Enhancing efficiency of Hybrid Transactional Memory via Dynamic Data Partitioning Schemes

Pedro Raminhas

Instituto Superior Técnico
Avenida Professor Cavaco Silva, 2744-016 Porto Salvo,
Portugal
pedro.raminhas@tecnico.ulisboa.pt

Abstract Multicore processors represent the *de-facto* standard architecture for a wide variety of devices, from high performance computers to mobile devices. However, developing concurrent programs to harness the capabilities of multicore architectures using traditional lock-based synchronisation is known to be a complex and error prone task, as the fine-grained locking approach is prone to deadlocks, livelocks and it is difficult to debug, while coarse-grained locking does not provide performance gains. Transactional Memory (TM) has emerged as a powerful paradigm that promises to simplify the life of programmers while still achieving performance similar to fine grained locking, programmers only need to specify which parts of the code have to appear as executed atomically, and not how atomicity should be achieved. This document first overviews the state of the art of TM, analysing some of the main TM implementations based on software (STM), hardware (HTM), or on combination thereof (Hybrid TM). Next, It overviews the state of the art benchmarks that has been created to evaluate Transactional Memory implementations and stress the advantages and disadvantages over the traditional locking mechanisms. In the light of this critical analysis of the state of the art, the document proposes a novel Hybrid TM algorithm that aims to minimise synchronisation overheads between HTM and STM by relying on a novel dynamic memory partitioning scheme.

Keywords: Transactional Memory, Concurrent Programming, Data Partitioning, Efficiency

1 Introduction

Multicore processors have become the standard architecture in today's computing systems, ranging from high-end servers, to personal computers, as well as smartphones and embedded systems, all have the multicore technology and the trend is to increase the number of cores towards manycore architectures. Unfortunately, the development of parallel applications that can fully exploit the advantages of the multicore technology is far from being a trivial task.

Indeed, when using traditional lock-based synchronisation mechanisms, developers are faced with a major dilemma. They can either opt for using coarse-grained locks, which simplifies significantly reasoning on the correctness of concurrent applications, but can severely hinder parallelism and hamper performance. Or they can opt for a second alternative which is to use fine-grained locking. This approach allows for extracting the most parallelism possible, avoiding the inefficiency of coarse-grained locking, but it is harder to devise and more prone to deadlocks and livelocks, which are notoriously hard to debug [33]. In fact, fine grained locking is considered difficult to use for the average programmer [33], who may not be necessarily well trained for the development of concurrent applications.

Transactional Memory (TM) [25] precisely answers this urge by specifying a new programming paradigm that alleviates the complexity of classical locking mechanisms while ensuring performance similar or even better than complex fine-grained locking schemes. TM borrows the concept of transactions from databases and applies it to parallel programming: with TM, programmers devise code into blocks that will be ensured by the TM system to run atomically. The underlying system is responsible for either committing the transactions, and making it's modifications globally visible, or to aborting the transactions and ensuring that no modifications are observed out of the context of the transaction. This approach leads to drastically simplifying the development of parallel applications and abate their time to market and costs.

Transactional Memory was initially proposed twenty years ago as an extension to multi-processors' cache coherence protocols [25]. Due to the difficulty of rapid prototyping in hardware environments, researchers resorted to Software Transactional Memory to advance the state of the art [12,19,20,22]. Simultaneously, hardware-based implementations have also been proposed, whose designs were validated using simulators.

Hardware Transactional Memory (HTM) is currently supported by the mainstream Intel Haswell and the IBM Power8 processors. These implementations are *best-effort*, which means that they always ensure correctness, but do not provide any progress guarantee. This means that transactions executing in hardware may never commit (even in absence of concurrency) due to inherent limitations of hardware, such as transaction exceeding capacity of hardware or running prohibited instructions. This raises the need for a software fallback path to ensure forward progress [18].

This fallback path can be either as simple as a global lock or as complex as a software transactional memory, which is designated by Hybrid Transactional Memory (HyTM) [8,11,13,29,36]. The goal of HyTM is to fully exploit *best-effort* HTM as much as possible, and in the case of abort, fall back to the more costly Software Transactional Memory (STM), which provides progress guarantee.

Despite the number of papers published in this area in recent years, HyTM still suffer from large overheads [18]. These overheads are noticed on HyTMs based on early STM implementations [36], which requires dealing with per-location metadata and consequently makes the implementation prone to capacity aborts due to the handling of metadata by HTM [11]; but also in HyTMs that use a software fallback without per location metadata, as these HyTMs suffer from a scalability bottleneck as every transaction must read a sequence lock. Further, they can induce spurious aborts of HTM transactions if any concurrent, non-conflicting HTM commits [18]. Benchmarks' tests indicate that the performance of state of the art HyTM can not surpass HTM performance for workloads characterised by short transactions with small read and write-set, and that it can not surpass the scalability of STM for workloads characterised by long transactions with large read and write-set. The results show that HyTM has costs incurred by synchronising both the execution of HTM and STM, as the memory locations accessed by one TM has to be validated with the read and write-set of both TMs. Results also show that the performance of HyTM is highly dependent on the workload.

The goal of this work is to provide a new class of HyTM without the overheads incurred by the synchronisation of HTM and STM. The solution proposed in this document is to dynamically partition the memory into two distinct partitions where both STM and an HTM can execute transactions disjointly, thus avoiding overheads of the synchronisation between both systems. The main challenge is how to ensure safety (i.e. accesses by HTM and STM are actually constrained to their current partitions) without imposing costly instrumentation overheads on HTM and also STM.

The rest of this document is structured as follows: Section 2 overviews the state of the art in the TM field. Section 3 describes the architecture of the proposed solution. Section 4 focuses on how the proposed solution will be evaluated and which metrics are going to be used and Section 5 contains the scheduling for the future work.

2 Related Work

This section overviews the state of the art of Transactional Memory, it first begins with STM and presents three implementations that address different design approaches. Then, it presents the state of the art of HTM and shows the virtues and limitations of each HTM available. Next, it presents the state of the art of HyTM and it presents six different approaches to synchronise both HTM and STM. Finally, is presented an overview of state of the art benchmarks used to evaluate TM.

2.1 Software Transactional Memory

Software Transactional Memory (STM) is a category of systems that implement the Transactional Memory abstraction through a purely software-based runtime library.

During the last decade, STM was object of a thorough research mainly because it is a solution that is not bound to a specific architecture, nor it is restricted by the underlying hardware, making STM a widely portable solution. In STM, transactional reads and writes are tracked by a software runtime, which has the responsibility of maintaining the read and write set of transactions. Besides maintaining the read and write sets, most STMs keep additional metadata about transactions, such as locking, ownership records and versioning.

Existing literature in the area of STM can be coarsely classified according to the following design choice choices: word-based vs object-based, lock-based vs lock-free, write-through vs write-back, eager vs lazy conflict detection. word-based STMs access the memory directly at the granularity of machine words or larger chunks of memory, yet object-based STMs access the memory at object granularity and it requires the TM to be aware of the object associated in every access. In terms of locking mechanisms, lock-based STMs uses locks to control the concurrency between transactions accessing the same data while lock-free STMs do not use any lock to protect memory from concurrent accesses. STMs can be write-through, writing their updates directly to memory and storing the previous ones on a undo log, or write-back which means that changes are only written to memory at commit-time. Finally, the conflict detection can be eager, which means that conflicts are detected at the moment of occurrence, or it can be lazy if the detection is performed at commit-time.

