain good scalability. However, by decrementing so much the transaction size, the final throughput is worst than running with the Glibc memory allocator. Bigger transaction sizes generate too much conflicts to have a good performance (Fig. 4.22).

With update intensive workloads the results are similar. However, we still achieve better results with the Glibc memory allocator. Moreover, the same happens at a smaller scale, as the 2 and 3 bucket sized transactions throughput is worse, due to the conflicts TCMalloc imposes to HTM.

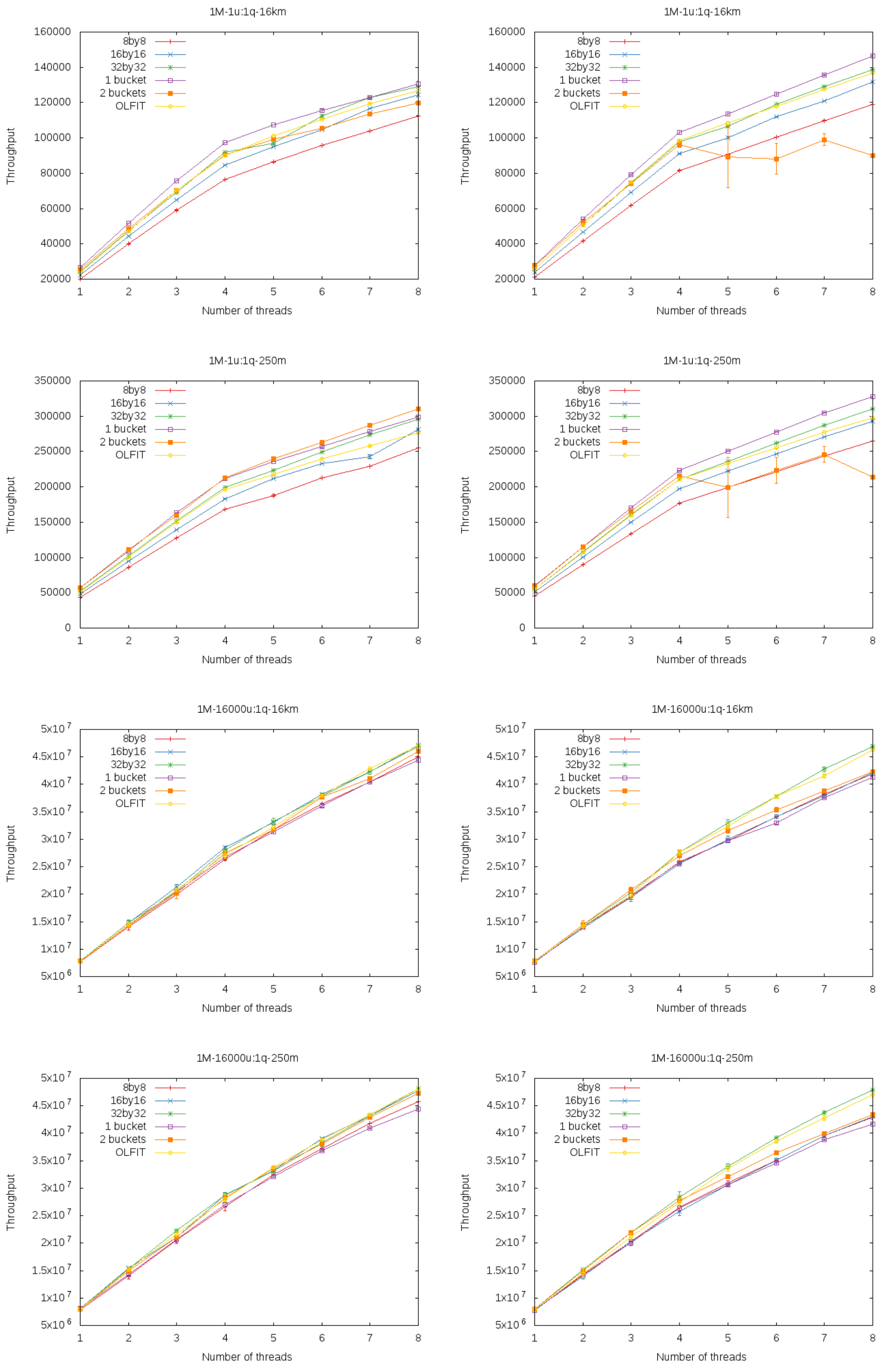


Figure 4.19: Haswell: Defining the optimal transaction size in PGridHTM. Left - Glibc : Right - TCMalloc

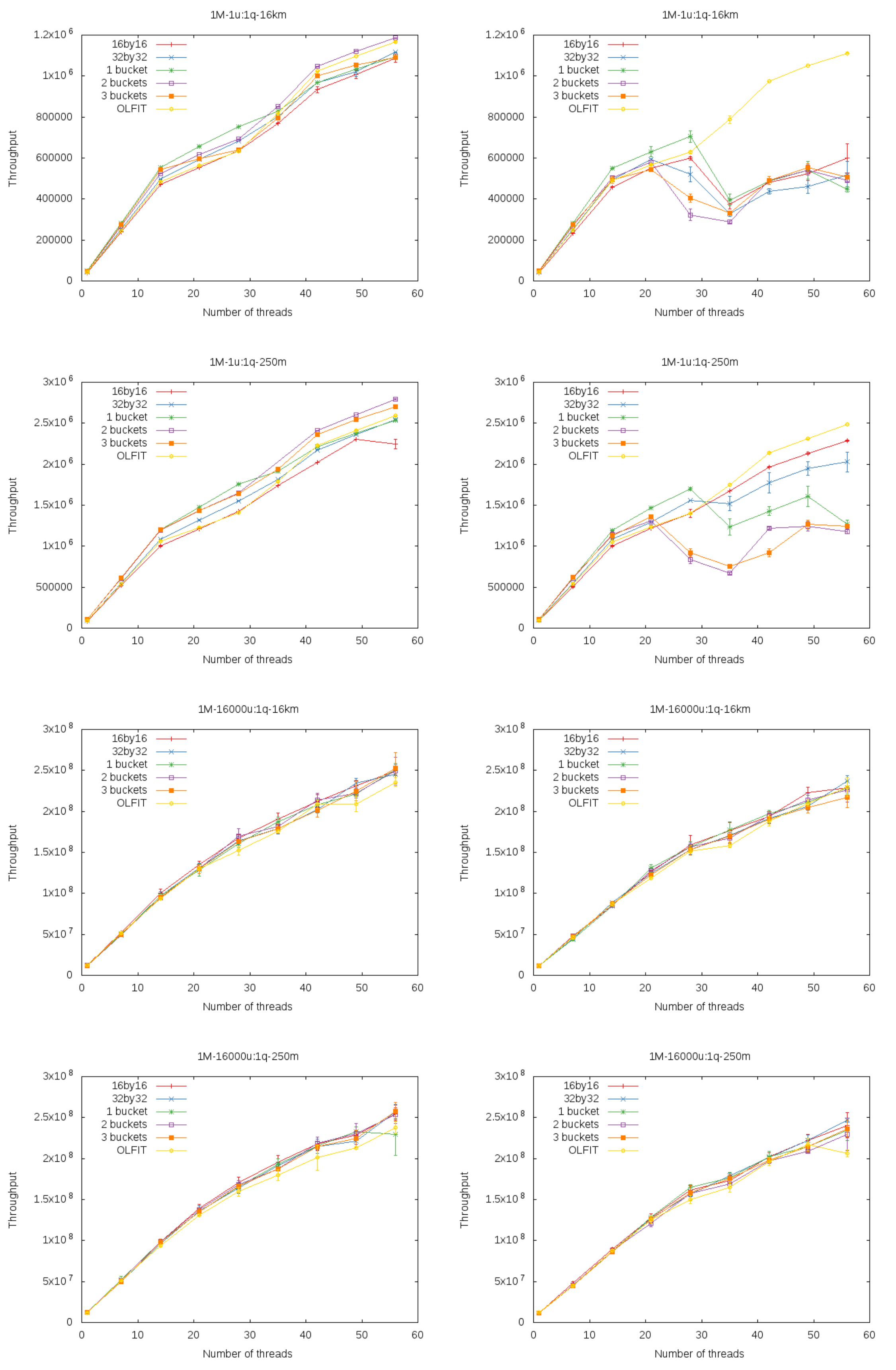


Figure 4.20: Broadwell: Defining the optimal transaction size in PGridHTM. Left - Glibc : Right - TCMalloc

