Ensuring dependability and improving performance of transactional systems deployed on multi-core architectures

Mohamed Mohamedin

Preliminary Examination Proposal submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Binoy Ravindran, Chair
Leyla Nazhandali
Mohamed Rizk
Paul Plassmann
Robert P. Broadwater
Roberto Palmieri

November 3, 2014
Blacksburg, Virginia

Keywords: Transaction Memory, Hardware Transaction Memory (HTM), Best-efforts HTM, Transactions Partitioning, Transactions Scheduling, Transient Faults, Soft-errors, Fault-tolerance

Ensuring dependability and improving performance of transactional systems deployed on multi-core architectures

Mohamed Mohamedin

(ABSTRACT)

The industrial shift from single core processors to multi-core ones introduced many challenges. Among them, a program cannot get a free performance boost by just upgrading to a new hardware because new chips include more processing units but at the same (or comparable) clock speed as the previous one. In order to effectively exploit the new available hardware and thus gain performance, a program should maximize parallelism. Unfortunately, parallel programming poses several challenges, especially when synchronization is involved because parallel threads need to access the same shared data. Locks are the standard synchronization mechanism but gaining performance using locks is difficult for a non-expert programmers. A new, easier, synchronization abstraction is required and Transactional Memory (TM) is the potential candidate.

TM is a new programming paradigm that simplifies the implementation of synchronization. The programmer just defines atomic parts of the code and the underlying TM system handles the required synchronization. In the past decade, TM researchers worked extensively to improve TM-based systems. Most of the work has been dedicated to Software TM (STM) as it does not requires special transactional hardware supports. Very recently (in the past two years), those hardware support have become commercially available in commodity processors. Hardware TM (HTM) provides the best performance of any TM-based systems, but current HTM systems are best-effort, thus transactions are not guaranteed to commit. In fact, HTM transactions are limited in size and time as well as prone to livelock at high contention levels.

Another challenge facing multi-core architectures is dependability. In the hardware design, factors such as increasing number of components on the same chip, reducing transistor's size, and reducing the operating voltage cooperate to increase the rate of soft-errors. Soft-errors are transient faults that can randomly flip registers and memory (likely cache) bits. Multi-core architectures meet all aforementioned factors, thus they are prone to soft-errors.

In this dissertation, we tackle the performance and dependability challenges of multi-core architectures by providing two solutions for increasing performance using HTM (i.e., Part-htm and Octonauts), and two solutions for solving soft-errors (i.e., Shield and SoftX).

Part-htm is an innovative Hybrid TM system that alleviates HTM resource limitation. A large transaction is partitioned into smaller partitions that fit in HTM. Thus, large transactions can achieve high performance by exploiting HTM speed. Transaction partitioning poses many correctness challenges which are handled by Part-htm. Results in benchmarks where transactions are mostly aborted for resource failures confirmed Part-htm effectiveness. In those benchmarks Part-htm is the best, gaining up to 74%.

Octonauts tackles the live-lock problem of HTM at high contention level. HTM lacks of advanced contention management (CM) policies. Octonauts is an HTM-aware scheduler that orchestrates conflicting transactions. It uses a priori knowledge of transactions' working-set to prevent the activation of conflicting transactions, simultaneously. Octonauts also accommodates both HTM and STM with minimal overhead by exploiting adaptivity. Based on the transaction's size, time, and irrevocable calls (e.g., system call) Octonauts selects the best path among HTM, STM, or global locking. Results show performance improvement

when OCTONAUTS is deployed in comparison with pure HTM with falling back to global locking.

SHIELD is a fault-tolerant system for transactional application using state-machine replication (SMR). It is optimized for centralized multi-core architectures specially message-passing based ones. It includes an efficient network layer for ordering transactional requests with optimistic delivery support. The other main component of SHIELD is the concurrency control. It supports concurrent execution of transactional requests while forcing the network layer order. SHIELD is also able to speculatively start transactions as soon as the request is optimistically delivered. SHIELD's results show limited overhead in comparison with non-fault-tolerant transactional systems and improvement over other non-optimized SMR approaches.

SOFTX is a low-invasive protocol for supporting execution of transactional applications relying on speculative processing and dedicated committer threads. Upon starting a transaction, SOFTX forks a number of threads running the same transaction, independently. The commit phase is handled by dedicated threads for optimizing synchronization's overhead. Results reveal better performance than classical replication-based fault-tolerant systems and limited overhead with respect to non fault-tolerant protocols. In comparison with SHIELD, SOFTX does not requires computational resource partitioning nor requests ordering but it has smaller fault coverage.

Our first major post-preliminary research goal is to develop a contention manager for HTM that works based on online information without a priori knowledge of the transaction's accesses. Dropping this knowledge increases the CM accuracy and allows more concurrency. The challenge posed by this goal is to allow the sharing of online information without introducing additional aborts due to the nature of HTM conflict resolution. Our second goal is to add advanced nesting capabilities (i.e., closed and open nesting) to HTM. HTM supports basic flat nesting only and it is limited to 7 levels only. Advanced nesting is known to improve the performance. For example, in closed nesting, aborting a child transaction does not abort the parent transaction. Our third goal is to build a comprehensive framework for Hybrid TM algorithms. In fact, analyzing the current literature of Hybrid TM algorithms, we can identify several common building blocks used to build those algorithm. Our proposed framework makes the implementation of hybrid TM algorithms easy. It aims for helping the research community in developing more hybrid TM algorithms, deploying extensive comparison studies and allowing the composability of different hybrid TM algorithms according to the transaction's characteristics.

This work is supported in part by US National Science Foundation under grants CNS 0915895, CNS 1116190, CNS 1130180, and CNS 1217385.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Multi-core architectures are the current trend. They are everywhere from super computers to mobile devices. The future trend is to increase the number of cores in each CPU and enhance the communication speed between cores as well as the number of CPU sockets. Without exploiting parallelism, a program cannot gain more performance when deployed on an upgraded hardware, which includes more cores with a clock speed that is similar to the previous hardware. Unfortunately, multi-core programming is an advanced topic, often suited only for expert programmers. Therefore, in order to gain more performance from current and emerging multi-core systems, we need an easy, transparent and efficient mechanism for programming them, a mechanism that allows more programmer to do concurrent programming efficiently and correctly. An effective abstraction is Transactional Memory (TM) [54, 51, 99, 63, 35]. TM programming model is as easy as coarse-grained locking. The entire critical section is marked as a transaction instead of guarding it with a single lock. Coarse-grained locking serializes all threads accessing the same critical section and it does not allow concurrent access. On the contrary, TM allows more concurrency. As long as transactions are not conflicting, they are allowed to run and commit concurrently. Two transactions conflict only when they access the same object and one of the operation is a write. TM also guarantees atomicity, consistency, and isolation. Thus, a transaction is executed all or nothing, always observing a consistent state, and it works in isolation from other transactions.

In terms of performance, TM is prone to have comparable performance to fine-grain locking, which in principle uses the minimal number of locks needed for preserving the correctness of the program. Unfortunately, fine-grain locking cannot be composable, e.g., two atomic blocks implemented using fine-grain locking cannot be naively put together in a single transaction because the resulting transaction is likely not atomic and isolated from other concurrent accesses. The TM's abstraction solves this problem, thus providing composability.

TMs are classified as *software* (STM) [38, 33], which can be executed without any transactional hardware support, *hardware* (HTM) [54, 51, 26], which exploits specific hardware

facilities, and *hybrid* (HyTM) [63, 35], which mixes both HTM and STM.

TM has been studied extensively for the past decade [54, 51, 99, 55, 105, 93, 61, 53, 17, 101, 39, 38, 33]. Most of the research efforts were directed towards STM. STM systems run on any hardware and do not requires any transactional hardware support. On the contrary, commercial Hardware TM support was lately introduced in the past two years. Before that, all HTM research was done on simulators or very specialized hardware. On the other hand, STM research continued to cover more scenarios (e.g., performance tuning, contention management, scheduling, nesting, semantic aware STM). STM gained industry traction starting by Intel C++ STM compiler [57], followed by including TM constructs in C++11/C1X new standards, and finally adopted by the famous GCC compiler starting from version 4.7.

Hardware TM is recently available commercially now in commodity CPUs (i.e., Intel Haswell) and HPC (i.e., IBM Blue Gene/Q and IBM Power 8 [18]). Both Intel's and IBM's HTMs are *best-efforts* HTM: a transaction is not guaranteed to commit, even if it runs alone. A fallback path must be provided by the programmer for transactions that fail in HTM. The default fallback path is to acquire a global lock. But, more advanced techniques have been recently introduced to improve HTM performance and overcome HTM limitations [19, 3, 20, 75, 76, 41, 2]. Unfortunately, the well studies STM techniques cannot be ported directly to HTM or Hybrid TM. With many high performance STM algorithms (e.g., TL2), the performance is highly degraded compared to both plain HTM and plain STM [94, 75]. Thus, Hybrid TM algorithms requires careful design to balance the added overhead on both HTM and STM components.

Another issue with multi-core architectures is dependability. Multi-core architectures manufacturing trend is to add more cores on the same chip and reduce transistors size. These two factors (i.e., smaller transactions and more components on the same chip) are shown to be responsible for an increased rate of soft-errors [16, 11]. Soft-errors are transient faults that may happen anytime during application execution. They are caused by physical phenomena [11], e.g., cosmic particle strikes or electric noise, which cannot be directly managed by application designers or administrators. As a result, when a soft-error occurs, the hardware is not affected by interruption, but applications may crash or behave incorrectly. A soft error can cause a single bit in a CPU register to flip (i.e., residual charge inverting the state of a transistor). Most of the time, such an event is likely to be unnoticed by applications because they do not use that value (e.g., unused register). However, sometimes, the register can contain an instruction pointer or a memory pointer. In those cases, the application behavior can be unexpected.

As a result, multi-core architectures are prone to transient faults (e.g., soft-errors), thus applications with dependability requirements (i.e., fault-tolerance, availability and reliability) cannot be directly migrated into a multi-core architecture without providing them with a mechanism for addressing transient faults . Current fault tolerance techniques are not optimized for multi-core architectures, whereas they are designed for distributed systems where

communication delays allow for special types of optimizations. Communication delays in multi-core architectures are very limited. Other fault tolerance techniques are designed to guarantee data correctness without liveness (e.g., checkpointing and high availability).

Motivated by these observations, this thesis focuses on two aspects: improving multi-core performance by exploiting best-efforts HTM while overcoming its limitations; providing multi-core dependability by developing fault tolerant software layer, which is optimized for multi-core architectures.