The best design can vary according to a number of factors like the underlying architecture, the number of cores per CPU and the size of caches. More important to note is that the efficiency that will result from these choices is highly dependent from the type of workload generated by the application.

In comparison with HTM, STM systems incur heavy instrumentation on the code, issuing modification on the atomic code blocks in order to account for transactional operations and other operations performed by the underlying system as log keeping and conflict detection. Despite of the drawbacks, state of the art STM normally provides stronger progress guarantees than HTM: being fully implemented in software, STM does not suffer from the restrictions that affect HTM and can support the execution of arbitrarily long transactions.

TinySTM TinySTM [20] is a STM implementation presented by Felber et al. in 2008.

Authors state that the characteristics of the workload of an STM implementation plays a major role in selecting the right choice and configuration parameters. Some of the characteristics are the ratios to update read-only transactions, the size of read and write-sets and contention on shared memory.

TinySTM is a *word-based* STM implementation, meaning that it allows for directly mapping of transactional accesses to the underlying memory subsystem.

This STM uses an encounter-time locking mechanism, as authors state that it increases the transactions throughput because transactions do not perform useless work. This mechanism also allows the efficient handling of reads-after-write conflicts without requiring expensive or complex mechanisms, a valuable feature especially when write-sets have non-negligible size.

Along with encounter time locking, two other strategies can be used in TinySTM to access memory: write-through and write-back access. In write-through access, transactions immediately write to memory and undo updates if aborted. In write-back access, transactions do not update until the commit time.

As most word-based STM designs, TinySTM relies upon a shared array of locks to manage concurrent accesses to memory. Each lock covers a portion of the address space. Addresses are per-stripe mapped to the lock based on a hash function. Each lock represents a size of address in memory, as the least significant bit shows whether lock is owned or not. If the lock is owned, the remaining bits of address store the owner transaction, if not, a version number based on the timestamp of the last owner transaction is stored in the remaining bits.

The locks are used to protect memory, when a transaction issues a read it first checks if the lock protecting the read item is currently being held by other transaction. Then, it reads the value of the item, and finally the lock again. If the lock is not held by other transaction or the value of the lock did not change between both reads, then the value read is consistent.

When a transaction writes to a memory location, it reads the lock entry from selected memory addresses. If it finds that the lock bit is set then it verifies either the current transaction is the owner or not. If current transaction is the owner, then it simply writes the new value to memory location. If it is not owner of the current transaction then the current transaction can wait for some time to get resources free or aborts immediately.

TinySTM guarantees that there is a consistent read-set upon each read for read-only transactions, therefore read-only transactions do not need to validate their read-sets upon commit. Update transaction have to validate their read-set before updating any memory location. The downside of this approach is that the validation of large read-set may be costly, to avoid this overhead the number of locks can be reduced. On the other hand reducing the number of locks can increase abort rate as it increases false sharing.

In order to solve this problem, authors proposed hierarchical locking. In addition to the shared array of locks, it is maintained a smaller array of counters which goal is to hold the number of commits done to locations in that region. An hash function is used to map memory addresses to counters, this function is consistent with the one used to map addresses to lock arrays, i.e. memory locations that are mapped to the same lock are also mapped to the same counter, which implies that a counter covers multiple locks and the associated memory addresses. This scheme allows transactions to determine whether locks have been acquired or not, however it has an overhead associated with it but authors state that it is amortised when transactions have large read sets or when there are few writes from competing transactions.

TinySTM periodically adapts the tuning parameters at runtime and measures the throughput over a period of approximately one second. The system registers the most recent throughput for each tuning configuration. Each tuning configuration is a triple consisting of the number of locks, the number of shifts, and the size of the hierarchical array. The tuning strategy is a hill climbing algorithm with a memory and forbidden areas in order to achieve the best configuration to the given workload.

STM with Data Partitioning Scheme In order to mitigate the concern of different parts of the data having different access patterns, thus requiring specific design optimisations, Riegel et. al [35] proposed the use of a data partitioning scheme integrated with TinySTM [20]. This approach enables STM to divide data structures in partitions and to compose different specialised optimisations for each partition accordingly to the workload or even to use a different STM backend in that partition, hence improving the transactional throughput of each partition. To identify the partitions of an application, it is constructed a Data Structure (DS) graph at compile-time for every function encountered in the program. The DS graph is determined by analysing to which node (partition) does the pointers in a program are permitted to point to, and whenever two pointers stored in the same field point to disjoint nodes, those nodes are unified. Each partition has a type that indicate the concurrency control. The partitions types vary from read-only partitions to partitions with multiple locks, each partition type is determined at runtime accordingly to the number of aborts. The inherent problem of this approach is the usage of pointers in certain languages like C. Pointers allow to access memory directly, thus another partition, without the knowledge of the STM.

SwissTM SwissTM [19] is a STM implementation presented by Dragojevic et al. in 2009.

Authors state that state of the art STMs only have good performance for workloads with small scale transactions, but in practice do not work well for large scale applications like business software and video games.

This STM uses four-word lock granularity to protect shared memory, as authors state that this approach outperforms both word-level and cache line-level locking for all benchmarks considered.

It uses a *mixed invalidation* conflict detection scheme which eagerly detects write/write conflicts. Thus preventing long transactions, that are doomed to abort, to continue their execution and possibly cause more conflicts with other running transactions. On the other hand, read/write conflicts, commonly caused by short transaction with longer ones, are lazily detected. By using invisible reads and allowing transactions to read objects acquired for writing, SwissTM detects read/write conflicts late, thus increasing inter-transaction parallelism. A time-based scheme is used to reduce the cost of transaction validation with invisible reads.

For conflict detection it uses a two-phase contention manager scheme that incurs no overhead on read-only and short read-write transactions. Upon a write, every transaction increments a local counter. If the value of this counter is below 10, then it is considered a short transaction and in case of conflict it will use a timid contention management scheme, aborting on the first encountered conflict. Transactions that are more complex increment the global counter above 10 are then switched dynamically to the Greedy mechanism that involves more overhead but favours these transactions, thus preventing starvation. This means that more complex transactions have higher priority than simpler transactions.

Additionally, transactions that abort due to write/write conflicts back-off for a period proportional to the number of their successive aborts, hence reducing the probability of aborting repeatedly because of the same conflict.

Authors state that SwissTM outperforms all state of the art STM in *mixed* workloads characterised by non uniform, dynamic data structures and various transaction sizes while also delivering good performance in smaller-scale scenarios.

NOrec NOrec is a STM library presented in 2010, by Luke Dalessandro, Michael Spear and Michael Scott [12].

Authors advocate a minimal approach on STM handling of metadata. NOrec does not need fine grained, shared metadata, only requires a single global sequence lock (*seqlock*) for concurrency control. Transactions buffer writes and log read address/value pairs in thread local structures.

Upon start, transactions poll back seqlock and check if there is any transactions in write-back phase; if there is, transactions waits until there is no transaction in the write-back phase. Committing writers increments the seqlock in order to signal that they have written new values to memory.

When a read is issued, active transactions poll the global seqlock and compare it with the one held locally by the transaction. A new value is evidence of possible inconsistency and triggers validation, which is carried out in a value-based style by comparing the pair address/value in log to the actual values in main memory.

In this protocol, only a single writer can commit and perform writeback at a time. This sequential bottleneck is minimised by validation prior to acquiring the *seqlock* for commit.

By successfully incrementing the seqlock, a writer transitions to a committed state in which it immediately performs it's writeback and then releases the lock. Read-only transactions can automatically commit because their reads are proven to be consistent because of the validation mechanism.