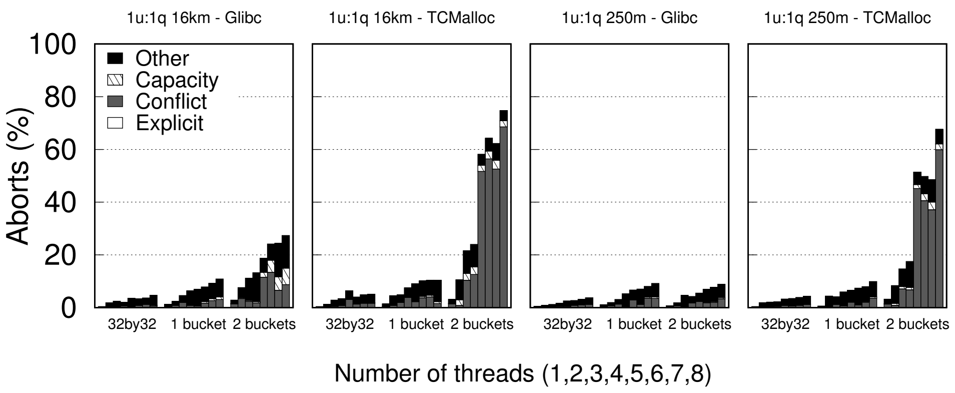


Figure 4.21: Haswell: Abort breakdown in PGridHTM. Glibc vs TCMalloc

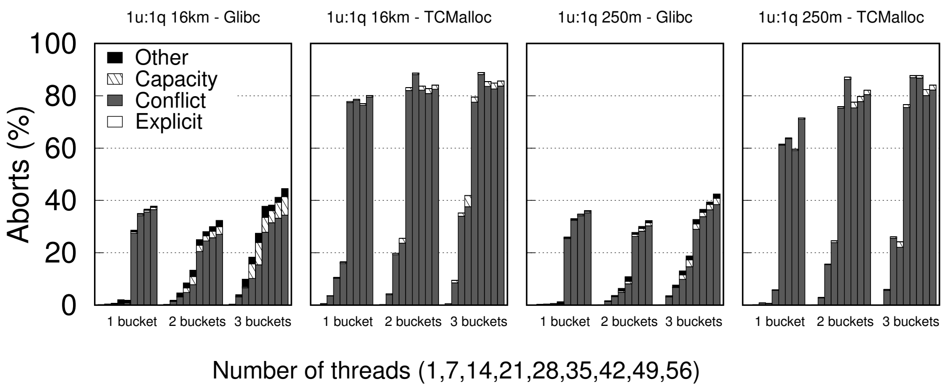


Figure 4.22: Broadwell: Abort breakdown in PGridHTM. Glibc vs TCMalloc

4.3.3.3 POWER8

Performance shifts occur due to the SMT-first pinning technique used in these experiments. When STM is activated performance drops, in contrast, when MS is used performance raises.

The partitioning of transactions in POWER8 is even more important due to its small L2 TMCAM cache size and its higher SMT-level. For query intensive workloads (Fig. 4.23), we found that OLFIT-only query scanning achieves 50% speedups. Using transactions to perform queries in POWER8 is too big of a burden. Transactions are restricted to small sizes in order not to exceed memory capacity (see 4.24), which forces the closer/opening of multiple more transactions, hindering performance. This trade-off is not worth it, hence, the OLFIT-only solution achieves the best performance.

Update intensive workloads show analogous trends. The OLFIT-only query scan ensures best overall results. Even though capacity aborts are negligible in these workloads, they still occur to the few queries performed. Moreover, conflicts are prone to happen due to the higher contention between updates and queries. Hence, the OLFIT-only query scan is able to achieve 5% speedups.

The TCMalloc tests for P8 are not present since their performance in query intensive is very poor. The low memory capacity and high cache granularity, generated up to 90% capacity aborts. Moreover, in update intensive workloads, TCMalloc also performed worst.