1.1 Summary of Current Research Contributions

1.1.1 Improving Multi-core Performance Exploiting HTM

Gaining more performance from multi-core architectures requires exploiting concurrency and parallelism. However, having multiple threads that act simultaneously usually means synchronizing their accesses once they want to operate on the same data, otherwise the program's correctness is broken. Coarse-grained locking is easy to program but serialize access to shared data. Hence, it reduces concurrency which affects performance and cannot scale. Fine-grained locking is used to allow more concurrency by fine tuning locking. It uses multiple locks and splits large critical section into smaller fine tuned ones. Fine-grained locking is algorithm based and cannot be generalized. In addition, only expert programmers can handle it due to its complexity. Non-expert programmers can easily face deadlocks, livelocks, lockconvoying, and/or priority inversion. Perhaps, the most significant limitation of lock-based synchronization is that it is not composable. For example, we have a concurrent lock-based hash table. It has two atomic operations (put and remove). Now, we have two of this hash table and we want to atomically remove an entry from one table and add it to the other. Composing add and remove operations cannot guarantee the atomicity of composition. Lock-free synchronization is based on atomic instruction (e.g., *compare-and-swap* (CAS)) but, as the fine-grain locking technique, it is algorithmic specific and hard to generalize.

Transactional Memory (TM) simplify parallel programming to the level of coarse-grained locking while achieving fine-grained locking performance. With the introduction of commercial HTM support, TM performance can finally exceed fine-grained locking performance. Thus, making TM programming model more appealing. The problem with current HTM support by Intel and IBM is being best-efforts HTM, thus a transaction is not guaranteed to commit even if it runs alone with no contention. Also, an HTM transaction has resource limitations (e.g., hardware buffer size, hardware interrupts) which forces the transaction to abort in HTM. Thus, an HTM transaction is limited in size and time. The space limitation is due to limited hardware transactional buffer size, while time limitation is due to the clock tick hardware interrupt that is used by the operating system's scheduler. A fallback path

is required by best-efforts HTM to guarantee progress. The default fallback path is to use a global lock (GL-software path). GL-software path limits concurrency and does not scale for transactions that do not fit in HTM due to resource limitation as it is simply serializing them.

Part-HTM. To tackle best-efforts HTM resource limitations, we propose Part-htm, an innovative transaction processing scheme, which prevents those transactions that cannot be executed as HTM due to space and/or time limitation to fall back to the GL-software path, and commit them still exploiting the advantages of HTM. As a result, Part-htm limits the transactions executed as GL-software path to those that retry indefinitely in hardware (e.g., due to extreme conflicting workloads), or that require the execution of irrevocable operations (e.g., like system calls), which are not supported in HTM.

Part-htm's core idea is to first run transactions as HTM and, for those that abort due to resource limitations, a partitioning scheme is adopted to divide the original transaction into multiple, thus smaller, HTM transactions (called sub-HTM), which can be easily committed. However, when a sub-HTM transaction commits, its objects are immediately made visible to others and this inevitably jeopardizes the isolation guarantees of the original transaction. We solve this problem by means of a software framework that prevents other transactions from accessing objects committed to the shared memory by sub-HTM transactions. This framework is designed to be minimal and low overhead: a heavy instrumentation would annul the advantages of HTM, falling back into the drawbacks of adopting a pure STM implementation. Part-htm uses locks, to isolate new objects written by sub-HTM transactions from others, and a slight instrumentation of read/write operations using cache-aligned signature-based structures, to keep track of accessed objects. In addition, a software validation is performed to serialize all sub-HTM transactions in a single point in time.

With this limited overhead, Part-htm gives performance close to pure HTM transactions, in scenarios where HTM transactions are likely to commit without falling back to the software path, and close to pure STM transactions, where HTM transactions repeatedly fail. This latter goal is reached through the exploitation of sub-HTM transactions, which are indeed faster than any instrumented software transactions. In other words, Part-htm's performance gains from HTM's advantages even for those transactions that are not originally suited for HTM resource failures. Given that, Part-htm does not aim at either improving performance of those transactions that are systematically committed as HTM, or facing the challenge of minimizing conflicts of running transactions. Part-htm has a twofold purpose: it commits transactions that are hard to commit as HTM due to resource failures without falling back to the GL-software path but still exploiting the effectiveness of HTM (leveraging sub-HTM transactions); and it represents in most of the cases an effective trade-off between STM and HTM transactions.

Part-htm ensures serializability [1], the well-known consistency criterion for transaction processing, and, relying on the HTM protection mechanism (i.e., sandboxing), it guarantees that aborted transactions cannot propagate corrupted execution to the software framework.

Octonauts. PART-HTM tackled the problem of HTM resource limitations, but another problem afflicts TM in general, which is handling efficiently conflicts. At high contention levels, transactions keep aborting with each other and can lead to livelock situation. In STM systems, this problem is solved by using a contention manager or a scheduler. A contention manager is consulted whenever two transactions are conflicting. And based on the contention manager rules, the conflict is resolved. For example, a conflict resolution rule can be "older transaction wins". In that case, old transactions are prioritized over newer ones. Thus, old transactions will not starve. A scheduler on the other hand uses information about each transaction and schedule transactions such that conflicting transactions are not scheduled concurrently.

Current Intel HTM implementation has a simple conflict resolution rule where the thread that detect the conflict aborts. This rule does not prevent starvation. In addition, there is no way for the programmer to define conflict resolution rules. Simply, in high contention scenarios, an HTM transaction will face several aborts and then fall back to global locking. Falling back to global locking limits the system's concurrency level and serializes non conflicting transactions, which results in bad performance and scalability.

In order to tackle this problem, we propose OCTONAUTS, an HTM-aware scheduler which aims at reducing conflicts between transactions and providing an efficient STM fallback path. OCTONAUTS basic idea is to use queues that guard shared objects. A transaction first publishes its potential objects that will be accessed during the transaction (working set). That information is provided by the programmer or by a static analysis of the program. Before starting a transaction, the thread subscribes to each object queue atomically. Then, when it reaches the top of all subscribed queues, it starts executing the transaction. Finally, it is dequeued from those queues, allowing the following threads to proceed with their own transactions. Large transactions that cannot fit in HTM are started directly in STM with commit phase as a reduced hardware transaction (RHT) [76]. In order to allow HTM and STM to run concurrently, HTM transactions runs in two modes. First mode is plain HTM, where a transaction runs as a standard HTM transaction. The second mode is initiated once an STM transaction is required to execute. In the second mode, a lightweight instrumentation is used to let the concurrent STMs know about executing HTM transactions. We focused on making this instrumentation transparent to HTM transactions. HTM transactions are not aware of concurrent STM transactions and use objects signature to notify STM transactions about written objects. STM uses concurrent HTM write signatures to determine if its read-set is still consistent. This technique does not introduce false conflicts in HTM transactions. If a transaction is irrevocable, it is stated directly using global locking.

1.1.2 Ensuing multi-core Dependability

In multi-core dependability, we focused on transactional applications. Transactional applications state can only be accessed via transactions. The application state must remain

consistent and fault tolerant. In addition to safety requirements, transactional application must maintain liveness also. In order to achieve these goals we present SHIELD and SOFTX.

Shield. SHIELD is an optimized state-machine replication (SMR) for multi-core architectures. It exploit the emerging message-passing hardware and the local synchronized cycle counter register. SHIELD has two main components, the network layer and the concurrency control. SHIELD's key idea is to partition the available resources, and run the application's transactions on all the partitions in parallel, according to the state-machine replication paradigm. As in state-machine replication, transaction requests are wrapped into network messages that the application submits to an ordering layer. This layer implements the specification of *Optimistic Atomic Broadcast* [59] (OAB), and ensures a global serialization order (GSO) among partitions. Each partition is thus notified with the same sequence of transactions and is able to process those transactions independently from the other partitions. Objects accessed by transactions are fully replicated so that each partition can access its local copy, avoiding further communications. Transactions are processed according to the GSO. Additionally, we use a *voter* for collecting the outcome of transactions and delivering the common response (i.e., majority of the replies) to the application. If one or more replies differ, actions are triggered to restore the consistent status of data in those faulty partitions. Assuming the majority of partitions are correct (i.e., non-faulty), the voter just selects the proper transaction reply to deliver back to the application thread and uses the collected replies that differ from the one delivered to the application for identifying faulty partitions.

With SHIELD, we focus on reducing network protocol latency and optimistically deliver requests in one communication step. We also focused on increasing concurrency in request processing to maximize multicore systems' utilization and reducing ordering layer latency by optimistically executing requests while the final order is established.

SHIELD makes the following contributions: *i)* A new network protocol for ordering requests, that is optimized for centralized systems and that supports optimistic delivery by exploiting the local synchronized cycle counter register; *ii)* An innovative concurrency control algorithm that supports the activation of parallel transactions while preserving a pre-defined commit order; *iii)* A comprehensive evaluation on two different architectures (message-passing-based and bus-based shared memory in Tilera [103] and x86 respectively).

SMR common ordering layers take multiple communication steps to reach consensus. In SHIELD and by exploiting the local synchronized cycle counter register which is available in all modern multi-core architectures, we developed a low latency ordering layer that supports optimistic delivery. SMR requires determinism in executing clients requests. Each replica executes requests independently and to have the same state all the time, replicas must follow the same order. In SHIELD, we developed a new concurrency control that can execute requests concurrently while preserving the ordering layer requests order. It also support optimistic execution of requests (i.e., speculatively) while the ordering layer is determining the final order.

SoftX. SMR on bus-based shared memory architectures (e.g., x86) showed limited scalabil-

ity and high network delays. SMR requires a high network bandwidth while having a single shared bus represented a bottleneck. This observation motivated us to develop low bandwidth fault tolerant transactional framework (SoftX). SoftX is a framework for making a transactional application resilient to transient faults. SoftX's main goal is preventing data corruption caused by a transient fault from propagating in main memory, without a significant impact on application's performance (i.e., 10× worse). This goal is reached by SoftX leveraging on speculative processing [84, 83]. When a transaction is activated, SoftX captures the transaction and forks a number of threads that replicate the execution of the transaction in parallel, speculatively on different cores of the system. We assume an underlying hardware architecture equipped with an enough core count such that the speculative execution does not hamper the application's progress. The outcome of each thread is then compared with the others using dedicated *committer threads*. If it corresponds to the majority outcome, then the transaction can commit and the control is returned to the application. This approach masks transient faults happened during the execution on some core with the speculative execution on other cores.