NOrec scales well when it's single-writer commit serialisation does not represent the overall application bottleneck, i.e., writeback does not dominate the runtime, and has been shown to have low latency, although it's read set entries are twice as large as the corresponding orec-based implementations, as NOrec stores both the address of the read location and the value that was seen.

NOrec guarantees consistency among transactional and non-transactional accesses to shared data, a characteristic named privatization [30,38], which is required for compatibility with the C++ draft TM standard [4]. Further, it provides support for closed nesting, where inner transactions can abort and restart separately from their parents, which is expected to improve performance in certain applications.

Authors conclude that NOrec have low overhead and high scalability, which is ideal for use in legacy system as well as a fallback mechanism in HyTM due to it's robust performance.

2.2 Hardware Transactional Memory

Although TM was originally proposed as an micro-architectural feature [25], the absence of hardware support switched a large body of TM research into STM [12,19,20,22]. Unlike Hardware solutions, STM require instrumentation of reads and writes in order to track the read and write-set of active transactions and detect conflicts between them. This instrumentation can introduce, in certain scenarios, large overheads that are non-negligible and can hinder performance with respect to conventional fine-grained locking.

HTM is thus desirable, as it relies on modified cache coherence protocol in order to achieve atomicity and isolation. The main advantage of HTM is that it requires no instrumentation of reads and writes, which degrades the performance of STM solutions, as the read and write-set are stored in L1 cache and conflicts are detected by the cache coherence protocol. A major drawback is that current HTM does not guarantee that a transaction will succeed, even without concurrency, due to it's limited nature, which is limited by the fact that read and write-set fits in L1 cache.

The maturing of TM led to a change in the industry of processor manufacturing, as manufacturers revealed the first processors that successfully implemented HTM: Azul [10] and Rock [17] processors. Unfortunately, these system were not usable, as Azul programming interface was not disclosed, thus making HTM hidden for the programmers and Rock processor was cancelled before reaching the market. IBM Blue Gene/Q was the first processor on the market that provide an HTM implementation [39], followed by zEnterprise EC12 [2], IBM Power8 [3] and Intel Haswell [40]. This represented a significant milestone for TM, as with Intel Haswell processor, TM became available on commodity hardware from high-end servers to mainstream laptops.

Haswell uses the L1 cache for conflict detection and maintaining versions and transactional metadata. However, the details about the conflict detection and transaction capacities have not been disclosed by Intel. The Intel Transactional Synchronization Extensions (TSX), i.e., the code-name for the HTM API in Haswell CPUs, comprises two possible interfaces, Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM). The former allows to elide locks [6] and execute code speculatively in a backwards-compatible manner. RTM leverages on the same hardware as HLE, but exposes a transaction interface in which the programmers only need to specify which blocks are atomic, leaving to the underlying system the responsibility to ensure the correct execution of transactions.

A very important detail is that the begin instruction requires to specify a software handler to deal with transaction aborts, and thus provide progress guarantee. The software fallback must co-operate with HTM in order to ensure correctness and the mechanism used has a great impact on the performance of TSX. The simplest approach and the one suggested by Intel's optimisation manual is the use of a single global lock as a fallback mechanism. When a hardware transaction aborts, it has the alternative to check and acquire the global lock if it is free. To ensure correctness, every hardware transaction must subscribe to the global lock and ensure it is free, as the underlying transactional semantics will guarantee that a transaction only commits when there is no ongoing fallback execution.

IBM Blue Gene/Q uses the L2 cache for conflict detection and updates buffering. It assigns a unique speculation ID to each transaction and uses the L2 cache to store the speculation ID upon a transactional access. IBM Blue Gene/Q has two transactional execution modes: a short-running mode a long-running mode. In the short-running mode, it uses only the L2 cache to buffer transactional data, and in the long-running mode, it uses the L1 cache to buffer some of the transactional data though it invalidates all of the L1 cache lines at the start of each transaction.

IBM zEC12 uses the L1 cache for both conflict detection and loads buffering, along with a special LRU-extension, which function is to record the evicted cache lines. The transactional stores are buffered in an 8-KB gathering store cache, which is private for each processor and is located between the L1 cache and the L2/L3 caches. IBM zEC12 also provides constrained transactions, which are transactions that are guaranteed to commit.

IBM Power8 uses content addressable memory (CAM) linked with the L2 cache for conflict detection [32]. The L2 TMCAM records the cache-line 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. Power8 support rollback-only transactions, which are transactions that support normal transactional semantics, i.e. store buffering, but not conflict detection. Also, Power8 support the use of suspend/resume operations, which allows to escape from a transactional context and access a variable without risking to incur data conflicts.

Nataike et. al in [32] compared all state of the art processors that provide HTM, by using a modified version of the STAMP's benchmark suite [31] better suited to accommodate the limited capacity of HTM. The experimental results showed that there is no single HTM system that is more scalable than the others in all of the benchmarks. Each HTM system has it's own implementations issues that limit the scalability in certain workloads.

Results shows that Intel Core has load and store capacity of 4MB and 22KB respectively. This processor has extra transaction aborts due to hardware prefetching, as it could raise a conflict that would not exist if the hardware did not prefetch other cache lines in advance. L1 and L2 size are respectively 32KB and 256 KB.

IBM Power8 has more capacity-overflow aborts than the other processors because of it's small transaction capacity, i.e., 8KB for loads and stores, respectively. L1 cache is one of the biggest in the study, with 64 KB is only surpassed by zEC12's 96KB. L2 cache size is 512 KB. Results show that the suspend and resume instructions are beneficial for avoiding data conflicts on a shared variable to implement ordered transactions.

Diegues et. al [18] and Goel et. al [21] experimentally evaluated Intel TSX implementation on a variety of workloads produced by different benchmarks, more specifically the workloads produced by STAMP benchmark suite [31], microbenchmark Eigenbench [27] and Memcached [26], as well as workloads produced by concurrent data structures. Authors state that TSX has outstanding performance for workloads characterised by small transactions, such as the ones produced by concurrent data structures and Memcached [26], but only with two of the STAMP benchmarks, namely Kmeans and SSCA.

Goel. et al [21] also compared the performance and energy consumption of TSX with TinySTM [20]. The results shows that TSX performance relies heavily on the access patterns to L1 cache, as every memory access performed inside a transaction is tracked, implying that long running transactions can lead to frequent capacity exceptions and spurious aborts. When transaction's intensity is medium, TSX is only the best choice for a limited degree of parallelism, and it is generally better on the energy side than on the performance side. However as the contention and duration of the workload increases, TinySTM begins to outperform TSX, as TSX deals worse with contention than TinySTM due to the detection of conflicts at granularity of cache cache line (bytes). TSX scales well up to four threads, however at eight threads, the L1 cache is shared between two threads running on the same core. This cache sharing degrades performance for the larger working set more than for the smaller working set because hyper-threading effectively halves the write-set capacity of TSX. In contrast, TinySTM scales well up to eight threads. In terms of energy efficiency, TSX proves to be more energy-efficient than either TinySTM or the sequential runs.

Lazy Subscription To ensure that a critical section executed by a hardware transaction does not observe partial effects of a critical section executed by another thread that acquires the lock, the transaction subscribes the lock, by reading it and confirming if it is available. Subscribing to the lock makes hardware transactions vulnerable to abort if another thread acquires the lock. Typically, transactions subscribe to the lock at the beginning of the critical section and are thus vulnerable to such abort during the entire execution of the critical section. Some papers [11,29] proposed the, so called, *lazy subscription* optimisation, which delays lock subscription, in order to reduce the duration of this vulnerability.