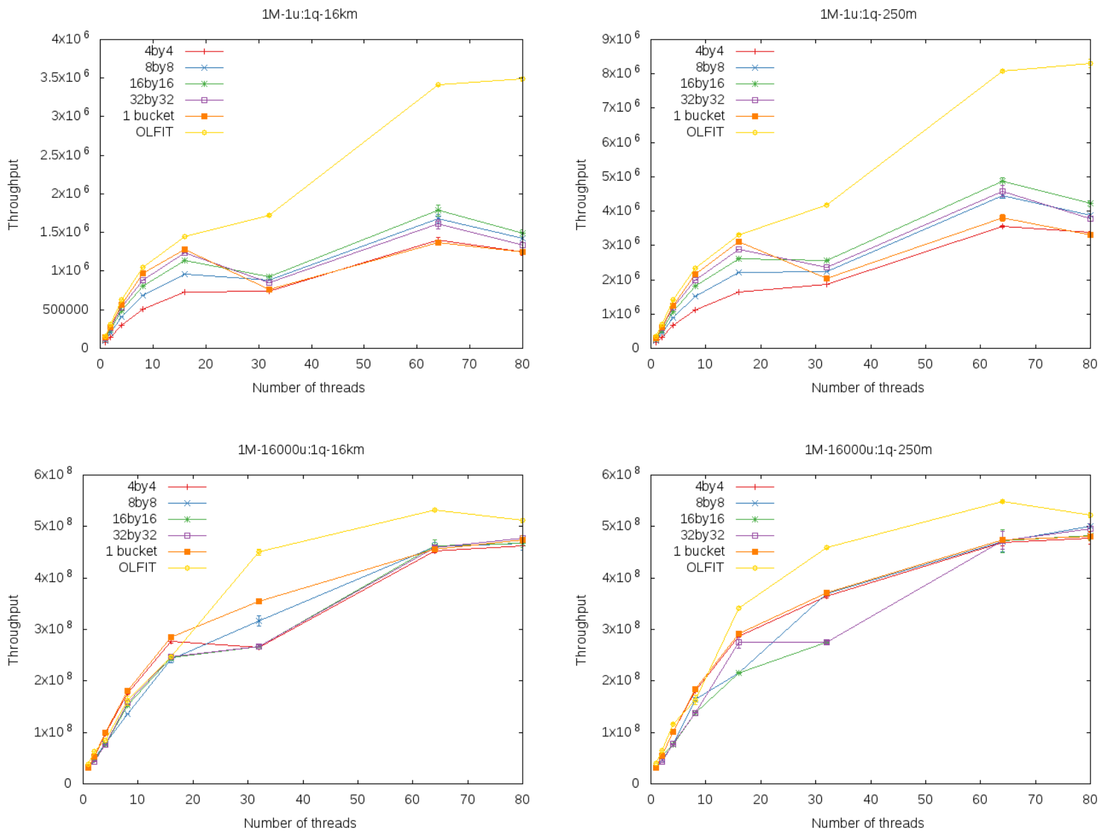


Figure 4.23: POWER8: Defining the optimal transaction size in PGridHTM

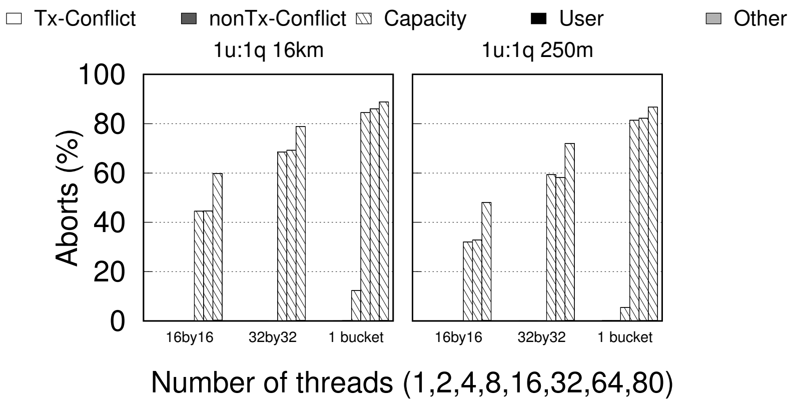


Figure 4.24: POWER8: Abort breakdown in PGridHTM.

4.3.4 Tuning Results

We summarize our HTM tuning results in order to have a general view of how each index and corresponding platform are tuned. We make such depiction in Table 4.2. To note, we split query transaction sizes between query intensive workloads (QIW) and update intensive workloads (UIW). Our results show us that the number of transactional retries is strongly depended to the platform used. Moreover, we also conclude that the higher the thread count of the platform, the higher is the need to increment the number of retries. The higher the thread, the higher the possibility of conflicts between threads, which ultimately resort to the fall-back path. Hence, increasing the retry number generally yields more robust HTM performances.

In terms of the partitioned transaction query size, we conclude that larger transactions are better for query intensive workloads. In contrast, update intensive workloads, a smaller transaction size avoids the contention generated by numerous concurrent updates. However, depending on the workload, a proper tuning must be made to retrieve the optimal transaction retry value. Finally, Intel processors, which as already mentioned ensure larger transactional capacities, favour the use of larger bucket sizes, when compare to POWER8.

As for the choice of the memory allocator, we observe that, TCMalloc is usually good when combined with u-GridHTM, except for the Broadwell's case. In contrast, Glibc is usually better when combined with PGridHTM. Most significantly, TCMalloc seems to increase performance at the cost of a superior conflict count. Hence, when applied to a platform with a low thread count (Haswell), it is able to improve performance. However, when used in platforms supporting a larger degree of hardware parallelism, the conflict count is too high, ultimately decreasing performance.

Finally, we also tested the optimal memory allocator for the non-HTM indexes. Even though no single memory allocator is best in all cases, the most consistent one which brought the overall better results is Glibc, in all architectures. These plots can be found in Appendix B.

Index	Architecture	Number of Retries	Query Transaction Size (QIW)	Query Transaction Size (UIW)	Memory Alloc
u-GridHTM	Haswell	5	–	–	TCMalloc
u-GridHTM	Broadwell	20	–	–	Glibc
u-GridHTM	POWER8	20	–	–	TCMalloc
PGridHTM	Haswell	5	1 bucket	32 elements	TCMalloc
PGridHTM	Broadwell	20	2 buckets	2 buckets	Glibc
PGridHTM	POWER8	20	OLFIT	OLFIT	Glibc

Table 4.2: HTM indexes tuned parameters

4.4 Performance Evaluation

In this section we compare the performance of the HTM-based indexes presented in Chapter 3 with various state of the art solutions. This evaluation is aimed to gain insights on which concurrency mechanism is better suited for spatio-temporal indexes, and to understand the impact of the workload's and platform's characteristics on the considered solutions. The evaluation is done using the parameters' tuning derived in the previous section, in order to guarantee the representativeness of the observed performance results. To note, we performed experiments with PGridHTM using OLFIT atomic operations as the non-blockable fall-back path, namely PGridOHTM.

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.

4.4.1 Haswell

In query intensive workloads (top plots of Fig. 4.25), 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.

Both PGrid versions (SIMD/OLFIT) have a scalable performance. Similarly, both versions of PGridOHTM (SGL/FGL) are scalable and able to achieve 17.5% speedups with respect to PGrid's both versions. Interestingly, the SGL version is able to reach better performance in comparison with FGL. This occurs due to the the low frequency of aborts incurred by these HTM-based indexes (see Fig. 4.26), which results in the infrequent activation of the fall-back path. Hence, the extra instrumentation require to subscribe multiple fine grained locks during transaction's execution, in PGridOHTM-FGL, introduces overheads that are not outweighed by the additional degree of concurrency achievable in case of fall-back activation.

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 (reviewed Sec. 2.3.2.1). 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 the need of wrapping the entire update and query functions in single transactions.

Update intensive workloads originate completely different results (bottom plots of Fig. 4.26). 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