In SoftX, computational resources and memory are not partitioned and no ordering protocol is required before to process transactions (e.g., [56, 84]) or after for validating transactions (e.g., [88]). Moreover, hardware cores are reused by other transactions after a commit event; they are not preassigned to certain application threads such that the overall system's load is more balanced.

SoftX is compatible with both message-passing-based and bus-based shared memory architectures. Its low synchronization bandwidth requirement leverages its performance on bus-based shared memory architectures as well as message-passing-based ones.

To the best of our knowledge, SoftX is the first attempt to reduce the scope of data corruption to transactional processing without the overhead of classical replication-based solutions. We propose a solution that reduces overhead and increases resources utilization while keeping safety and liveness.

1.2 Summary of Proposed Post-Prelim Work

Our post preliminary work will focus on continuing improving best-efforts HTM performance in different situations, not yet explored in the current literature. In this dissertation we started by overcoming HTM resource limitation and then we tackled HTM high contention scenarios by using scheduling. We will continue tackling HTM high contention problem but using a contention manager which does not require a priori knowledge of transactions' working-set. The new proposal works according to online information collected at encounter time. Without static information, the CM increases its accuracy and allows more concurrency. Instead of overestimating all potential accessed objects, actual accesses only are considered based on the current execution of transactions. The real challenge of this

proposal is in sharing online information without introducing more aborts due to HTM conflicts. HTM transaction runs speculatively and we cannot know extract information from an aborted transaction as all writes are not available outside the HTM transaction and just discarded after an abort.

As a second future direction we propose to provide nesting supports to HTM. Current Intel HTM implementation supports only flat nesting. We plan to extend HTM nesting support by including closed and open nesting in order to alleviate HTM's flat nesting limitations. Closed nesting improves the performance by increasing the concurrency level [80] and providing a lightweight partial abort mechanism. In fact, with closed nesting when a conflict happens during the execution of a child transaction, only that transaction is aborted and retried, rather than the entire transaction. Open nesting improves the concurrency further [80] because children transactions are allowed to commit their changes to the main memory before the commit of the parent transaction. When the parent transaction is aborted, an undo action is required for compensating children transactions' changes.

We plan to merge all our HTM enhancements and build a comprehensive framework for Hybrid TM algorithms. Hybrid algorithms is gaining more traction as it promises better performance than falling back to global locking. In addition, hybrid TM algorithms shares many building blocks. Our proposed framework makes embedding and testing new hybrid TM algorithms easy. It aims also at helping the research community in developing more hybrid TM algorithms, easily comparing them, and adaptively selecting the best one for a certain situation.

1.3 Proposal Outline

The rest of the proposal is organized as follows. Chapter 2 summarize and overview related work. Then we give a background on relevant topics in Chapter 3. PART-HTM details are in Chapter 4 where HTM resource limitations problem is tackled. Chapter 5 describes OCTONAUTS, our HTM-aware scheduler. Then, we move to multi-core dependability in Chapter 6 which describes SHIELD and how we optimized state-machine replication for centralized multi-core architectures. SOFTX and redundant execution details are in Chapter 7. The proposal conclusions and our proposed post-preliminary work are listed in Chapter 8.

Chapter 2

Related Work

2.1 Performance Improvement Using HTM

Research in Hybrid TM (HyTM) [35, 63, 97] started before the recent release of Intel's Haswell processor with HTM support.

PhTM [69] was designed for the Sun's Rock processor, which is a research processor (never been commercially released) that supports HTM. After that, AMD proposed Advanced Synchronization Facility (ASF) [26], which attracted researchers to design the initial Hybrid TM systems [94, 32]. They used ASF support for non-transactional load/store inside HTM transactions to optimize their HyTM proposals. Recently, IBM and Intel released processors with HTM support. On the one hand, IBM's HTM processors are available in Blue Gene/Q and Power8 [18]. On the other hand, Intel released Haswell processors with HTM support.

The release of Haswell processors attracted more research on how to boost Haswell HTM capabilities via software [19, 3, 20, 75, 76, 41, 2]. Haswell is a best-effort HTM and requires a software fall back path. Intel suggested global locking (GL-software path) as a default fall back approach, but, having just one global lock limits parallelism and concurrency. This way, even short transactions are forced to fall back to global locking in high conflict scenarios. This motivated researchers to tackle this problem proposing different approaches.

- Improving the global locking fall back path to increase concurrency [20, 41].
- Using STM as a fall back path in order to reduce conflict between concurrent HTM and STM transactions [94, 32, 19].
- Using reduced hardware transactions where only the STM commit procedure is executed as HTM transaction [76, 75].

More in detail, in [20] authors propose to defer the check of the global lock at the end of an HTM transaction, rather than at the beginning as usual (this process is also called lazy subscription). This approach increases the concurrency but allows HTM transactions to ac-

cess inconsistent states during the execution. The latter problem is (partially) solved relying on the Haswell HTM sandboxing protection mechanism. In [41], a self-tuning mechanism is proposed to decide the best suited fall back path for an aborted transaction.

In [94, 32], a hybrid version of the NOrec [33] algorithm is proposed. NOrec is an algorithm very suitable for being enriched with HTM supports (hybrid approach) as it uses a single global lock to protect the commit procedure. Optimizing the meta-data shared between HTM and STM is the key point to achieve high performance. Currently, Hybrid NOrec is considered as a state-of-the-art hybrid transactional memory. In [19], a hybrid version of InvalSTM [47] is proposed (Invyswell). Invyswell uses different transaction types: a lightweight HTM, an instrumented HTM, an STM, an irrevocable STM, and a global locking transaction. A transaction moves from one type to another based on the current composition of concurrent transactions and taking into account the contention level.

In [76], reduced hardware transactions (RHT) are introduced. Transactions that fail in hardware are restarted in the slow-path, which consists of an STM transaction that rely on a RHT for accomplishing the commit procedure. If a transaction fails in the slow-path it is finally restarted in slow-slow-path where it execute as plain STM transaction. In [75], the RHT idea is extended to NOrec.

Part-htm takes a different direction from the above proposals. Instead of falling back to global locking or STM, we partition transactions that fails in hardware due to resource limitations and execute each partition as sub-HTM transaction. We fall back to global locking only when a transaction can never succeed in HTM (e.g., due to hardware interruption or irrevocable operations) or when the contention between transaction is very high.

The problem of partitioning memory operations to fit as a single HTM transaction is described also in [4]. In this approach authors used HTM transactions for concurrent memory reclamation. If the reclamation operation, which involves a linked-list, does not fit in a single HTM transaction, they split it using compiler supports to make the operation suited. Part-htm differs with [4] because they do not provide a software framework for ensuring consistency and isolation of sub-HTM transactions as we do.

In [3], a similar partitioning approach is used to simulate IBM Power8's rollback-only hardware transaction via Intel Haswell HTM. They solve the opacity problem between hardware transaction and global locking fallback path. The solution is to hide writes that occur in the GL-software path until the end of the critical section without monitoring the reads. They also split the transaction in multiple sub-HTM transactions, and each of them keeps both an undo-log and a redo-log. Before committing, the undo-log is used to restore memory's old values (i.e., hiding the transaction's writes). At the beginning of the next HTM sub-transaction, the redo-log is used to restore the previous sub-transaction values. Following this approach, the undo-log and redo-log keep growing from one sub-HTM transaction to the next, consuming an increasing amount of precious HTM resources. As a result, such an approach is not suited for solving the problem of aborts due to resource failures. In fact, the last sub-HTM transaction will still have a write-set that is as big as the original transaction

before the splitting process.

2.2 Transactional Memory Scheduling

Transactional memory scheduling has been studied extensively in Software Transactional Memory systems [43, 9, 113, 73, 46, 8]. However, TM scheduling for HTM systems has not been explored.

Dragojević et. al. in [46] presented a technique to schedule transactions dynamically based on expected working-sets. In [8], the Steal-On-Abort transaction scheduler is presented. Its idea is to queue aborted transaction behind concurrent conflicting transactions. Thus, prevent them from conflicting again.

Adaptive Transaction Scheduler (ATS) [113] monitors the contention level of the system. When it exceeds a threshold, transactions scheduler takes control. Otherwise, transactions proceeds normally without scheduling.

CAR-STM is presented in [43]. In CAR-STM, each core has its own queue. Potentially conflicting transactions are scheduled on the same core queue to minimize conflicts. In addition, when a transaction is aborted, it is scheduled on the same queue behind the one that conflicted with it. Thus, preventing them from conflicting again in the future.

Proactive Transactional Scheduler (PTS) [14] idea is to proactively schedule transactions before accessing hot spots of the programs. Instead of waiting for transactions to conflict before scheduling them, they are proactively scheduled to reduce contention in the program's hot spots. PTS showed an average of 85% improvement over backoff retry policy in STAMP[21].

In [41], a self-tuning approach for Intel's HTM (Tuner) is presented. The approach is workload-oblivious and does not require any offline analysis or priori knowledge of the program. It uses lightweight profiling techniques. Tuner controls the number of retries in HTM before falling back to global locking. It analyzes a transaction capacity and time to decide the best number of retrials in HTM. This decision is used the next time when the same transaction is executed. If the previous decision does not fit the current run of the transaction, the tuning parameters are reevaluated. Compared to Octonauts, Tuner does not require priori knowledge of the transactions and it is adaptive also. It avoids unnecessary HTM trials for transactions that does not fit in HTM based on its online profiling.

Octonauts is an adaptive scheduler. It is only activated when the contention level is medium to high. It uses a priori knowledge of expected working-set of each transaction to schedule them. Our queues are one per each object. And it allows multiple read-only transactions to proceed concurrently. Octonauts also is HTM-aware scheduler.

2.3 Dependability

Many techniques are proposed to handle transient faults that cause data corruption or *silent* data corruption. All techniques are targeting safety where correct results are the main target. But not all of them guarantee liveness.

Checkpointing is a well known technique in High Performance Computing (HPC). The system state is checkpointed periodically and fault detection mechanisms are installed to detect data corruption and restore the system to a correct system state. A major challenge in these system is determining how many checkpoints are required. If the error is detected long time after it occurred, we may have no valid checkpoint to restore. On the other hand, keeping infinite number of checkpoints is not practical. Checkpointing only guarantees safety and has a downtime to restore a checkpoint and re-execute lost requests. Another problem of checkpointing is the time required to checkpoint the system which can be large if the data is huge. [72] has a good survey about checkpointing techniques.