Dice et. al in this study [15], reported a number of issues associated with *lazy subscription.* Lazy subscription can cause a transaction to deviate from the behaviour allowed by the original program, as it can result in the thread's registers containing values that could not occur in an execution of the original program. This could result in an access by a transaction to memory that could never be accessed in the original program. If transaction commits, it results in a observably incorrect behaviour.

Compiler support suggested by other publications [9] for avoiding such issues in single global lock-based transaction system is not sufficient to avoid all the pitfalls associated with *lazy subscription.* Authors argue that the complexity required to address these issues via static analysis is unlikely to be worthwhile, as there are numerous pitfalls associated with lazy subscription, so manual confirmation of it's safety in specific cases is likely to be error prone.

2.3 Hybrid Transactional Memory

The usage of a single global lock as a fallback mechanism in *best-effort* HTM motivated the researchers of TM to create Hybrid Transactional Memory (HyTM). The goal of HyTM is to exploit *best-effort* HTM whenever possible due to it's cheaper capabilities, and fallback to the more costly STM when a transaction can not complete in hardware. This approach promises to scale well and incur low latency, with the worst-case overhead and scalability comparable to the underlying STM.

To perform as a coherent whole, the HyTM system must be able to detect conflicts in hardware and software transactions simultaneously. This requirement is usually achieved by logging additional metadata while executing hardware transactions, so that the conflict detection system has access to both sides information. Certain STM algorithms [19,20] required interaction with per-location metadata, and hybrid versions of these algorithms wasted limited hardware capacity on these metadata. The interaction of HTM with STM metadata could lead to capacity aborts, or could lead to false sharing of cache lines that hold the metadata, resulting in additional HTM aborts, and increased fallback to the STM path. Table 1 summarises the key characteristics of state of the art STMs and HTMs and compares with 5 state of the art HyTM implementations, Hybrid NOrec [11], Hybrid-LSA [36], Hybrid NOrec-2 [36], Invyswell [8] and RH NOrec [29].

Hybrid NOrec Dalessandro et al. [11] used NOrec STM [12] (see section 2.1) as a fallback STM of their Hybrid NOrec to avoid per-access overheads. In this algorithm, the commits of write transactions are executed sequentially, which means that only one transaction can commit at a time. A shared global clock, *seqlock*, is used to notify concurrent transactions about the updates to both hardware and software.

The original Hybrid NOrec has each transaction start to execute in hardware, and if repeatedly fails to commit, it falls back to executing the NOrec STM in software. To coordinate the execution of hardware and software transactions, three possible integrations between NOrec and HTM are proposed. In naive integration, the same global clock from NOrec, *seqlock*, is used by hardware transactions to subscribe to software transactions commits notifications. Upon the beginning of hardware transactions, the global clock is read and if is already owned by a software transaction, the hardware transaction spins until the owner completes the writeback and releases the lock. Upon commit, both software and hardware transactions update the global clock to signal their commits to other transactions, which triggers a validation phase in all concurrent transactions. Unfortunately, each hardware transaction conflicts with every software transaction due to the early subscription of the global clock, and aborts when any software transaction commits, regardless of actual data conflicts. Similarly, each hardware transaction conflicts with every other hardware transaction and aborts when any hardware transaction commits.

The second integration approach is 2-Location, in this approach a second shared location is added, *counter*, which decouples subscribing from signalling. Hardware transactions still subscribe to software transactions by reading the global clock but they use *counter* to signal their own commits to other hardware transactions. The drawback of having a second shared counter is that software transaction must have a snapshot for both the global counter and the second shared counter and must perform additional validation by reading the second shared counter.

The third approach is P-Location, which differs from 2-Location because hardware transactions no longer conflict with other running hardware transactions, as the *counter* is now distributed. With this approach, software transactions must poll *n + 1 counters*, increasing the overheads. To mitigate that, the number of counters is dynamically determined based on current system conditions.

In order to optimise the algorithm, it was proposed the usage of mechanisms such as lazy subscription, sandboxing and communication filters. In Lazy subscription, the read of the global clock is delayed prior to commit. This increase the concurrency between hardware and software transactions, however it sacrifices opacity [23], as nothing prevents a hardware transaction from reading inconsistent data during software writeback. Authors re-establishes opacity by adding a hardware read barrier that polls the *seqlock* nontransactionally and pauses if it is locked. Sandboxing is also proposed as an alternative to opacity, rather than instrumenting all loads for consistency, instructions are only instrumented when it's effects may expose a transaction inconsistency. With the SW-Exists communication filter, a portion of shared memory is allocated in order to software transactions make their presence visible. When there is no software transaction running, hardware transactions have the possibility of eliding counter updates.

Hybrid-LSA Riegel et al. [36] presented two HyTM algorithms that can execute HTM and STM transactions concurrently and can thus provide good performance over a large spectrum of workloads. The algorithms exploit the concept of speculative (transactional) and nonspeculative accesses, which are non-transactional accesses to memory, i.e., without adding the memory addresses to the read and write-set. This mechanism avoids conflicts with other running transactions, which decreases the transactions' runtime overhead, abort rate and hardware capacity requirements compared with the versions that use speculative operations.

The paper assumes the availability of nonspeculative operations, which are provided by AMD's Advanced Synchronization Facility (ASF) specification [5], however they are not available in any current HTM implementation, except for IBM Power8, which supports them at a coarser granularity via the suspend/resume mechanism.

The first algorithm, Hybrid-LSA, extends the lazy snapshot algorithm first presented in [34] (overviewed in Section 2.1). LSA relies on the use of ownership records *(orecs)* protecting regions of memory and a global time base to check the consistency of transactions. Hybrid-LSA decides at runtime whether to use hardware or software transactions, which both use *orecs* in order to mediate the access to regions of memory.

The eager variant of the Hybrid-LSA algorithm first performs a load of the *orec*, this operation starts monitoring the *orec* for changes and will lead to an abort if the *orec* is updated. If the *orec* is not locked, the transaction uses a nonspeculative load operation to read the target value, the usage of nonspeculative operation is an efficient way to get the value without adding the address to the read-set.

The write operation is similar to the load operation, but before writing speculatively to memory, the *orec* protecting the memory region is checked for concurrent load and stores (via the PREFETCHW operation, provided by the ASF assembly [5]).

Upon commit, hardware transactions nonspeculatively acquire and increment the clock. This operation sends a synchronisation message to software transactions, notifying them that a hardware transaction is on commit phase and thus, they might have to validate. Next, they speculatively write all updated *orecs* to memory and try to commit. If transactions successfully commit, then is guaranteed that no conflict with other transactions exists.

Hybrid NOrec-2 The second algorithm, Hybrid NOrec-2 [36], is an optimisation of the algorithm informally described in [12]. The main approach of Hybrid NOrec proposed by [12] is to use a word-sized global sequence lock *gsl* and an extra sequence lock *esl*. Software transactions acquire both locks and increment their versions upon commit, whereas hardware transactions monitor *esl* for changes and increment *gsl* only on commit. Thus, software transactions are notified about data being concurrently modified by *gsl*, and use *esl* to abort concurrent transactions and prevent them from executing during software transactions writeback. The two major problems with this approach is that it does not scale well in practice, as an update of *esl* will make every hardware transaction to abort, even if both transactions have disjoint working sets. Also, every update of *gsl* done by a hardware transaction cause every other hardware transactions to abort, even if it is made close to the end of a transaction, as suggested by [12].