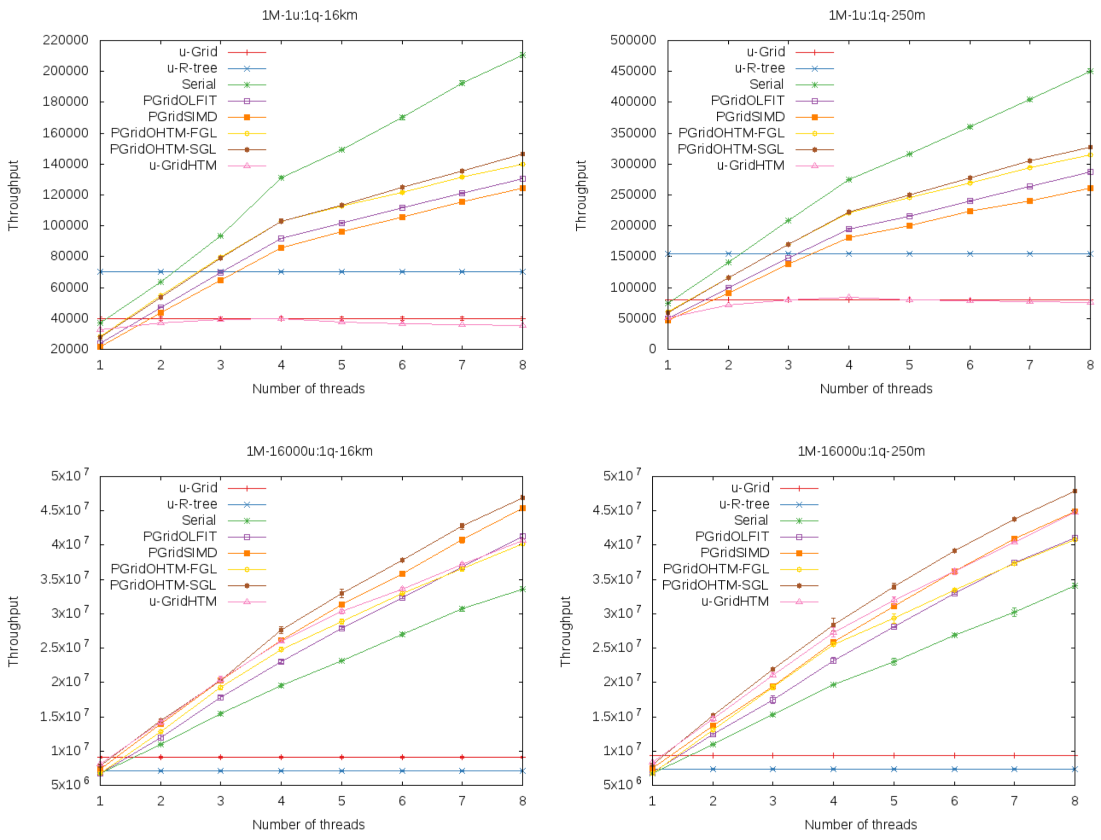


Figure 4.25: Haswell: All indexes performance evaluation

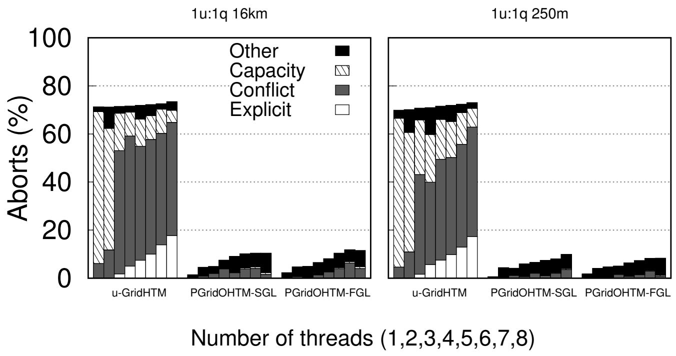


Figure 4.26: Haswell: Abort Breakdown of HTM indexes

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 single 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.

4.4.2 Broadwell

Figure 4.27 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 4.28 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 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 out-perform 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,

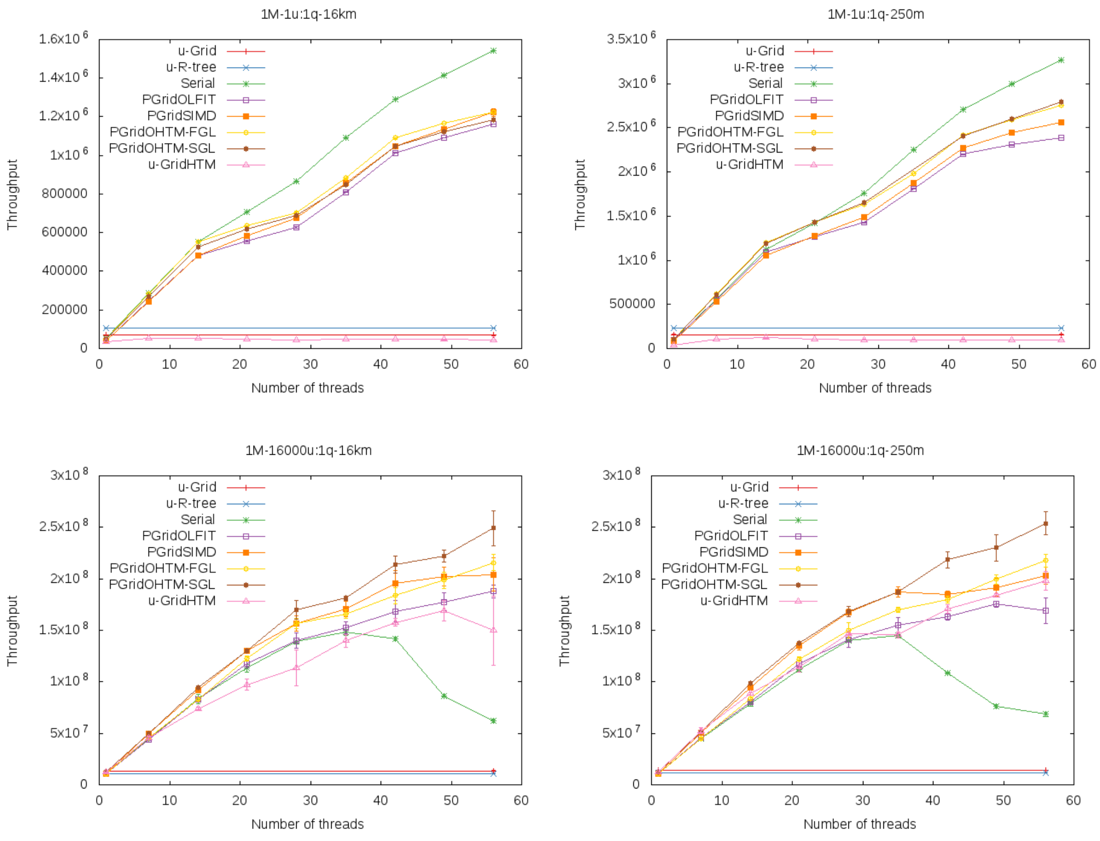


Figure 4.27: Broadwell: All indexes performance evaluation

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.

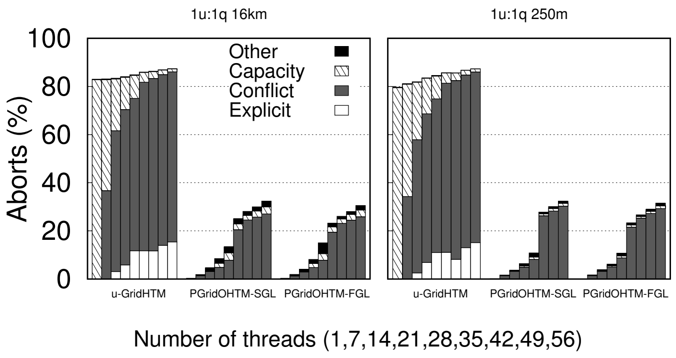


Figure 4.28: Broadwell: Abort Breakdown of HTM indexes

4.4.3 POWER8

Performance regarding HTM indexes is expected to be the most affected in POWER8, due to its more constrained HTM environment. Figure 4.29 reports the performance of the considered indexes in this architecture.