Other approaches in HPC rely on more complex algorithms. They are known as fault-tolerant algorithms. Some scientific problems can be modified to support error detection and correction using some encoding mechanisms like checksums. These approaches are algorithm specific and they are difficult to generalize. Encoded processing incurs high processing overhead and requires extra memory. In [72] more information about such techniques can be found.

Some less efficient error detection techniques, in terms of error coverage, are bases on assertions/invariants or symptoms [90]. Assertions are conditions that must exist before and after program state change. Symptoms refer to program activities that are usually related to faults. For example, executing an unknown instruction, and misprediction of a high confidence branch. These techniques do not provide full error coverage. Moreover, these techniques only detect errors and require another mechanism for recovery like checkpointing.

Hardening crash-tolerant systems can tolerate transient faults by using error detection mechanisms. When an error is detected, it is converted to a crash fault by taking the node down. Hardening can be applied to replication-based fault tolerant systems that tolerate crash faults. In this case, it supports both safety and liveness. But it depends on the accuracy of the error detection. Also, error detection must detect faults quickly before they can propagate. PASC [30] is an example of hardening techniques.

Byzantine fault tolerant (BFT) systems can tolerate transient faults by default. When a transient fault occurs, the program can continue its execution (i.e., without crash) while producing incorrect behaviors. But BFT systems have a broader scope as they also target malicious activities (e.g., intrusion). As a result, usually their overhead is high.

Most BFT systems are distributed systems and rely on state-machine replication. In state-machine replication, requests must be totally ordered before execution to guarantee reaching

the same state on each replica independently. Each replica executes requests in the same order. Reaching order agreement requires exchanging multiple messages. Some BFT systems execute requests sequentially to guarantee determinism [22, 62]. PBFT [22] is one of the first BFT systems with good performance. It uses batching to improve performance and reduce the number of messages. Zyzzyva [62] reduces the number of messages to the minimum in the gracious execution by using speculation and involving clients in the protocol for detecting faults. Both PBFT and Zyzzyva execute requests sequentially to guarantee determinism. Other BFT systems execute independent requests in parallel to increase system throughput [107, 58, 60]. The key idea is to find independent requests that can run in parallel while forcing dependent requests to run sequentially in the same order in all replicas. In [107], the primary replica executes transactions concurrently. Then a scheduler uses information from primary execution of transactions to drive secondary replicas' execution. It is applicable to 2-phase strict locking systems like some database systems. Eve [58] tries to find independent requests in each batch using a deterministic application-specific mixer. It follows the execute-verify model instead of the agree-on-input model. In [60], operating system transactions of TxOS are used to execute requests and are forced to commit in a predefined order in all replicas. The replica state is limited to kernel-level only. There is no support for general memory transactions or optimistic delivery. In [27], a centralized system is split using isolated virtual machines which represent typical BFT replicas.

Hardware fault-tolerant architectures are more expensive and limited in the number of faults they can tolerate due to the cost of the redundant hardware. For example, NonStop advanced architecture [12] uses two or three replicas. Each replica executes the same instructions stream independently and a voter compares outputs. Hardware resources (memory, disks) are split between replicas and isolated from each other.

Shield is software-based and is transparent from the hardware's specification. It is optimized for centralized systems. It uses optimistic delivery combined with transactional execution to achieve high concurrency. Then, it analyzes transaction outcomes and, when a corruption occurs, a restoration is issued.

State-machine replication is a well known paradigm in transaction processing [82, 59]. These works exploit OAB protocols for processing transactions speculatively, overlapping their execution with the coordination time for ordering. They provide resilience to crash/stop failures; in the case of transient faults, data consistency can be corrupted.

Algorithms for processing uncommitted transactions in-order have been proposed in [82, 10]. However, some of them simply ignore the overhead of implementing meta-data forwarding from speculative threads. This overhead can be alleviated in the presence of higher costs like remote communication. In Shield, delays are smaller, therefore impacting the overall performance. Thus, Shield does not use this technique.

All replication-based fault tolerant systems requires a network with a good bandwidth. All requests are wrapped in a network message and reaching consensus requires some more messages. SoftX's main goals is to minimize the synchronization overhead. For this reason,

we decided to not wrap transaction requests into network messages and total order them. SoftX aims for reducing the overhead of the fault-tolerant protocol without partitioning the machine resources between different replicas and without replicating system memory.

Compared to the above techniques, SoftX is a pure software solution and it inherits both the checkpointing and replication advantages. Using transactional memory to speculatively execute a transaction is similar to taking a checkpoint at the beginning of each transaction. If an error is detected, the execution returns to that checkpoint and retry. SoftX uses also the same principle of replication by executing the same transaction multiple times in parallel. However, SoftX's speculative execution is more lightweight: data are not replicated and there is no fixed number of replicas. Thus, synchronization overhead is minimal and cores can be reused by other threads.

SoftX also supports parallel execution of transactions where multiple thread groups can execute together in the same time. Transactional memory mechanisms coordinate access to shared data. Using dedicated committer threads reduces the synchronization overhead more by minimizing the number of CAS operations. Instead of having all threads competing on the same shared lock, each thread communicates directly with committer threads via a private variable.

Works tackling similar problems (i.e., redundant parallel execution) of SoftX has been presented in [89, 114]. In [89], authors use relaxed determinism which is similar to relaxed memory consistency model in modern processors. However, developer must insert determinism hints when precise ordering is needed. SoftX is transparent to the developer. In [114], Delta Execution is presented. It aims for increasing software reliability using multiple, almost redundant, executions. Two almost identical instances run together and split execution only at different code sections. Execution joins again later. Delta Execution requires modifications to the operating system, compiler, and programming language constructs. SoftX uses transactions which define exactly where parallel speculated execution starts and ends. Moreover, SoftX is implemented as part of an STM library and requires no changes to the operating system, compiler, and programming language constructs.

Transactional applications are increasingly using Software Transactional Memory (STM) algorithms [100, 34, 104, 42] to overcome the programmability and performance challenges of in-memory processing on multicore architectures.

A locking mechanism based on the idea of executing lock-based critical sections in remote threads has been recently explored in [71, 52]. SoftX exploits a similar approach using dedicated committer threads for processing transactions' commit phases.

Chapter 3

Background

3.1 Parallel Programming

Amdahl's law [5] specifies the maximum possible speedup that can be obtained when a sequential program is parallelized. Informally, the law states that, when a sequential program is parallelized, the relationship between the speedup reduction and the sequential part (i.e., sequential execution time) of the parallel program is non-linear. The fundamental conclusion of Amdahl's law is that the sequential fraction of the (parallelized) program has a significant impact on overall performance. Code that must be run sequentially in a parallel program is often due to the need for coordination and synchronization (e.g., shared data structures that must be executed mutual exclusively to avoid race conditions). Per Amdahl's law, this implies that synchronization abstractions have a significant effect on performance.

Lock-based synchronization is the most widely used synchronization abstraction. Coarse-grained locking (e.g., a single lock guarding a critical section) is simple to use, but results in significant sequential execution time: the lock simply forces parallel threads to execute the critical section sequentially, in a one-at-a-time order. With fine-grained locking, a single critical section now becomes multiple shorter critical sections. This reduces the probability that all threads will need the same critical section at the same time, permitting greater concurrency. However, this has low programmability: programmers must acquire only necessary and sufficient locks to obtain maximum concurrency without compromising safety, and must avoid deadlocks when acquiring multiple locks. Moreover, locks can lead to livelocks, lock-convoying, and priority inversion. Perhaps, the most significant limitation of lock-based code is their non-composability. For example, atomically moving an element from one hash table to another using those tables' (lock-based) atomic methods is not possible in a straightforward manner: if the methods internally use locks, a thread cannot simultaneously acquire and hold the locks of the methods (of the two tables); if the methods were to export their locks, that will compromise safety.

3.2 Transactional Memory

Transactional Memory (TM) borrows the transaction idea from databases. Database transactions have been successfully used for a long time and have been found to be a powerful and robust concurrency abstraction. Multiple transactions can run concurrently as long as there is no conflict between them. In the case of a conflict, only one transaction among the conflicting ones will proceed and commit its changes, while the others are aborted and retried. TM transactions only access memory, thus they are "memory transactions".

TM can be classified into three categories: Hardware Transactional Memory (HTM), Software Transactional Memory (STM), and Hybrid Transactional Memory (HyTM). HTM [49, 6, 102, 25, 23] uses hardware to support transactional memory operations, usually by modifying cache-coherence protocols. It has the lowest overhead and the best performance. The need for specialized hardware is a limitation. Additionally, HTM transactions are limited in size and time. STM [99, 55, 105, 93, 61, 53, 17, 81, 101, 39] implements all TM functionality in software, and thus can run on any existing hardware and it is more flexible and easier to change. STM's overhead is higher, but with optimizations, it outperforms fine-grained locking and scales well. Moreover, there are no limitations on the transaction size and time. HyTM [70, 68, 91, 36, 79, 111] combines HTM and STM, while avoiding their limitations, by splitting the TM implementation between hardware and software.

3.2.1 TM Design Classification

TM designs can be classified based on four factors: concurrency control, version control, conflict detection, and conflict resolution[50].

Concurrency Control

A TM system monitors transactions' access to shared data to synchronize between them. A conflict between transactions go through the following events (in that order):

1. A conflict *occurs* when two transactions write to the same shared data (write after write), or one transaction writes and the other reads the same shared data (read after write or write after read).
2. The conflict is *detected* by the TM system.
3. The conflict is *resolved* by the TM system such that each transaction makes progress.

There are two mechanisms for concurrency control: *pessimistic* and *optimistic*. In the pessimistic mechanism, a transaction acquires exclusive access privilege to shared data before

accessing it. When the transaction fails to acquire this privilege, a conflict occurs, which is detected immediately by the TM system. The conflict is resolved by delaying the transaction. These three events occur at the same time.

The pessimistic mechanism is similar to using locks and can lead to deadlocks if it is not implemented correctly. For example, consider a transaction T1 which has access to object D1 and needing access to object D2, while a transaction T2 has access to object D2 and needs access to object D1. Deadlocks such as these can be avoided by forcing a certain order in acquiring exclusive access privileges, or by using timeouts. This mechanism is useful when the application has frequent conflicts. For example, transactions containing I/O operations, which cannot be rolled-back can be supported using this mechanism.

In the optimistic mechanism, conflicts are not detected when it occurs. Instead, they are detected and resolved at any later time or at commit time, but no later than the commit time. During validation, conflicts are detected, and they are resolved by aborting or delaying the transaction.