Authors proposed optimisations to this algorithm using both speculative and nonspeculative operations, which results in a better performance than the original algorithm, with shorter read and write-sets, using simple optimisations.

The first straightforward optimisation consists in having hardware transactions to update *gsl* only if they will actually update shared state on commit, relieving other hardware transactions from aborting due to the update of *gsl*. Also, in order to provide more concurrency between hardware transactions that access disjoint data, the write of *gsl* can be replaced by a *nonspeculative* atomic fetch-and-increment operation, which allows the algorithm to scale better.

Additionally, hardware transactions do not monitor *esl* using speculative accesses anymore. The purpose of *esl* is to prevent hardware transactions from reading inconsistent state such as partial updates by software transactions. To detect such cases and thus still obtain a consistent snapshot, hardware transactions first read the data speculatively and then wait until they observe with nonspeculative loads that *esl* is not locked . If this succeeds and the transaction reaches the end without being aborted, it is guaranteed that it had a consistent snapshot valid.

Invyswell Calciu et al. [8] proposed Invyswell, a HyTM that combines Haswell RTM [18,21] transactions with software transactions from a heavily modified version of InvalSTM [22]. The main goal of this paper is to improve performance of small to medium-sized transactions that are executed in STM, whose instrumentation cost causes them to perform poorly.

InvalSTM [22] is based on commit-time invalidation. The read and write-sets of a transaction are stored in transaction specific Bloom Filters and upon commit the transaction has complete knowledge of the conflicting running transactions. Authors state that for these reasons, InvalSTM works well with Haswell RTM. RTM is used for short transactions and low thread counts, while InvalSTM is used for large transactions and high thread counts. RTM can leverage InvalSTM use of Bloom filters for conflict detection by augmenting Haswell's hardware transactions with Bloom filters to enable many hardware transactions to execute concurrently with many software transactions. This enables RTM to perform without interference when read-only software transactions are executing within InvalSTM, regardless of their size.

To manage the shared-memory between RTM and InvalSTM, Invyswell performs the conflict detection after the hardware transaction commits. This is due to the constraints of Haswell RTM, since every write operation done by a transaction is held until commit-time. RTM does not support escape actions, hence when a hardware transaction conflicts with a software one, it aborts. By combining invalidation and conflict detection after a hardware transaction commits, this scheme minimises the chance to abort a hardware transaction due to an in-flight software one.

In order to adapt to different types of workloads, Invyswell support 5 types of different transactions, each one with different characteristics:

- Speculative Software Transactions (*SpecSW*), which is a type of transactions similar to an InvalSTM transaction, i.e, uses Bloom filters to track the read and write-sets of the memory addresses accessed. The invalidation is done after the commit of hardware transactions.
- Bloom filter Hardware Transaction (*BFHW*), which uses Bloom filters in order to handle conflicts with SpecSW transactions. After the BFHW transactions commit, it invalidates concurrent SpecSW transactions.
- *LiteHW* transactions, which are lightweight hardware transactions that execute without read or write instrumentation. These transactions can only commit if there are no in-flight software transaction when they begin their commit phase. Because LiteHWs do not maintain read or write-set metadata, if a software transaction is in-flight when a LiteHW enters it's commit phase, Invyswell must assume a conflict exists between the LiteHW and the software transaction and, therefore, must abort the LiteHW.

– *IrrevocSW* transactions, which is a software transaction that by being repeatedly aborted, gets a higher priority and acquire a global lock in the beginning of the execution. IrrevocSWs transactions write directly to memory, thus there is no commit phase.
– *SglSW* transactions, which are small software transactions that execute instructions not supported by Haswell RTM, thus need to be executed in software. This transaction type does not use metadata as it writes directly to memory.

Transactions are scheduled in a performance descending order: first the high-risk hardware transactions, then the low-risk software transaction. The transitions between the types of transactions are decided at runtime, based on an application dependent heuristic.

Invyswell has been tested with STAMP benhmark [31] and results show that it performs better in a range of benchmarks than state of the art NOrec [12].

Reduced Hardware NOrec The most recent approach in HyTM is the reduced hardware (RH) proposed by Matveev et. al [28]. The authors applied it to the Hybrid NOrec algorithm, resulting in a new HyTM that overcomes the scalability limitations of Hybrid NOrec [11].

In order to eliminate Hybrid NOrec's scalability problems, two hardware transactions are added to the software transactions resulting in a *mixed-slow path*. Both of these bottlenecks are caused by reading of the shared clock too early by software and hardware transactions. The first hardware transaction added to the *mixed-slow path*, called *HTM postfix*, encapsulates all the slow path writes at commit point and executes them all together. This change to the *mixed-slow path* enables the hardware transactions to delay the read of the clock to just before committing, avoiding the frequent false-aborts of the original Hybrid NOrec. The other hardware transaction added to the *mixed-slow path*, called *HTM prefix*, executes the largest possible prefix of slow-path reads in a hardware transaction, this is done by starting the *mixed-slow path* as a hardware transaction and executing within it as much reads as possible, or until the first write is encountered. This hardware prefix allows deferring the read of the global clock to after the reads, which significantly reduces the chances of aborting.

Authors state that the algorithm preserves opacity and privatization, as the original Hybrid NOrec. Finally if the *mixed-slow path* fails to commit, the algorithm reverts to the original Hybrid NOrec.

The results of Reduced Hardware NOrec with STAMP benchmark suite [31] indicate that RH NOrec is able to reduce the number of HTM conflicts comparing to Hybrid NOrec [11].

Table 1 summarises all TM implementations presented on the document. First, it is characterised the type of TM as STM, HTM or HyTM. Second, it is described the existence of metadata and their usage by the TM implementation. Next, it overviews the existence of spurious aborts, i.e. transactions that were aborted unnecessarily, even when they did not threaten correctness. Then, it overviews the usage of hardware capacity, which is only available on HTM and

HyTM. Finally, it overviews if the TM implementation has privatisation safety, and also the existence of invisible reads, i.e. reads made by one transaction that are not visible to the other active transactions.

TM implementation	Type	Metadata	Spurious Aborts	HW capacity used for	Invisible Reads	Privatisation Safety
TinySTM [20]	STM	Yes, orecs	No	-	No	Yes
SwissTM [19]	STM	Yes, four-word lock granularity	No	-	No	Yes
NOrec [12]	STM	No	No	-	Yes	Yes
TSX HLE	HTM	No	Yes	Read and write-set	Yes	Yes
TSX RTM with Global lock	HTM	No	Yes	Read and write-set	Yes	Yes
IBM Power8	HTM	No	Yes	Read and write-set	No, because of *pause/resume* instructions	Yes
Hybrid NOrec [11]	HyTM	lock counters	Less frequent	Data and lock counters	Yes	Yes
Hybrid-LSA [36]	HyTM	Yes, Orecs	Yes	Orecs and Data updates	Yes	No
Hybrid NOrec-2 [36]	HyTM	Yes, Orecs	Yes	Orecs and Data updates	Yes	No
Invyswell [8]	HyTM	Read and Write Bloom-Filters	Yes	Depends on the state of the system	No	Yes
RH NOrec [29]	HyTM	Yes, Lock counters	Yes	Data and locks	Yes	Yes

Table 1: Comparison of various TM state of the art implementations: TinySTM, SwissTM, NOrec, TSX HLE, TSX RTM with global lock, IBM Power8, Hybrid NOrec, Hybrid-LSA, Hybrid NOrec-2, Invyswell and Reduced Hardware NOrec.