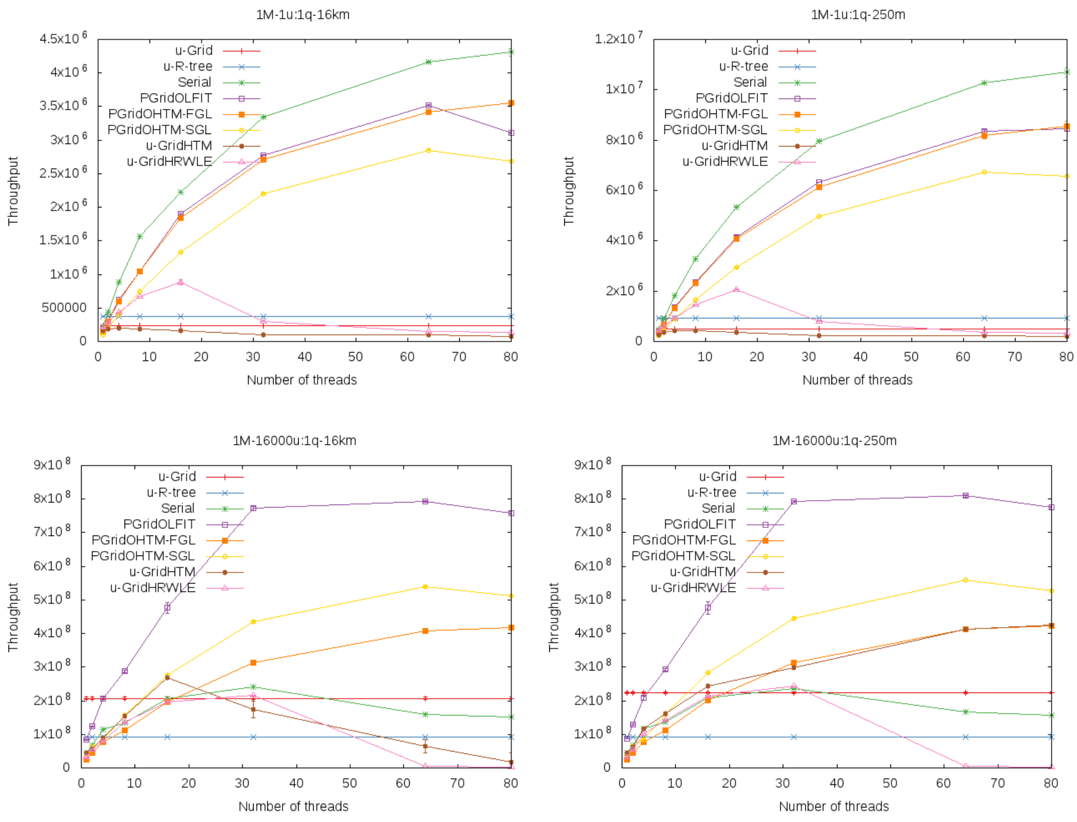


Figure 4.29: POWER8: All indexes performance evaluation

In query intensive workloads (top plots of Fig.4.29), 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 4.30, both versions 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 capac-

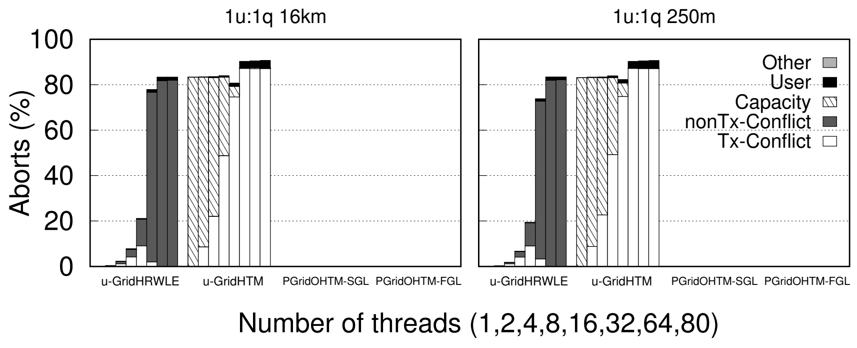


Figure 4.30: POWER8: Abort Breakdown of HTM indexes

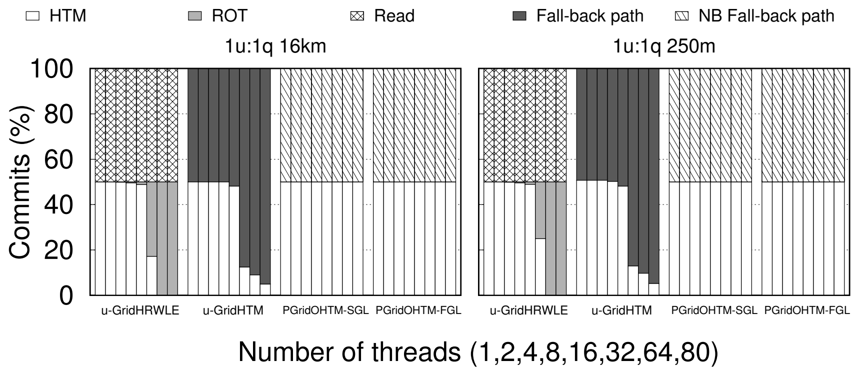


Figure 4.31: POWER8: Commit Breakdown of HTM indexes

ity 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. We include the commit breakdown in POWER8 (see Fig. 4.31) to deepen our comprehension on the reasons behind u-GridHRWLE's poor performance. As we can see, at the 32 thread mark, updates are mostly committing using the serializable ROTs. Hence, contention is forcing ROT's to be predominant, which drastically hinders performance. 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.

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.

4.5 Experimental Summary

We conclude our evaluation chapter with a summary of the obtained results in the extensive experimental study.

Architectural considerations

- From an architectural perspective, POWER8 is the platform which achieves the higher throughput. Nevertheless, the use of update intensive workloads hinders the most HTM performance: even though aborts are negligible in these kind of workloads, PGridHTM is not able to reach performance comparable with PGrid. Conversely, on the Intel's platforms, PGridHTM outperforms the complex lock based algorithm, PGrid.

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?

- The conclusions we reached suggest that, on the one hand the speculative approach used by HTM has good potential - we have shown that it is indeed possible, in the considered application context, to achieve performance competitive with highly efficient fine grained locking scheme at a fraction of the complexity.

  On the other hand, the ability to concretize such potential is strongly dependant on whether the limitations (especially in terms of capacity) of existing HTM implementations meet the workload characteristics.

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?

- HTM can in fact be useful to boost optimized lock-based algorithms: Up to 40% gain vs PGrid. An interesting consideration is that, in optimized synchronization schemes that use fine-grained locking, like PGrid, locks are typically maintained for a very short time period and, as such, access a relatively few number of items. When using HTM to elide these locks, the probability of success of running in hardware are normally quite high. Conversely, in Power8, PGridHTM was able to surpass PGrid's performance in update intensive workloads.

  By applying HTM in PGrid we were able to achieve the following extra gains:

  1. Eliding the locks, which provides optimistic updates and queries, most significantly, optimistic insertion/deletion of objects in cells, and optimistic subscription into the cell, performed by queries.
  2. Concurrency between updates and queries, whether running transactionally or in the fall-back path. Provided by the use of transactions in conjunctions with OLFIT atomic operations.
  3. Transactional Partitioned Queries, which provides optimistic scans to buckets in cells, and avoids extra costs as OLFIT (OLFIT only used as fall-back path). Moreover, the size of these

transactions can be pre-defined in order to avoid contention and exceeding transactional capacity.

FGL vs SGL fall-back paths

- Another, somewhat surprising, finding emerged from the results gathered in this work is related to the performance of fine grained vs single global lock fall-back paths: approaches based on a single global lock have most frequently outperformed, in our study, approaches that acquire, the originally elided fine grained locks. Despite the latter can theoretically achieve a higher degree of parallelism than the former, it also imposes additional instrumentations in the critical path of execution of transactions. Our results have shown that these instrumentation costs end up, in most of the considered workloads, outweigh the potential for higher concurrency of fine grained lock -based fall-back paths. The exception to this being both 1u:1q-16km/250m workloads in POWER8, we argue that POWER8s constrained HTM environment forces additional transactions to acquire the fall-back path, hence, despite the additional overheads, the fine-grained locking fall-back path still achieves higher performance.

Serial

- Interestingly, the index with best performance in query intensive workloads is always Serial. This index has lightweight instrumentation and achieves best performance when hotspots do not occur, which is exactly what occurs with query intensive workload scenarios. In future work, it would be interesting to design and evaluate mechanisms based on HTM to accelerate Serial. — in a way analogous to what was done in this work for PGrid.

HRWLE

- HRWLE did not have an easy run with the workloads scenarios used. Certainly, if we considered scenarios with a superior read ratio, its performance would greatly improve. Most significantly, HRWLE does not solve contention between updates and queries, hence, conflicts between queries and updates are still predominant and the reason of its poor performance at high thread count.

Glibc vs TCMalloc

- Interestingly, the memory allocator which proved to be generally best in our experiments is the traditional Glibc memory allocator (in HTM and non-HTM indexes). Despite TCMalloc being able to achieve better performance in some experiments, its results were not as consistent as Glibc's. Moreover, when combined with partitioned queries, the conflict and capacity abort count exponentially raised, drastically lowering performance.