The optimistic mechanism can lead to livelocks if not implemented correctly. For example, consider a transaction T1 that reads from an object D1, and then a transaction T2 writes to object D1, which forces T1 to abort. When T1 restarts, it may write to D1 causing T2 to abort, and this scenario may continue indefinitely. Livelocks can be solved by using a *Contention Manager*, which waits or aborts a transaction, or delays a transaction's restart. Another solution is to limit a transaction to validate only against committed transactions that were running concurrently with it. The mechanism allows higher concurrency in applications with low number of conflicts. Also, it has lower overhead since its implementation is simpler.

This approach is also know as *deferred updates*.

Conflict Detection

TM systems use different approaches for when and how a conflict is detected. There are two approaches for when a conflict is detected: *eager* conflict detection and *lazy* conflict detection [79]. In eager conflict detection, the conflict is detected at the time it happens. At each access to the shared data (read or write), the system checks whether it causes a conflict.

In lazy conflict detection, a conflict is detected at commit time. All read and written locations are validated to determine if another transaction has modified them. Usually this approach validates during transactions' life times or at every read. Early validations are useful in reducing the amount of wasted work and in detecting/preventing *zombie* transactions (i.e., a transaction that gets into an inconsistent state because of an invalid read, which may cause it to run forever and never commit).

3.3 Intel's HTM

The current Intel's HTM implementation of Haswell processor, also called Intel Haswell Restricted Transactional Memory (RTM) [92], is a best-effort HTM, namely no transaction is guaranteed to eventually commit. In particular, it enforces space and time limitations. Haswell's RTM uses L1 cache (32KB) as a transactional buffer for read and write operations. Accessed cache-lines are marked as "monitored" whenever accessed. HTM synchronization management is embedded into the cache coherence protocol. The eviction and invalidation of cache lines defines when a transaction is aborted (it reproduces the idea of read-set and write-set invalidation of STM).

This way, the cache-line size is indeed the granularity used for detecting conflicts. When two transactions need the same cache-line and at least one wants to write it, an abort occurs. When this happens, the application is notified and the transaction can restart as HTM or can fall back to a software path. The transaction that detects the data conflict will abort. The detection of a conflict is based on how the cache coherence protocol works. We cannot know exactly which thread will detect the conflict as the details of Intel's cache coherence protocol are not publicly available.

In addition to those aborts due to data conflict, HTM transactions can be aborted for other reasons. Any cache-line eviction due to cache depletion or associativity causes the transaction to abort, which means that hardware transactions are limited in space by the size of the L1 cache for its write-set. For a transaction's read-set, Intel's HTM implementation can go beyond the L1 cache capacity using a special hardware reads buffer [29]. For the read-set, the value of the read operation is not required for validation as conflicts can be detected by the object's memory address only. Thus, a cache line eviction from the read-set does not always abort the transaction. Also, any hardware interrupt, including the interrupt from timer, force HTM transactions to abort.

Cache associativity places another limitation on transactional size. Intel's Haswell L1 cache has an associativity of 8. Thus, some transactions accessing just 9 different locations that are mapped to the same L1 cache set (due to associativity mapping rules) will be aborted. In addition, when Hyper-Threading is enabled, L1 cache is shared between the two logical cores on the same physical core.

Intel's HTM programming model is based on three new instructions: `_xbegin`; `_xend`, and `_xabort`. `_xbegin` is used to start a transaction. All operations following the execution of `_xbegin` are transactional and speculative. If a transaction is aborted due to a conflict, transactional resource limitation, unsupported instruction (e.g., CPUID) or explicit abort, then all updates done by the transaction are discarded and the processor returns to non-transactional mode. `_xend` is used to finish and commit a transaction. `_xabort` is used to explicitly abort a transaction.

When a transaction is aborted (implicitly or explicitly), the program control jump to the

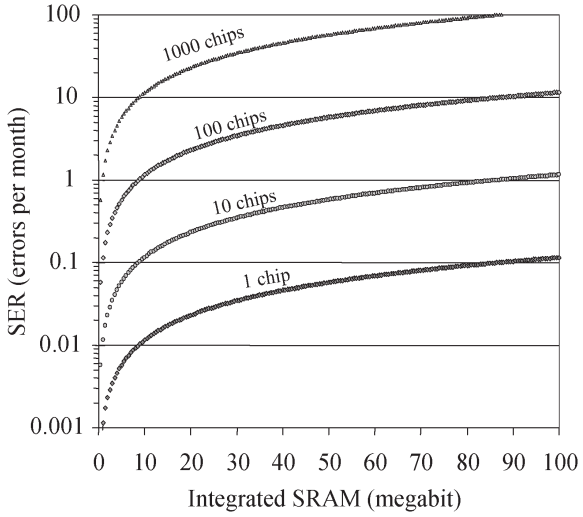


Figure 3.1: Relation between number of components on the same chip and soft-errors rate. (Figure from [11])

abort handler and the abort reason is provided in the EAX register. The abort reason let the programmer know whether the transaction is aborted due to conflict, limited transactional resources, debug breakpoint, or explicit abort. In addition, an integer value can be passed from `_xabort` to the abort handler.

A programmer must provide a software fallback path to guarantee progress. For example, a transaction that faces a page fault interrupt, can never commit in HTM. It will be aborted every time when the interrupt is fired. In addition, the interrupt will not handled as the transaction is running speculatively.

3.4 Soft-errors

Soft-errors [16, 11] are transient faults that may happen anytime during application execution. They are caused by physical phenomena [11], e.g., cosmic particle strikes or electric noise, which cannot be directly managed by application designers or administrators. As a result, when a soft-error occurs, the hardware is not affected by interruption, but applications may crash or behave incorrectly. A soft error can cause a single bit in a CPU register to flip (i.e., residual charge inverting the state of a transistor). Most of the time, such an event is likely to be unnoticed by applications because they do not use that value (e.g., unused register). However, sometimes, the register can contain an instruction pointer or a memory pointer. In those cases, the application behavior can be unexpected.

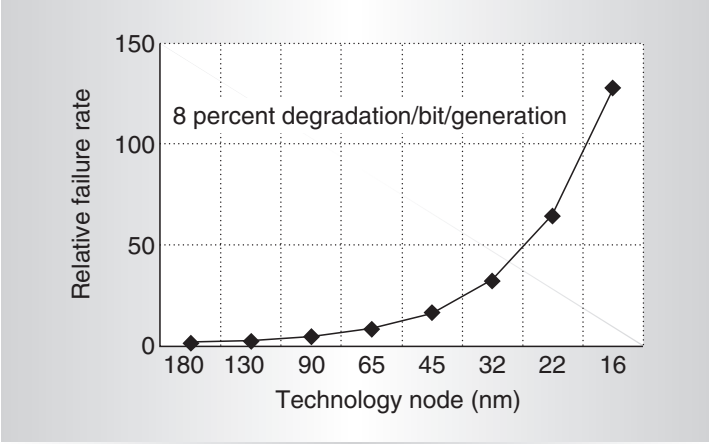


Figure 3.2: Relation between transistor size and soft-errors rate. (Figure from [16])

Figure 3.1 shows an example on the relation of the number of components on the same chip and the soft-errors rate. As the number of the components increases, the soft-error increases. Multi-core architectures are getting more cores on the same chip and the number of transistors is doubled approximately every two years.

Figure 3.2 shows the relation of the transistor size and the soft-errors rate. As the transistor size gets smaller, the rate of soft-errors increase. Currently, Intel uses 22nm manufacturing technology and it is expected to move 16nm technology with the next generation of processors.

3.5 Fault tolerance techniques

In this section, we will go through a number of fault tolerance solution that focus on transient faults including silent data corruption.

3.5.1 Checkpointing

Checkpointing [72] means taking a snapshot of the system state periodically while the system is correct. The system is monitored by a failure detector, which signals when the system is faulty (e.g., detect data corruption). When a fault is detected, the system state is restored to latest correct checkpoint. From the time when the fault is detected until completing the checkpoint restoration, the system is unreachable.

Transient faults represents a challenge for the checkpointing recovery mechanism. The symptom of a transient fault do not appear instantaneously, whereas the transient fault could be detected only after a period of time, which cannot be bound. Thus, the system has to keep a number of checkpoints that covers an arbitrary period of time. Another issue with transient faults is how to distinguish between a correct checkpoint from a faulty checkpoint. Usually a transient fault is detected by its symptoms and it is difficult to identify the source of the transient fault.

3.5.2 Encoding

Some algorithms can be enforced by encoding in software [72]. For example, an additional operation can be encoded such that the result indicates whether an error occurred or not. In hardware, encoding is widely used to detect and correct faults. For example, memory modules equipped with Error-Correcting Code (ECC) can detect and correct errors in the memory module. Based on the encoding technique used, an error can be only detected or also fixed. In addition, the number of detectable/correctable errors is limited.

3.5.3 Assertions/Invariants

This technique [90] can detect faults only without recovery. It is based on monitoring the system for unexpected behaviors (e.g., a change in an invariant). The error's coverage for this technique is however very limited. An assertion is added after each change to the program state to detect faults during that operation. Invariants refer to the program expected behavior, e.g., executing an unknown instruction or a misprediction of a high confidence branch.

3.5.4 Replication

Replicating the system state on multiple isolated replica guarantees full error recovery in case of a replica is hit by a fault. State-machine replication (SMR) is a famous replication technique which is used by many replication fault-tolerant systems (e.g., Byzantine fault tolerant (BFT)). In SMR, request are executed by each replica independently. In order to reach the same final state on each replica, replicas have to execute requests in the same order. Thus, a consensus must be reached on the requests order between all replicas. Based on the fault model, a consensus technique can tolerate permanent faults (e.g., crash) only (e.g., Paxos [65]) or the consensus technique can tolerate malicious behavior also (e.g., the three generals problem [66]).

For permanent faults, one replica response is enough to tolerate a permanent fault. On the other hand, transient and malicious faults requires voting and having a majority to tolerate

a transient fault. Permanent faults consensus requires $2f + 1$ replicas to tolerate f faults, while Byzantine faults consensus requires $3f + 1$ replicas to tolerate f faults. Byzantine fault voting requires only $2f + 1$ replicas to tolerate f faults [112].

Chapter 4

Part-HTM

4.1 Problem Statement

Transactional Memory (TM) [51, 54] is one of the most attractive recent innovations in the area of concurrent and transactional applications. TM is a support that programmer can exploit while developing parallel applications such that the hard problem of synchronizing different threads, which operate on the same set of shared objects, is solved. In addition, in the last few years a number of TM implementations, each optimized for a particular execution environment, have been proposed [38, 33, 32]. The programmer can take advantage of this selection to achieve the desired performance by simply choosing the appropriate TM system. TMs are classified as *software* (STM) [38, 33], which can be executed without any transactional hardware support, *hardware* (HTM) [51, 26], which exploits specific hardware facilities, and *hybrid* (HyTM) [63, 35], which mixes both HTM and STM.