2.4 Benchmarks for TM

While several TM systems have been proposed in the literature, there was an urge to develop benchmarks that can correct analyse and compare the proposals. Most TM systems have been evaluated using microbenchmarks, such as linkedlists or red-black trees, which may not be representative of any real-world behaviour, or individual applications, which do not stress a wide range of execution scenarios. Consequently, it has been argued that non-trivial, or realistic, benchmarks are needed to further TM research and to present the "real" benefits of TM.

Some desirable features of non-trivial TM benchmarks are the existence of large amounts of parallelism, executed by code difficult to parallelize using locks, which makes TM more attractive than fine-grain locking. The existence of real-world applications are also desirable because it gives confidence to programmers to use it. Several types of workloads can stress the TM implementations, from long to short transactions, high to low level of contention and different sizes of read and write-set.

STMBench7 Guerraoui et. al proposed STMBench7 [24], a benchmark for evaluating STM implementations. The underlying data structure of STMBench7 consists of a set of graphs and indexes intended to be suggestive of more complex applications like CAD, CAM and CASE. A collection of operations is supported to model a wide range of workloads and concurrency patterns. There are *long traversals*, which go through all the assemblies and/or all atomic parts and update some of them. Second, there are *short traversal* that traverse via a randomly chosen path. Third, there are *short operations*, which randomly chose some object (or a few objects) in the structure and perform an operation on the object or it's local neighbourhood. Finally, there are *structure modification operations*, which are operations that create or delete elements of the structure or links between elements randomly.

STMBench7 is likely to benefit STM over HTM and HyTM, as transactions will likely have larger read and write-sets to keep track of objects visited, thus causing capacity aborts on HTM. STM is expected to perform better as the size of read and write-set does not degrade the performance of it.

Lee-TM Ansari et al. proposed Lee-TM [7], a benchmark based on Lee's routing algorithm, which consists in circuit routing, the process of automatically producing an interconnection between electronic components. This is achieved based on two phases of the algorithm, *expansion* and *backtracking*. Lee-TM has five implementations of Lee's routing algorithm, which generates a very heterogeneous workload encompassing a wide range of transactions' duration and length. The benchmark starts by routing the shortest junction in the circuit, generating transactions whose local processing lasts just a few milliseconds. HTM is highly probable to have better performance in this first phase than STM as transactions are shorter, and so the read and write-set, which lead to overheads due to instrumentation in STM that are avoided by the pure hardware execution of HTM. The benchmark progressively lays junctions of increasing length, generating workloads whose local processing lasts up to a few seconds, which in turn is likely benefit STM over HTM, due to the capacity aborts of HTM incurred by the size of the read and write-set.

STAMP Minh et al. proposed STAMP [31], a benchmark suite, which enable the analysis of a wide range of TM systems through the use of a wide range of transactional characteristics such as transaction lengths and sizes of read and write-sets.

STAMP consists in eight applications with 30 different sets of configurations and input data that recreate applications from diverse domains, such as ecommerce, Delaunay triangulation, graph processing and circuit routing. The workloads produced by the benchmarks variate from short to long transactions, small to large read and write-set, low to high time spent on the transaction and different levels of contention.

STAMP is portable across different TM implementations, i.e. HTM, STM and HyTM. However, STAMP generally does not work as expected on existing HTM, as it causes nonessential transaction aborts in some of the programs, which motivated researchers to propose a modified version that can fairly compare the intrinsic performance of HTM systems [32].

Memcached Memcached [37] is a distributed in-memory software cache, where multiple clients place key-value pairs on multiple servers. Memcached uses an *in-memory* hash table that stores key-value pairs. It provides operations such as *get*,*replace*,*delete* and *compare-and-swap* on these keys. Clients can also request the current status cache, and related statistics.

Tests made by Holla et al. [26] indicate that cache locks and stats have high contention but not the data, and propose the use of *lock elision* [16,26]. Authors find that *lock elision* can provide non-trivial benefits to both power and performance, although not all hardware configurations were beneficial due to the existence of false conflicts.

Memcached workload is characterised by small transactions, which should benefit HyTM and HTM, as normally transactions read and write-set fit in cache.

Kyoto Cabinet Kyoto Cabinet [1] is a benchmark suite of Data Base Management data stores written in C++ and produced by Fal Labs. The in-memory component of the suite, Kyoto CacheDB, splits the database into slots, where each slot is a hash table of binary search trees. Each key is hashed into a slot and then hashed again into a hash table of the slot. Kyoto CacheDB first fills the database to a fixed initial size, and then executes operations, e.g. gets, puts or deletes, with random keys. This benchmark will be likely to benefit STM and HyTM, as with the available operations (gets, puts and deletes) it will search through the index and increase the length of the transaction, and also the size of read and write-set, which consequently cause HTM to abort.

3 Solution architecture

The analysis of the state of the art conducted in the previous section highlights that in the literature there are no solutions that can efficiently handle various types of workloads. HTM has good performance for short transactions with small read and write-set, however it's performance degrades when transaction size increase. STM has good performance for workloads characterised by long transactions with various types of read and write-set yet, in presence of transactions with small length it's performance is inferior than HTM. HyTM has non-negligible costs incurred by synchronising HTM and STM in order to work as a coherent whole. Consequently, the performance of HyTM is not as good as HTM for short transactions and not as good as STM for long transactions.

This dissertation aims to fill this relevant gap by proposing a dynamic memory partitioning mechanism that will allow HTM and STM to execute concurrently on disjoint memory partitions without incurring any instrumentation overhead.

Preliminary tests were made in order to study the execution of HyTM on disjoint data with no synchronisation over HTM and STM. The HyTM implementation used is composed by Intel TSX (see section 2.2) and TinySTM (see section 2.1) with no synchronisation done between the two backends. A synthetic benchmark was generated, composed of two disjoint linked lists, one shorter (prone to benefit HTM) and one larger (prone to benefit STM). Whenever an operation is issued to the shorter linkedlist, it is used the HTM implementation to handle the execution of the operation. And similarly, whenever an operation is issued to the larger linkedlist, STM executes the operation. The same experiment was made with state of the art Hybrid NOrec [11], Intel Haswell using a single global lock as the fallback path, TinySTM [20] and NOrec [12].

The results of all experiments are shown on Figure 1 and on Figure 2, on the throughput side the prototype scaled better than the other TMs, as longer transactions are handled by the STM fallback (TinySTM), and the shorter by HTM. TinySTM had the second best performance, due to it's scalability and the design approach of reducing instrumentation. NOrec had better performance than both HTM-SGL and Hybrid-NOrec, as HTM-SGL has overheads due to capacity aborts incurred. Hybrid-NOrec had worse performance due to the read of the global clock by transactions, which causes transactions to abort upon commit of any transaction. The results of the energy spent on the execution of the workload show that the prototype consume less energy on the execution of the workload than the other TM implementations. HTM had similar energy consumption to the prototype until 2 threads, however with the increase of the number of threads, the energy consumption increases. TinySTM has the second lowest energy consumption at both four threads and eight followed by NOrec. Hybrid-NOrec had the highest consumption of energy because of high abort-rate, mainly due to the read of the global clock.

As discussed in Section 2.1, the idea of partitioning memory and to use different TM algorithms has first been introduced by Riegel et al. [35]. This work has showed that for several STAMP benchmarks, memory can be **statically** partitioned in an effective way, i.e., so to use in each partition different software TM algorithms tailored for different workload characteristics.