HTM ease of use

- Another comment is that the ease of use of HTM currently is limited by a number of factors: first, lack of robust tools for automated self-tuning. Works like Tuner [47] or ProteusTM [48] could potentially simplify this task. The cost of/time spent while tuning HTM is in fact quite high: in our study we have considered at least 4 factors that had a strong impact on HTM's performance. Manually tuning them all is prohibitive and urges better tool supports. Second, debugging: when debugging HTM, the programmer is forced to inject superfluous code to monitor HTMs behaviour (e.g,. counters and flags), which are error prone and do not contribute to the programmers intentions. Most significantly, the inability to pinpoint where aborts occur in transactions poses quite some problems. Usually, this forces programmers to have an abstract idea of where transactions could be having conflicts in certain points of the program, and manually compute experiments to see check whether they were right, which be an iterative process until the cause is reached. Hence, proper debugging tools for HTM could potentially simplify this task and ensure greater productivity.

Architecture	**Query Intensive W**	**Update Intensive W**
Haswell	Serial	PGridHTM
Broadwell	Serial	PGridHTM
POWER8	Serial	PGridOLFIT

Table 4.3: Best performing indexes

Chapter 5

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 [5, 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 MOTO.

5.1 Future Work

In this section, we discuss the possibilities for future research in regards to this dissertation. This document makes an extensive study on the behaviour of HTM when applied to spatio-temporal indexes. Moreover, it highlights the most important factors in spatio-temporal indexes, which influence the most HTM performance.

A first direction would be to apply HRWLE to promising index structeres as Serial and PGridHTM (with partitioned queries). Serial uses a read/write lock scheme to synchronize updates and queries. Since HRWLE has uses the same concept with transactions, it could be worth to evaluate its gains/losses in performance. Moreover, Serial is best in query intensive workloads, which further suits the strengths of HRWLE. PGridHTM could also be an index interesting to study with HRWLE, as with its partitioned queries it solves the main difficulty of HRWLE, which is contention between updates and queries.

Another direction could be to prolong this study with STM or HyTM. HTM is the TM mechanism which can achieve the best throughput, due to its cache-coherency conflict detection, which ensures no software instrumentation and fast memory access. However, it is also the most volatile, due to its best-effort nature. Hence, depending on the workloads, indexes and platforms, it may not be the best option as the concurrency mechanism for spatio-temporal indexes. To deepen the study made on this dissertation, future research on the integration of spatio-temporal indexes with STM and HyTM would be a great opportunity.

STM offers great advantages which HTM cannot, for example, abundant memory capacity. However, it also has attached disadvantages not present in HTM, as software instrumentation. Similarly, HyTM is able to use the best of each TM system, for example, using the abundant memory capacity of STM for large transactions, and the quickness of HTM for small transactions. However, it also has disadvantages attached to it, synchronizing concurrent activities of STM and HTM transactions, has been shown to incur non-negligible overheads.

Bibliography

[1] K. Cukier. *Data, data everywhere: A special report on managing information*. Economist Newspaper, 2010.

[2] M. Selinger and L. Schmidt. Adaptive traffic control systems in the united states. *HDR Engineering, Inc*, 2009.

[3] A. Khanna. Facebook's privacy incident response: a study of geolocation sharing on facebook messenger.

[4] 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.

[5] 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.

[6] R. K. V. Kothuri, S. Ravada, and D. Abugov. Quadtree and r-tree indexes in oracle spatial: a comparison using gis data. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 546–557. ACM, 2002.

[7] D.-T. Lee and C. Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica*, 9(1):23–29, 1977.

[8] C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient b+-tree based indexing of moving objects. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 768–779. VLDB Endowment, 2004.

[9] S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi. Md-hbase: A scalable multi-dimensional data infrastructure for location aware services. In *Mobile Data Management (MDM), 2011 12th IEEE International Conference on*, volume 1, pages 7–16. IEEE, 2011.

[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] 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.

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

[13] 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.

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

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

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

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

[18] M. Herlihy and J. Moss. Transactional memory: architectural support for lock-free data structures. sigarch comput. archit. news 21 (2), 289–300 (1993).

[19] R. Guerraoui and M. Kapalka. Opacity: A correctness condition for transactional memory. Technical report, 2007.

[20] 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.

[21] 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.

[22] N. Diegues, P. Romano, and L. Rodrigues. Virtues and limitations of commodity hardware transactional memory. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 3–14. ACM, 2014.

[23] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of blue gene/q hardware support for transactional memories. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 127–136. ACM, 2012.

[24] C. Jacobi, T. Slegel, and D. Greiner. Transactional memory architecture and implementation for ibm system z. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, pages 25–36. IEEE, 2012.

[25] 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.

[26] 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.

[27] I. Calciu, T. Shpeisman, G. Pokam, and M. Herlihy. Improved single global lock fallback for best-effort hardware transactional memory. In *Transact 2014 Workshop. ACM*, 2014.

[28] D. Dice, T. L. Harris, A. Kogan, Y. Lev, and M. Moir. Pitfalls of lazy subscription. In *Proceedings of the 6th Workshop on the Theory of Transactional Memory, Paris, France*, 2014.

[29] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 258–264. ACM, 2005.

[30] L. Dalessandro, M. F. Spear, and M. L. Scott. Norec: streamlining stm by abolishing ownership records. In *ACM Sigplan Notices*, volume 45, pages 67–78. ACM, 2010.

[31] L. Dalessandro, D. Dice, M. Scott, N. Shavit, and M. Spear. Transactional mutex locks. In *Euro-Par 2010-Parallel Processing*, pages 2–13. Springer, 2010.

[32] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ACM Sigplan Notices*, volume 41, pages 336–346. ACM, 2006.

[33] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid norec: A case study in the effectiveness of best effort hardware transactional memory. *ACM SIGARCH Computer Architecture News*, 39(1):39–52, 2011.

[34] M. Duckham, M. F. Goodchild, and M. Worboys. *Foundations of geographic information science*. CRC Press, 2004.

[35] E. ESRI. Shapefile technical description: An esri white paper, 1998, 2014.

[36] T. Brinkhoff. Generating network-based moving objects. In *Scientific and Statistical Database Management, 2000. Proceedings. 12th International Conference on*, pages 253–255. IEEE, 2000.

[37] J. Dittrich, L. Blunschi, and M. A. V. Salles. Indexing moving objects using short-lived throwaway indexes. In *International Symposium on Spatial and Temporal Databases*, pages 189–207. Springer, 2009.

[38] G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier, 2001.

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

[40] S. Hwang, K. Kwon, S. K. Cha, and B. S. Lee. Performance evaluation of main-memory r-tree variants. In *Advances in Spatial and Temporal Databases*, pages 10–27. Springer, 2003.

[41] M. L. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Teo. Supporting frequent updates in r-trees: A bottom-up approach. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 608–619. VLDB Endowment, 2003.

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

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

[44] S. K. Cha, S. Hwang, K. Kim, and K. Kwon. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *VLDB*, volume 1, pages 181–190, 2001.

[45] 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.

[46] J. R. Bulpin and I. Pratt. Hyper-threading aware process scheduling heuristics. In *USENIX Annual Technical Conference, General Track*, pages 399–402, 2005.

[47] N. Diegues and P. Romano. Self-tuning intel transactional synchronization extensions. In *ICAC*, pages 209–219, 2014.

[48] D. Didona, N. Diegues, A.-M. Kermarrec, R. Guerraoui, R. Neves, and P. Romano. Proteustm: Abstraction meets performance in transactional memory. In *ACM SIGOPS Operating Systems Review*, volume 50, pages 757–771. ACM, 2016.

Appendix A

Fall-Back Paths

In this appendix we include the algorithms used to implement PGridHTM's SGL and FLG fall-back paths.