Very recently two events confirmed TM as a practical alternative to the manual implementation of threads synchronization: on the one hand, *GCC*, the famous GNU compiler, embedded interfaces for executing atomic blocks since its version 4.7; on the other hand, Intel released to the customer market the *Haswell* processor equipped with the *Transactional Synchronization Extensions* (TSX) [92], which allow the execution of transactions directly on the hardware through an enriched hardware cache-coherence protocol.

Hardware transactions (or HTM transactions) are much faster than their software version, however, their downside is that they do not have commit guarantees, therefore they may fail repeatedly, and for this reason they are categorized as *best-effort*[1]. The commit of an HTM transaction aborted multiple times is guaranteed through a software execution defined by the programmer (called *fall back path*). The default fall back path consists of executing the aborted transaction protected by a single global lock (called GL-software path). In addition, there are other proposals that take the choice of falling back to a pure STM path [32], as

[1]IBM is approaching to release a similar best-effort HTM processor, i.e., IBM Power8 [18].

well as to a hybrid-HTM scheme [75, 76, 19].

The STM community is investing considerable efforts on developing new efficient and high performance synchronization schemes. However, despite the excellent abstraction, STM implementations need a heavy instrumentation of application source code in order to ensure atomicity and the desired level of isolation. This instrumentation represents an overhead that, often, pays off in specific workloads, rather than in generic scenarios. Conversely, HTM transactions have the strength to ensure high performance because they do not need instrumentation due to the presence of the hardware support, but, unfortunately, current HTM implementations are best-effort, thus high performance are still not ensured in all scenarios.

Leveraging the experience learnt from recent papers on HTM [41, 20, 19, 32], three reasons that force a transaction to abort have been identified: *conflict*, *capacity*, and *other*. Conflict failure occurs when two transactions try to access the same object and at least one of them wants to write it; a transaction is aborted for capacity if the number of cache-lines accessed by all concurrent transactions is higher than the maximum allowed; any extra hardware intervention, including interrupts, is also a cause of abort.

Many recent papers propose solutions to: *i)* handle aborts due to conflict efficiently, such that transactions that run in hardware minimize their interference with concurrent transactions running in the software fall back path [32, 19, 20, 75, 76]; *ii)* tune the number of retries a transaction running in hardware has to accomplish before falling back to the software path [41]; *iii)* modify the underlying hardware support for allowing special instructions so that conflicts can be solved more effectively [110, 7, 26].

Despite this body of work, one of the main unsolved problem of best-effort HTM is that there are transactions that, by nature and due to the characteristics of the underlying architecture, are impossible to be committed as hardware transactions. Examples include transactions that, even accessing few objects, require non minimal execution time and thus they are aborted due to a timer interrupt, which triggers the actions of the OS scheduler (these aborts are indeed classified as "other"); or those transactions accessing several objects, such that the problem of exceeding the cache size arises (known as "capacity" failure). We group these two types of failures into one superset, where, in general, a hardware transaction is aborted if the amount of resources, in terms of space and/or time, which are required to commit, are not available. We name this superset as *resource* failures.

None of the past works targets this class of aborted transactions and we turn this observation into our motivation: solving the problem of resource failures in HTM. To pursue this goal, we propose PART-HTM, an innovative transaction processing scheme, which prevents those transactions that cannot be executed as HTM due to space and/or time limitation to fall back to the GL-software path, and commit them still exploiting the advantages of HTM. As a result, PART-HTM limits the transactions executed as GL-software path to those that retry indefinitely in hardware (e.g., due to extreme conflicting workloads), or that require the execution of irrevocable operations (e.g., like system calls), which are not supported in

HTM.

PART-HTM's core idea is to first run transactions as HTM and, for those that abort due to resource limitations, a partitioning scheme is adopted to divide the original transaction into multiple, thus smaller, HTM transactions (called sub-HTM), which can be easily committed. However, when a sub-HTM transaction commits, its objects are immediately made visible to others and this inevitably jeopardizes the isolation guarantees of the original transaction. We solve this problem by means of a software framework that prevents other transactions from accessing objects committed to the shared memory by sub-HTM transactions. This framework is designed to be minimal and low overhead: a heavy instrumentation would annul the advantages of HTM, falling back into the drawbacks of adopting a pure STM implementation. PART-HTM uses locks, to isolate new objects written by sub-HTM transactions from others, and a slight instrumentation of read/write operations using cache-aligned signature-based structures, to keep track of accessed objects. In addition, a software validation is performed to serialize all sub-HTM transactions in a single point in time.

With this limited overhead, PART-HTM gives performance close to pure HTM transactions, in scenarios where HTM transactions are likely to commit without falling back to the software path, and close to pure STM transactions, where HTM transactions repeatedly fail. This latter goal is reached through the exploitation of sub-HTM transactions, which are indeed faster than any instrumented software transactions. In other words, PART-HTM's performance gains from HTM's advantages even for those transactions that are not originally suited for HTM resource failures. Given that, PART-HTM does not aim at either improving performance of those transactions that are systematically committed as HTM, or facing the challenge of minimizing conflicts of running transactions. PART-HTM has a twofold purpose: it commits transactions that are hard to commit as HTM due to resource failures without falling back to the GL-software path but still exploiting the effectiveness of HTM (leveraging sub-HTM transactions); and it represents in most of the cases an effective trade-off between STM and HTM transactions.

PART-HTM ensures serializability [1], the well-known consistency criterion for transaction processing, and, relying on the HTM protection mechanism (i.e., sandboxing), it guarantees that aborted transactions cannot propagate corrupted execution to the software framework (Section 4.5).

We implemented PART-HTM and assessed its effectiveness through an extensive evaluation study (Section 6.8) including a micro-benchmark, a data structure (i.e., linked-list), as well as applications from the STAMP suite [77]. The first two are important because, through them, we can stimulate PART-HTM using various configurations and explore different workloads. With STAMP we assess the performance of PART-HTM using complex applications. As competitors, we select a pure HTM with GL-software path as a fall back, two STM protocols and a HybridTM. Results confirmed the effectiveness of PART-HTM. In those benchmarks where transactions are mostly aborted for resource failures, PART-HTM is the best, gaining up to 74%. In the other benchmarks, PART-HTM behaves very close to the best competitor,

in almost all tested cases.

In Section 4.4 we describe also how PART-HTM's approach can take advantage of the upcoming best-effort HTM processor (i.e., IBM Power8 [18]) with the new support for suspending and resuming an HTM transaction.

4.1.1 Intel's HTM Limitations

In this section we briefly overview the principles of Intel's HTM transactions in order to highlight their limitations and motivate our proposal. The current Intel's HTM implementation of Haswell processor, also called Intel Haswell Restricted Transactional Memory (RTM) [92], is a best-effort HTM, namely no transaction is guaranteed to eventually commit. In particular, it enforces space and time limitations. Haswell's RTM uses L1 cache (32KB) as a transactional buffer for read and write operations. Accessed cache-lines are marked as "monitored" whenever accessed. This way, the cache-line size is indeed the granularity used for detecting conflicts. When two transactions need the same cache-line and at least one wants to write it, an abort occurs. When this happens, the application is notified and the transaction can restart as HTM or can fall back to a software path.

In addition to those aborts due to data conflict, HTM transactions can be aborted for other reasons. Any cache-line eviction due to cache depletion or associativity causes the transaction to abort, which means that hardware transactions are limited in space by the size of the L1 cache. Also, any hardware interrupt, including the interrupt from timer, force HTM transactions to abort. As introduced in the previous section, we name the union of these two causes as resource limitation and in this chapter we propose a solution for that.

To strengthen our motivation, in Table 4.1 (in the evaluation section) we report a practical case. The table contains statistics related to the Labyrinth application of STAMP benchmark. Here we can see how the sum between the percentage of HTM transactions aborted for *capacity* and *other* forms more than 91% of all aborts, forcing HTM to execute often its GL-software path. This is because more than 50% of Labyrinth's transactions exceed the size and time allowed for an HTM execution.

4.2 Algorithm Design

The basic idea of PART-HTM is to partition a transaction that likely (or certainly) is aborted in HTM (due to resource limitations) into smaller sub-transactions, which could cope better with the amount of resources offered by the HTM.

Despite the simple main idea of partitioning a transaction into smaller hardware sub-transactions, executing them efficiently in a way such that the global transaction's isolation and consistency is preserved poses a challenging research problem. In this section we describe the

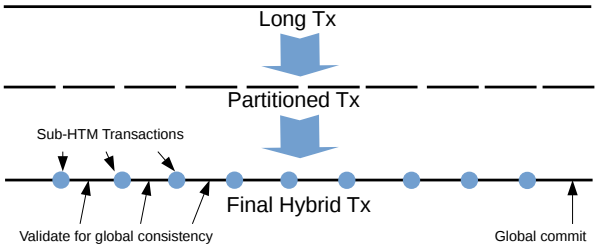


Figure 4.1: PART-HTM's Basic Idea.

design principles that compose the base of PART-HTM, as well as the high level transaction execution flow. The next sections describe the details of the algorithm and its implementation.

Hereafter, we refer to the original (single block) transaction as *global* transaction and the smaller sub-transactions as *sub-HTM transactions*.

A memory transaction is a sequence of read and write operations on shared data that should appear as atomically executed in a point in time between its beginning and its completion, and in isolation from other transactions. This also entails that changes on the shared objects performed by a transaction should not be accessible (visible) to other transactions till that transaction is allowed to commit. The latter point clashes with the above idea: when a sub-HTM transaction T_{S1} of a global transaction T commits, its written objects are applied directly to the shared memory, by nature. This allows other transactions to potentially access these values, thus breaking the isolation of T. Moreover, once T_{S1} is committed, there is no record of its read and written objects during the rest of T's execution, therefore also the correctness of T becomes hard to enforce.

All these problems can be naively solved by instrumenting HTM operations for populating the same meta-data commonly used by STM protocols for tracking accesses and handling conflicts. However, applying existing STM solutions easily leads to loose the effectiveness of HTM and, consequently, poor performance. In the following we point out some of these reasons.