However, this work has also highlighted that static memory partitioning can often be inadequate, in particular with applications that use pointers extensively (as it is often the case in practice). Further, this work considered, as alternative TM mechanism, only software-based implementations. This leaves open the question of whether approaches based on memory partitions remain practically viable also when considering hybrid TM systems encompassing both hardware and software implementations.

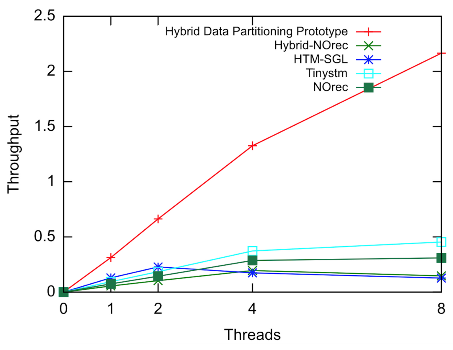

Figure 1: Throughput comparison on the synthetic benchmark

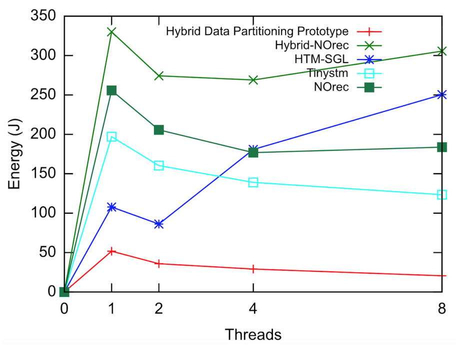

Figure 2: Energy comparison on the synthetic benchmark

In my dissertation, I plan to address these shortcomings by proposing a dynamic memory partitioning mechanism specifically targeted to tackle the challenges raised by the need of supporting concurrent execution of hardware and software TM implementations.

The idea is to exploit the output of an initial static code analysis to determine an initial set of disjoint memory partitions, analogously to what is proposed by Riegel et al. [35]. Unlike in this solution, though, the assignment of data partitions to different TM algorithms does not require to be perfect. Conversely, accesses to memory partitions not assigned to a given TM mechanism shall be detected at run-time and trigger either: a) the reassignment of the transaction to a different TM implementation (in case one exists given the current memory partitioning scheme), or b) the alteration of the memory partitioning scheme and the remapping of the accessed memory region (in case this is deemed as beneficial for the system as a whole), or c) the switch to a state of the art, conventional HyTM implementation, which can support concurrent execution of both HTM and STM, although with additional overheads/instrumentation.

One key challenge that shall be addressed in order to make the proposed solution viable in practice is how to detect illegal accesses to shared memory regions in an efficient way, i.e., without requiring checking additional metadata. This could be very problematic in particular for HTM, which would consume valuable cache capacity just to load these additional metadata (as discussed in Section 2.3).

The key idea that I plan to exploit to tackle this issue is to exploit conventional virtual memory protection mechanisms in an innovative way. The idea is to partition threads into two classes, one allowed solely to use HTM and one STM. In order to enforce access of each class of threads to their assigned memory regions, the *mprotect* system call will be used: this system call can be used to assign different access rights to different threads (within the same process) at the page granularity. This way, any violation of the intended memory partitioning scheme will be detected at the operating system level, by exploiting the dedicated hardware mechanisms for memory protection, without requiring any additional application level checks. The runtime library will be notified of possible violations of the memory partitioning via a SIGSEV signal, and will be able to react implementing the different reconfiguration strategies sketched above.

Clearly, there are a set of challenges and of potential pitfalls involved with this design that shall be assessed and evaluated during the following phases of my work.

The first challenge is associated with the costs associated with the usage of system calls to dynamically redefine the memory mapping, as well as of signal handlers to react to violations of the expected memory partitioning.

Another relevant issue that deserves further study is how to avoid the risk of frequent "ping-ponging" memory pages between different partitions. In fact, reconfiguring the memory partitions has a non-negligible cost, given that it entails the execution of one or more system calls. The dynamic partitioning scheme should therefore minimise the frequency of relocation of a given memory page. Conversely, when such problematic situations are detected, the system should fallback to using a conventional hybrid TM. The challenge here lies in detecting such scenarios in an timely, yet efficient way, as well as to support the fast switch to the fallback synchronisation mechanism.

4 Evaluation

The proposed system will be evaluated and compared to state of the art TM implementations, namely Intel TSX, TinySTM [20], SwissTM [19], NOrec [12], Hybrid NOrec [11], Hybrid-LSA [36], Reduced Hardware NOrec [29] and coarse/fine-grained locking solutions. The key metrics that will be used to evaluate the solutions are the following :

- Throughput
- Abort Rate
- Response Time
- Energy Consumption
- Energy Delay Product
- Exponential Energy Delay Product
- Reconfiguration Latency

Throughput and Abort Rate are used to measure respectively the number of transactions committed and aborted per second, this metrics are indicators of the TM's performance, as less aborted transactions and more committed transactions represent a primary goal of this work. Response Time represents the time spent by a transaction since the beginning until the commit. Energy Consumption is measured via Intel's RAPL [14] interface and represents the Joules consumed by the execution of the solution; Energy Delay Product, as the name suggests, is the product of energy by time expended in execution; Exponential Energy Delay Product, or ExpEDP is instead given by the product of the energy spent times the square of the execution time, and, hence, gives more emphasis to time.

The system will be tested in a wide variety of workloads, from short to long transactions, high to low contention, and various types of read and write-sets sizes. Different number of threads will be used and different hardware, namely Intel Haswell and IBM Power8, to stress the solution and highlight the advantage of running different TMs on disjoint data partitions.

In the evaluation of the implementation it will be used the redblack tree and linkedlist microbenchmarks, as well as the following benchmarks: STMBench7 [24], Lee-TM [7], STAMP [31], Memcached [26] and Kyoto Cabinet [1].

5 Scheduling of Future Work

- 15 January - 15 February: Implementation of a preliminary prototype that uses a simple initial static partitioning, and exploits the *mprotect* system call to detect violations of the expected partitioning. Also, porting of the microbenchmarks to use the APIs exposed by current prototype.
- 15 February - 15 April: Extension of the preliminary prototype to support dynamic remapping of the memory regions between HTM and STM depending on the applications access pattern. Also, benchmarking and profiling of the costs associated with the use of system calls to dynamically re-map the memory regions to different threads.
- 15 April - 15 June: Porting of STAMP benchmarks and of Kyoto Cabinet to use the APIs exposed by the current prototype. Also, analysis of the effectiveness of the partitioning, via either static and dynamic methods, of the STAMP benchmarks and investigation and experimentation of various dynamic memory partitioning schemes.
- 15 June - 15 August: Integration with fallback path based on a conventional Hybrid TM. Also Profiling, benchmarking and performance optimisations of the solution.
- 1 September - 1 October: Thesis and paper preparation.

6 Conclusions

In this document I conducted a study on state of the art of Transactional Memory, focusing on the problem of enhancing efficiency of HyTM. The analysis of the state of the art highlighted different approaches on implementing TM: software (STM), Hardware (HTM) and a hybrid of hardware and software (HyTM). My analysis highlighted that none of these approaches successfully delivers a constant performance for all types of workloads, i.e. HTM has high performance for workloads characterised by short transactions with read and write-sets that can fit in cache, yet STM has high performance for long transactions with various types of read and write-sets, while HyTM has a performance in between of HTM and STM but with the incurring costs of synchronising HTM and STM.

During the next phase of my dissertation I aim at addressing precisely this issue as in this document I have already identified some of the key design choices and challenges that I will have in implementing a hybrid solution that can partition data and execute HTM and STM without any instrumentation cost. Further, I have described a detailed roadmap of my future research activities.