A.1 SGL fall-back path

Algorithm 23: TM_BEGIN_SGL( )

```
attempts_left = MAX_ATTEMPTS ;
fallback = false ;
while 1 do
    while IS_LOCKED(single_global_lock) do
        __asm__ ( "pause;");
    end
    unsigned status = begin_htm_tx();
    if status == _XBEGIN_STARTED then
        if IS_LOCKED(single_global_lock) then
            _xabort("abortCode");
        end
        break;
    end
    attempts_left–;
    if attempts_left <= 0 then
        while CAS(single_global_lock, 0, 1) == 1 do
            __asm__ ("pause;");
        end
        fallback = true;
        break;
    end
end
```

Algorithm 24: TM_END_SGL( )

```
if attempts_left > 0 then
    _xend();
else
    single_global_lock = 0;
end
```

A.2 FGL fall-back path

Algorithm 25: TM_BEGIN_FGL(lock)

```
attempts_left = MAX_ATTEMPTS ;
fallback = false ;
while 1 do
    while IS_LOCKED(lock) do
        __asm__ ( ”pause;”);
    end
    unsigned status = begin_htm_tx();
    if status == _XBEGIN_STARTED then
        if IS_LOCKED(lock) then
            _xabort(”abortCode”);
        end
        break;
    end
    attempts_left–;
    if attempts_left <= 0 then
        while CAS(lock, 0, 1) == 1 do
            __asm__ (”pause;”);
        end
        fallback = true;
        break;
    end
end
```

Algorithm 26: TM_END_FGL(lock)

```
if attempts_left > 0 then
    _xend();
else
    lock = 0;
end
```

Algorithm 27: TM_BEGIN_INNER_FGL(lock)

```
if attempts_left <= 0 then
    while CAS(lock, 0, 1) == 1 do
        __asm__ (”pause;”);
    end
else
    while IS_LOCKED(lock) do
        __asm__ ( ”pause;”);
    end
end
```

Algorithm 28: TM_END_INNER_FGL(lock)

```
if attempts_left <= 0 then
    lock = 0;
end
```

Algorithm 29: TM_BEGIN_INNER_DEADLOCK_FGL(lock)

```
if attempts_left <= 0 then
    if CAS(lock, 0, 1) == 1 then
        return false ;                                   // Avoid dead-locks
    end
else
    while IS_LOCKED(lock) do
        __asm__ ( "pause;");
    end
end
```

A.3 Non-blockable fall-back path

Algorithm 30: TM_BEGIN_NB_TX( )

```
attempts_left = MAX_ATTEMPTS;
fallback = false;
while 1 do
    begin_htm_tx();
    if status == _XBEGIN_STARTED then
        break;
    end
    attempts_left–;
    if attempts_left <= 0 then
        fallback = true;
        break;
    end
end
```

Algorithm 31: TM_END_NB_TX( )

```
if attempts_left > 0 then
    _xend();
end
```

Appendix B

Memory allocators in non-HTM indexes

In this appendix are the tests evaluating performance of the different memory allocators (Glibc and TCMalloc) in the non-HTM indexes. We recall that the best memory allocator for these indexes in all platforms was Glibc. Figures B.1, B.2 and B.3 are the tests respective to Haswell, Broadwell and POWER8.

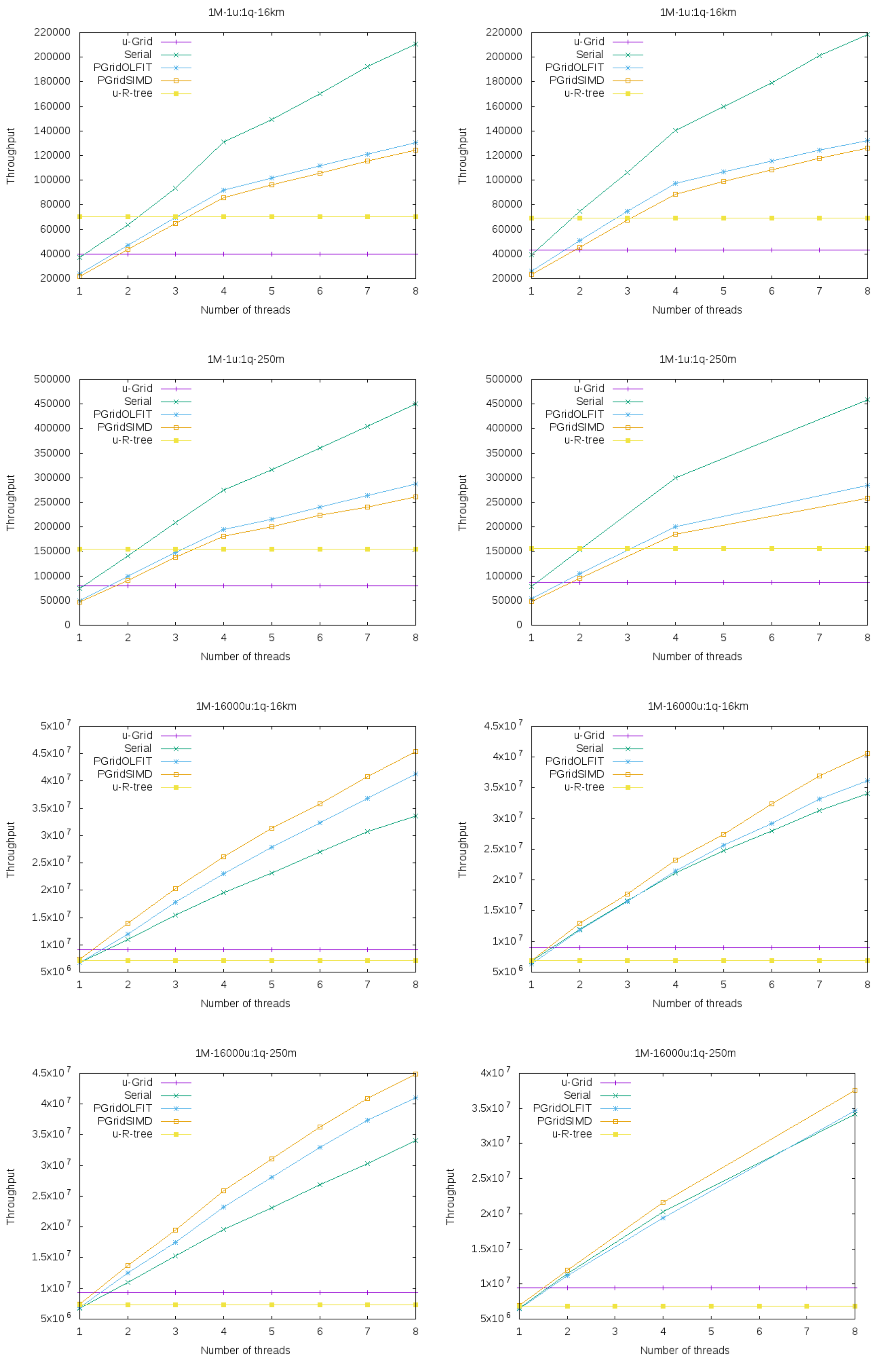


Figure B.1: Haswell - Non HTM indexes performance evaluation with different memory allocators. Left - glibc — Right - TCMalloc