- STM's meta-data are not designed for minimizing the impact on memory capacity. Adopting them for solving our problem would stretch both the transaction execution time and the number of cache-lines needed, thus consuming precious HTM resources;
- the HTM already provides an efficient conflict detection mechanism, which is faster than any software-based contention manager;
- the HTM monitors any memory access within the transaction, including those on the meta-data or local variables. This takes the flexibility for implementing smart contention policy away from the programmer.

PART-HTM faces the following challenge: how to exploit the efficiency of sub-HTM trans-

actions, which write in-place to the shared memory, by minimizing the overhead of the additional instrumentation for maintaing the isolation and the correctness of the global transaction. Such a system does not only overcome the limitation of aborting transactions that cannot commit as HTM due to missing resources, but it also performs similar (but still slightly worse, due to the few instrumentations) than HTM in its favorable's workloads, and better that STM in scenarios where HTM behaves badly.

TM protocols are classified according to the time they decide to write into the shared memory. On the one hand, in the *lazy* approach, a transaction buffers its writes in a private buffer (redo-log) until reaching the commit time. If the commit is successful, this buffer is written back to the shared memory, otherwise, the whole buffer is discarded. On the other hand, in the *eager* approach, transaction's updates are written directly to the shared memory and old values are kept in a private undo-log. If the transaction commits successfully, the state of the shared memory is already updated, if not, the transaction is aborted and the undo-log is used to restore the old values. Given that HTM transactions do not use any redo-log and PART-HTM always executes transactions using HTM (expect for those that fall back to the GL-software path), we opt for using an eager concurrency control.

In order to cope also with transactions that do not fail for resource limitation, PART-HTM first executes incoming transactions as HTM with few instrumentations (called *first-trial* HTM transactions). In case they experience a resource failure, then our software framework "kicks in" by splitting them. Figure 4.1 shows the intuition behind PART-HTM. Let T^x be a transaction aborted for missing resources, and let T^x_1, T^x_2, ..., T^x_n be the sub-HTM transactions obtained by partitioning T^x. Let T^x_y be a generic sub-HTM transaction.

At the core of PART-HTM there is a software component that manages the execution of T^x's sub-HTM transactions. Specifically, it is in charge of: *1)* detecting accesses that are conflicting with any T^x_y already committed; *2)* preventing any other transaction T^k from reading or overwriting values created by T^x_y before T^x is committed; *3)* executing T^x in a way such that the transaction always observe a consistent state of the shared memory.

The software framework does not handle those conflicts that happen on T^x_y's accessed objects when T^x_y is still running; the HTM solves them efficiently. This represents the main benefit of our approach over a pure STM fall back implementation.

For the achievement of the above goals, the software framework needs a hint about objects accessed by sub-HTM transactions. In order to do that, we do not use the classical address/value-based read-set or write-set as commonly adopted by STM implementations [38, 33, 45], rather we rely only on cache-aligned bloom-filter-based meta-data (just bloom-filter hereafter) to keep track of read/write accesses. We recall that a bloom-filter [15] is an array of bits where the information (address in our case) is hashed to a single entry in the array (i.e., single bit). Just before to commit, a sub-HTM transaction updates a shared bloom-filter for notifying its written objects, so that no other transaction can access them. It is worth to note that HTM monitors all memory accesses, thus if two HTM transactions write different parts of the bloom-filter (thus different objects), one transaction will be

aborted anyway (*false conflict*). We reduce the probability of false conflicts by postponing the update of the global bloom-filter at commit time (i.e., lazy subscription [32, 20]), and extending the bloom-filter's size to multiple cache-lines for decreasing the conflict granularity (more details in the next sections).

Two bloom-filters per global transaction are used for recording the objects read and written by its sub-HTM transactions. In fact, these bloom-filters are passed by the framework from one sub-HTM transaction to another. Therefore, they are not globally visible outside the transaction. The purpose of these bloom-filters is to let read and written objects to survive after the commit of a sub-HTM transaction, allowing the framework to check the validity of the global transaction, anytime.

In addition, a value-based undo-log is kept for handling the abort of a transaction having sub-HTM transactions already committed. Unfortunately this meta-data cannot be optimized as the others because it needs to store the previous value of written and committed objects. We consider the undo-log as the biggest source of our overhead while executing HTM transactions. However, even though first-trial HTM transactions need to take into account all previous bloom-filters, they can omit the undo-log because they are still not part of a global transaction thus when they abort, there is no other committed sub-HTM transaction to undo. Sparing first-trial HTM transactions from this cost enables comparable performance between PART-HTM and pure HTM execution, in scenarios where most HTM transactions successfully commit without being split.

The design of PART-HTM solves also the problem of having heavy non transactional computation included in HTM transactions. In fact, such a transaction can be split in a way the non transactional computation is executed as part of the software framework, whereas only the transactional part executes as sub-HTM transaction, thus inheriting the gain of HTM.

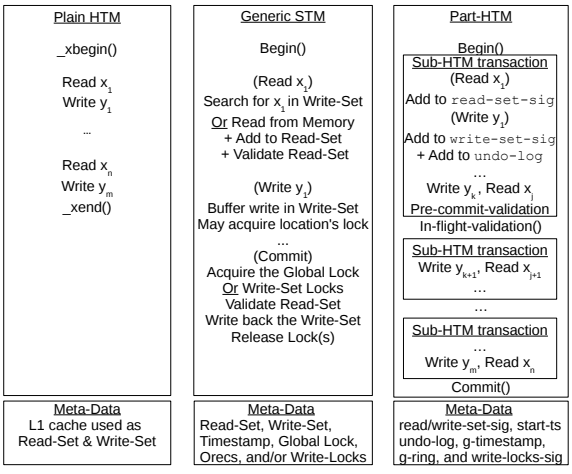


Figure 4.2: Comparison between HTM, STM, and PART-HTM.

Before we proceed with PART-HTM's algorithmic details in Section 4.3, a comparison be-

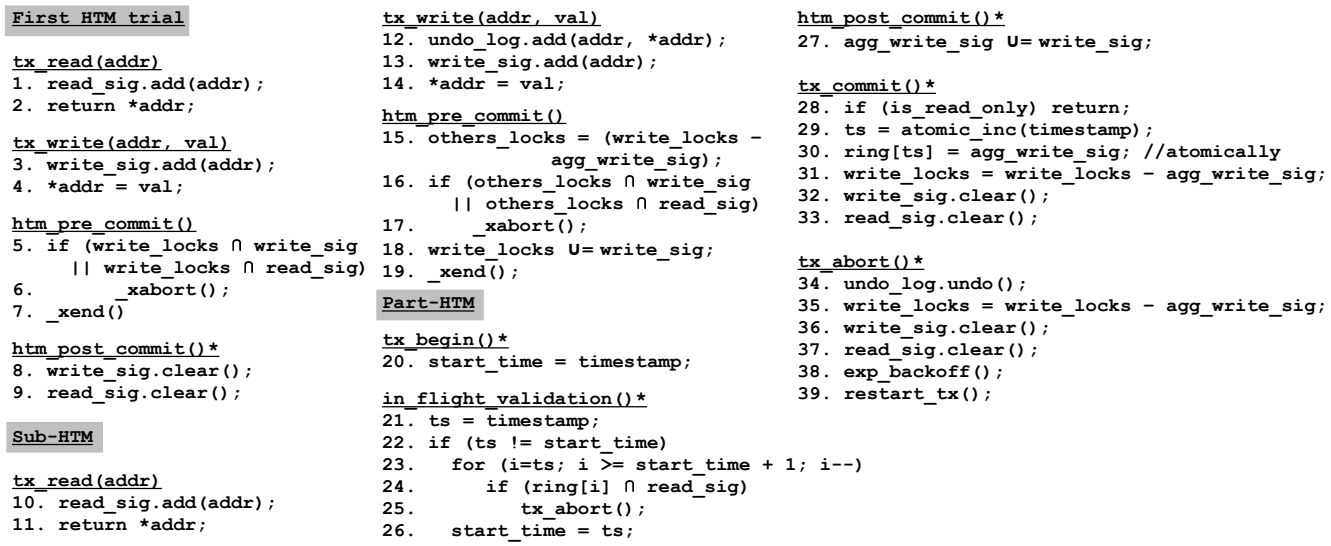


Figure 4.3: PART-HTM's pseudo-code.

tween the executions of a pure HTM, a lazy STM (e.g., [33, 101, 45, 38]), and PART-HTM is reported in Figure 4.2. In an HTM transaction, a group of reads and writes are wrapped in between a transaction begin and end. HTM internally manages the transaction atomicity, consistency and isolation by using the cache as a transactional buffer. STM needs to instrument each transactional read and write. In this example writes are buffered in the write-set and other meta-data, such as locks, are handled internally by the STM to maintain the transactions' correctness. In our system, all transactional operations are executed through slightly instrumented sub-HTM transactions. A software "wrapper" is just used for enforcing isolation and correctness (see next section), it never reads or writes objects into the shared memory.

4.3 Algorithm Details

PART-HTM splits transactions into multiple sub-HTM transactions. We refer the objects committed by some sub-HTM transaction whose global transaction is still executing as *non-visible* objects. Lastly, when we say HTM transactions, we imply both sub-HTM and first-trial HTM transactions.

Figure 4.3 shows the pseudo-code of PART-HTM's core operations. The pseudo-code, together with the scheme in Figure 4.2 are meant for increasing the clarity of the protocol. In the following subsections we refer to specific pseudo-code line using the notation [Line X]. Procedures marked by * are those that are executed in software.

4.3.1 Protocol Meta-data

PART-HTM uses meta-data for handling its operations. Some of them are local, thus visible by only one transaction, others are global and shared by all transactions. In order to reduce their size, most of them are bloom-filters [15] (i.e., a compact representation of the entire information). We refer to those meta-data as *signature*. Conflict detection using bloom-filters can cause *false conflicts* because the hash function could map more than one address into the same entry. To reduce the impact of false conflict, we set the size of our bloom-filters as a multiple of the memory cache-line size, and we reduce the conflict granularity by extending its size till 4 cache-lines.

Local Meta-data. Each transaction has its own:

- `read-set-signature`, where the bit at position i is equal to 1 if the transaction read an object at an address whose hash value is i; 0 otherwise.
- `write-set-signature`, where the bit at position i is equal to 1 if the transaction wrote an object at an address whose hash value is i; 0 otherwise.
- `undo-log`, it contains the old values of the written objects, so that they can be restored upon the transaction aborts.
- `starting-timestamp`, which is the logical timestamp (see the `global-timestamp` later) of the system at the time the transaction begins.