References

1. http://fallabs.com/kyotocabinet/
2. z/Architecture Principle of Operations. SA22-7832-09
3. Power ISA$^{\mathrm{TM}}$ Version 2.07 (2013), https://www.power.org/wp-content/uploads/2013/05/PowerISA_V2.07_PUBLIC.pdf
4. Adl-tabatabai, E.A.r., Shpeisman, T., Gottschlich, J.: Draft Specification of Transactional Language Constructs for C ++. Intel Manual pp. 1–35 (2012)
5. Advanced Micro Devices, Inc.: Advanced Synchronization Facility - Proposed Architectural Specification, 2.1 edition (Mar 2009)
6. Afek, Y., Levy, A., Morrison, A.: Programming with hardware lock elision. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 295–296. PPoPP '13 (2013)
7. Ansari, M., Kotselidis, C., Jarvis, K., Luján, M., Kirkham, C., Watson, I.: Lee-tm: A non-trivial benchmark for transactional memory. In: ICA3PP '08: Proceedings of the 7th International Conference on Algorithms and Architectures for Parallel Processing. LNCS, Springer (June 2008)
8. Calciu, I., Gottschlich, J., Shpeisman, T., Pokam, G., Herlihy, M.: Invyswell: A hybrid transactional memory for haswell's restricted transactional memory. In: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation. pp. 187–200. PACT '14 (2014)
9. Calciu, I., Shpeisman, T., Pokam, G., Herlihy, M.: Improved single global lock fallback for best-effort hardware transactional memory. 9th ACM SIGPLAN Wkshp. on Transactional Computing (2014)
10. Click, C.: Azul's experiences with hardware transactional memory. In: HP Labs' Bay Area Workshop on Transactional Memory (2009)
11. Dalessandro, L., Carouge, F., White, S., Lev, Y., Moir, M., Scott, M.L., Spear, M.F.: Hybrid norec: A case study in the effectiveness of best effort hardware transactional memory. SIGPLAN Not. 46(3), 39–52 (Mar 2011)
12. Dalessandro, L., Spear, M.F., Scott, M.L.: NOrec: streamlining STM by abolishing ownership records. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP) pp. 67–78 (2010)
13. Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., Nussbaum, D.: Hybrid transactional memory. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 336–346. ASPLOS XII (2006)
14. David, H., Gorbatov, E., Hanebutte, U.R., Khanna, R., Le, C.: Rapl: Memory power estimation and capping. In: Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design. pp. 189–194. ISLPED '10 (2010)
15. Dice, D., Harris, T.L., Kogan, A., Lev, Y., Moir, M.: Hardware extensions to make lazy subscription safe. CoRR abs/1407.6968 (2014)
16. Dice, D., Kogan, A., Lev, Y., Merrifield, T., Moir, M.: Adaptive integration of hardware and software lock elision techniques. In: Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures. pp. 188–197. SPAA '14 (2014)

17. Dice, D., Lev, Y., Moir, M., Nussbaum, D., Olszewski, M.: Early experience with a commercial hardware transactional memory implementation. Tech. rep., Mountain View, CA, USA (2009)
18. Diegues, N., Romano, P., Rodrigues, L.: Virtues and limitations of commodity hardware transactional memory. In: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation. pp. 3–14. PACT '14 (2014)
19. Dragojević, A., Guerraoui, R., Kapalka, M.: Stretching transactional memory. ACM SIGPLAN Notices 44, 155 (2009)
20. Felber, P., Fetzer, C., Riegel, T.: Dynamic performance tuning of word-based software transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 237–246. PPoPP '08 (2008)
21. Goel, B., Titos-Gil, R., Negi, A., McKee, S.A., Stenstrom, P.: Performance and energy analysis of the restricted transactional memory implementation on haswell. In: Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium. pp. 615–624. IPDPS '14 (2014)
22. Gottschlich, J.E., Vachharajani, M., Siek, J.G.: An efficient software transactional memory using commit-time invalidation. In: Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization. pp. 101–110. CGO '10 (2010)
23. Guerraoui, R., Kapałka, M.: Opacity: A correctness condition for transactional memory. Tech. Rep. LPD-REPORT-2007-004, EPFL (Aug 2007)
24. Guerraoui, R., Kapalka, M., Vitek, J.: Stmbench7: A benchmark for software transactional memory. In: Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007. pp. 315–324. EuroSys '07 (2007)
25. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proceedings of the 20th Annual International Symposium on Computer Architecture. pp. 289–300. ISCA '93 (1993)
26. Holla, A., Herlihy, M.: Lock Elision for Memcached: Power and Performance analysis on an Embedded Platform. pp. 1–9 (2013)
27. Hong, S., Oguntebi, T., Casper, J., Bronson, N., Kozyrakis, C., Olukotun, K.: Eigenbench: A simple exploration tool for orthogonal tm characteristics. In: Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10). pp. 1–11. IISWC '10 (2010)
28. Matveev, A., Shavit, N.: Reduced hardware transactions: A new approach to hybrid transactional memory. In: Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures. pp. 11–22. SPAA '13 (2013)
29. Matveev, A., Shavit, N.: Reduced hardware norec: A safe and scalable hybrid transactional memory. In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 59–71. ASPLOS '15 (2015)
30. Menon, V., Balensiefer, S., Shpeisman, T., Adl-Tabatabai, A.R., Hudson, R.L., Saha, B., Welc, A.: Practical weak-atomicity semantics for java stm. In: Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures. pp. 314–325. SPAA '08 (2008)
31. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: Stamp: Stanford transactional applications for multi-processing. In: IISWC. pp. 35–46. IEEE Computer Society (2008)
32. Nakaike, T., Odaira, R., Gaudet, M., Michael, M.M., Tomari, H.: 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. pp. 144–157. ISCA ’15 (2015)
33. Pankratius, V., Adl-Tabatabai, A.R.: A study of transactional memory vs. locks in practice. In: Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures. pp. 43–52. SPAA ’11 (2011)
34. Riegel, T., Felber, P., Fetzer, C.: A lazy snapshot algorithm with eager validation. In: Proceedings of the 20th International Conference on Distributed Computing. pp. 284–298. DISC’06 (2006)
35. Riegel, T., Fetzer, C., Felber, P.: Automatic data partitioning in software transactional memories. In: Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures. pp. 152–159. SPAA ’08 (2008)
36. Riegel, T., Marlier, P., Nowack, M., Felber, P., Fetzer, C.: Optimizing hybrid transactional memory: The importance of nonspeculative operations. In: Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures. pp. 53–64. SPAA ’11 (2011)
37. Ruan, W., Vyas, T., Liu, Y., Spear, M.: Transactionalizing legacy code: An experience report using gcc and memcached. In: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 399–412. ASPLOS ’14 (2014)
38. Spear, M.F., Marathe, V.J., Dalessandro, L., Scott, M.L.: Privatization techniques for software transactional memory. In: Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing. pp. 338–339. PODC ’07 (2007)
39. Wang, A., Gaudet, M., Wu, P., Amaral, J.N., Ohmacht, M., Barton, C., Silvera, R., Michael, M.: Evaluation of blue gene/q hardware support for transactional memories. In: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques. pp. 127–136. PACT ’12 (2012)
40. Yoo, R.M., Hughes, C.J., Lai, K., Rajwar, R.: Performance evaluation of intel® transactional synchronization extensions for high-performance computing. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 19:1–19:11. SC ’13 (2013)