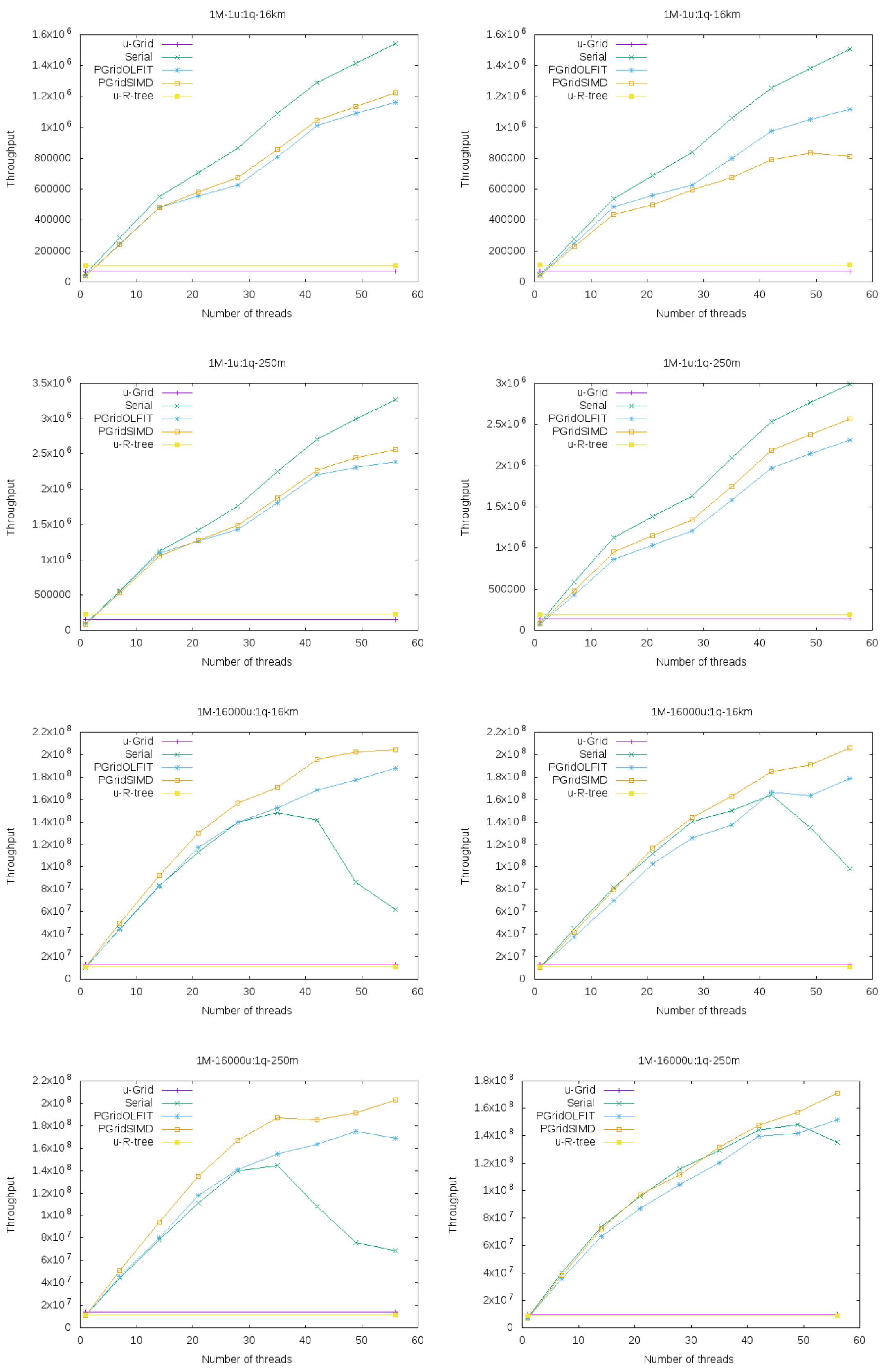


Figure B.2: BroadWell - Non HTM indexes performance evaluation with different memory allocators. Left - glibc — Right - TCMalloc

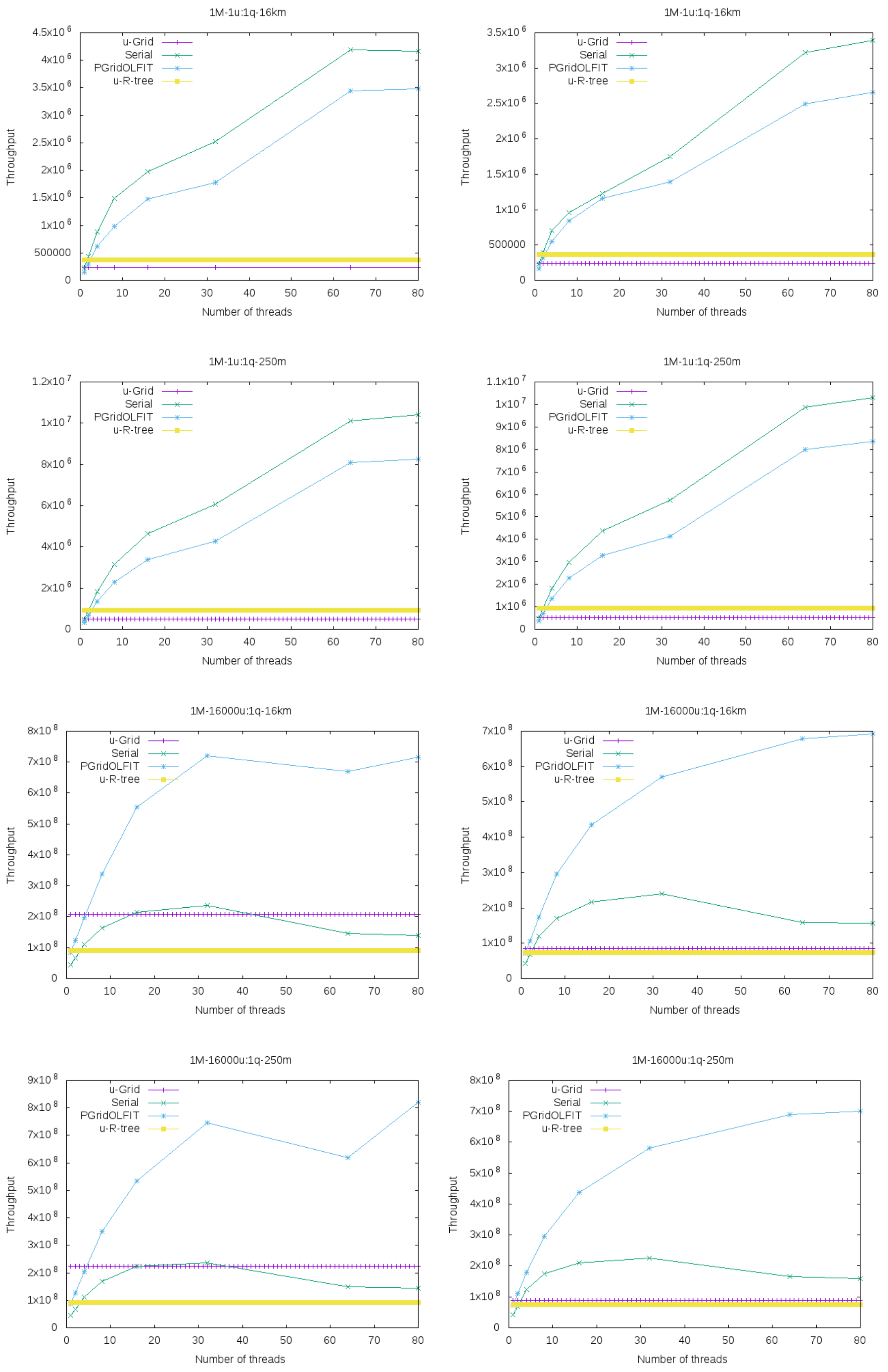


Figure B.3: POWER8 - Non HTM indexes performance evaluation with different memory allocators. Left - glibc — Right - TCMalloc