Global Meta-data. PART-HTM relies on three shared structures to manage contention:

- `write-locks-signature`, a bloom-filter that represents the write-locks array, where each bit is a single lock. If the bit in position i is equal to 1, it means that some sub-HTM transaction committed a new object stored at the address whose hash is i. The write-locks-signature has the same size and hash function as other bloom-filters.
- `global-timestamp`, which is a shared counter incremented whenever a write transaction commits.
- `global-ring`, which is a circular buffer that stores committed transactions' `write-set-signatures`, ordered by their commit timestamp. The `global-ring` has a fixed size and is used to support the validation of executing transactions, in a similar way as proposed in RingSTM [101].

The last two global meta-data are only used by the software framework outside the execution of sub-HTM transactions.

4.3.2 Begin Operation and Transaction's Partitioning

PART-HTM processes transactions at the beginning as HTM transactions (i.e., first-trial). These first-trial HTM are not pure HTM transactions, because they are slightly instrumented according to the rules illustrated later in this section.

When the first-trial HTM transaction fails for resources limitation, then the software framework splits the transaction in multiple sub-HTM transactions. This process does not constitute the main contribution of PART-HTM because there are several efficient policies that can be applied. Examples include those using compiler supports, such as [4, 3], or techniques that estimate the expected usage of cache-lines so that they can propose an initial partitioning. In our implementation we adopt the following strategy: the compiler marks basic source code blocks, that are used by the runtime of PART-HTM to decide where a transaction can be split.

When a transaction starts, it reads the `global-timestamp` and stores it as the transaction's `starting-timestamp` [Line 20]. All local meta-data, except the `starting-timestamp`, are passed from the software framework to the first sub-HTM transaction, which updates them according to the outcome of its performed operations. When a sub-HTM transaction commits, the software framework forwards the updated local meta-data to the next sub-HTM transaction and so on, until reaching the global commit phase.

4.3.3 Transactional Read Operation

Read operations are always performed by HTM transactions in PART-HTM, thus they can happen either during the execution of first-trial HTM transactions or sub-HTM transactions. In both cases the behavior is identical and straightforward because, essentially, every read operation always accesses the shared memory [Line 2 or 11]. In case a previous sub-HTM transaction, belonging to the same global transaction, committed a new value of that object, this new value is already stored into the shared memory since HTM uses the write in-place technique. If the read object has been already written during the current HTM transaction, then the HTM implementation guarantees the access to the latest written value.

In addition, in order to detect if the accessed object is a version non-visible yet, the read memory location is recorded into the `read-set-signature` [Line 1 or 10]. This information is fundamental for guaranteeing the isolation from other transactions having sub-HTM transactions already committed.

4.3.4 Transactional Write Operation

Similar to read operations, writes also always execute within the context of an HTM transaction, thus objects are written directly into the shared memory [Line 4 or 14]. However, a write operation cannot just overwrite any value in the shared memory because there could be non-visible objects, which cannot be accessed. In other words, and as for the case of reads, a write operation should not break the isolation of other transactions having sub-HTM transactions already committed, by means of overwriting objects that are non-visible. To prevent this scenario, write operations add the locations of the written objects to the

transaction's `write-set-signature` [Line 3 or 13]. This information will be used by the HTM transaction before proceeding with the commit phase.

In addition to the above steps, if the write operation is executed by a sub-HTM transaction, two other important actions must be taken into account. These actions do not apply to first-trial HTM transactions. First, the global transaction could abort in the future, even after committing the current sub-HTM transaction. If this happens, the previous values of written objects should be replaced into the shared memory. For this reason, before to finalize the write operation, the old value of the object is logged into the transaction's `undo-log` [Line 12]. Second, the new value of the object should be protected against accesses from other transactions, and this is done by updating the global `write-locks-signature`. However, since the write operation is executed inside an HTM transaction, it is speculative and it will not take place until the HTM transaction commits. For this reason, we delay the update of the `write-locks-signature` until the end of the sub-HTM transaction [Line 18]. It is worth to note that, every update to a shared meta-data, such as the `write-locks-signature`, causes the abort of all HTM transactions that read the specific cache-line where the meta-data is located, even if they updated or tested different bits (false conflict). For this reason, delaying the update of the `write-locks-signature` at the end of the sub-HTM transaction is effective because it minimizes also false conflicts.

In practical, the task of notifying that a new object has been just committed, but is non-visible, is very efficient and uses the technique showed in Figure 4.4(a): the `write-locks-signature` is updated to the result of the bitwise `OR` between transaction's `write-set-signature` and the `write-locks-signature` itself.

Semantically, the `write-locks-signature` contains the information regarding locked objects already stored into the shared memory. Beside the terminology, PART-HTM does not use any explicit lock for protecting memory locations. As an example, no *Compare-And-Swap* operation is required for acquiring the locks on written objects. Updating the `write-locks-signature` (i.e., the lock acquisition) is delegated to the sub-HTM transaction itself. If no HTM transaction is accessing to the `write-locks-signature`, then it is updated, otherwise the sub-HTM transaction is aborted.

4.3.5 Validation

PART-HTM requires two types of validation. One executed by HTM transactions before commit (called *HTM-pre-commit-validation*), and one executed by the software framework after the commit of a sub-HTM transaction (called *in-flight-validation*).

HTM-pre-commit-validation. This validation is performed at the end of each HTM transaction (both sub-HTM and first-trial HTM transactions) and it has a twofold purpose.

First, HTM transactions should not overwrite any non-visible memory location (i.e., locked), because, in this case, the sub-HTM transaction that committed that object has its global

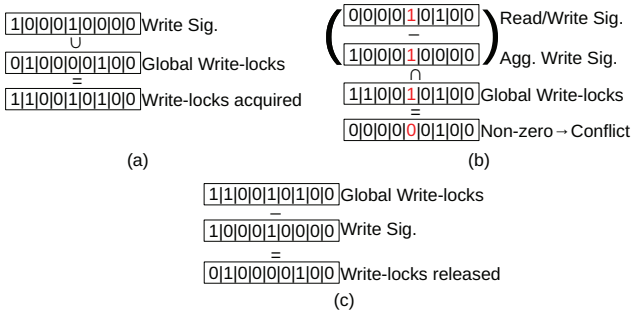


Figure 4.4: Acquiring write-locks (a). Detecting intermediate reads or potential overwrites (b). Releasing write-locks (c).

transaction not yet committed. Overwriting that object means breaking the isolation of the global transaction. To prevent this, any HTM transaction compares its `write-set-signature` with the global `write-locks-signature` through a bitwise `AND` (i.e., the intersection between the two bloom-filters [Line 5 or 16]) as shown in Figure 4.4(b). If the result is a non-zero bloom-filter, it means that the HTM transaction wrote some object that was locked, thus it should abort [Line 6 or 17].

Due to the nature of the bloom-filters, a lock is just a bit and has no ownership information. Thus, a transaction is not able to distinguish between its own locks, which are acquired by previous sub-HTM transactions, and others' locks. We solve this issue by separating the current sub-HTM transaction's `write-set-signature` from the aggregated `write-set-signature` of the global transaction. This way, each sub-HTM transaction knows whether the locked location is owned by its global transaction or not [Line 15]. The aggregated `write-set-signature` is updated in software after each sub-HTM transaction [Line 27].

Second, HTM transactions should not read the value of locked (i.e., non-visible) objects, in order to prevent the exposition of uncommitted (partial) state of an executing transaction. To enforce this rule, during the HTM-pre-commit-validation the transaction's `read-set-signature` is intersected with the `write-locks-signature` [Line 5 or 16] (as in Figure 4.4(b)). A resulting non-zero bloom-filter suggests to abort the current HTM transaction for avoiding the corruption of the isolation of other executions.

The HTM-pre-commit-validation is mandatory for the correctness of PART-HTM, thus it cannot be skipped. However, there are rare cases that could allow an HTM transaction to commit without performing the HTM-pre-commit-validation. These cases are related to possible invalid objects read inside the HTM by doomed transactions, which could generate unexpected behaviors. To address this problem the Haswell's RTM provides a *sandboxing* mechanism so that a transaction is eventually aborted if its execution hangs or loops indefinitely. However, the sandboxing has some limitation, specifically when a corrupted value is used as a destination address of an *indirect jump* instruction. If, by chance, the target

address of this incorrect jump is the `_xend` instruction (i.e., the instruction used for demarcating the bound of an HTM transaction) [19], then the commit is called without executing the HTM-pre-commit-validation. We solve this issue by invoking the HTM-pre-commit-validation before any indirect jump.

In-flight-validation. This validation is performed by the software framework after the commit of every sub-HTM transaction whereas first-trial HTM transactions do not need to call it. The in-flight-validation is needed for ensuring that the memory snapshot observed by the global transaction is still consistent after the commit of a sub-HTM transaction and before invoking the next one.

Assuming the scenario with two global transactions T^x and T^y, both having two sub-HTM transactions each. Let us assume that T_1^x reads the value of object o and commits. Let us also assume that o is not locked at this time. After that, T_2^y is scheduled. It overwrites and locks o, invalidating T^x. T^x is able to detect this conflict through the HTM-pre-commit-validation invoked before the T_2^x's commit, but let us assume that the commit of T^y is scheduled before T_2^x's commit (in fact, T_2^y is the last sub-HTM transaction of T^y). As we will show later in the commit procedure, all transaction's locks are cleared from the `write-locks-signature` when the global transaction commits. This means that, the intersection between T_2^x's `read-set-signature` and the `write-locks-signature` does not report any conflict on o, therefore T_2^x can successfully commit even if T^x's execution is not consistent anymore.

The in-flight-validation solves this problem by comparing the transaction's `read-set-signature` against the `write-set-signature` of all concurrent and committed transactions [Line 21-26]. Retrieving committed transactions, as we will show later, is easy because they have an entry in the `global-ring`, associated with their commit timestamp. The selection of those that are concurrent is straightforward because they have a commit timestamp that in higher than the `starting-timestamp` of the transaction that is running the in-flight-validation [Line 23].

After a successful in-flight-validation, the transaction's `starting-timestamp` is advanced to the current `global-timestamp` [Line 26]. This way, subsequent in-flight-validations do not pay again the cost of validating the global transaction against the same, already committed, transactions.

It is worth to notice that the in-flight-validation is done after each sub-HTM transaction mainly for performance reason. In fact, in order to ensure serializable executions, the in-flight-validation could be done just one time after the commit of the last sub-HTM transaction and before commit. We decided to perform it after each sub-HTM transaction for two reasons: *i)* this way the software framework always observes the consistent value of read and written objects; *ii)* detecting invalidated objects early in the execution avoids unnecessary computation and saves precious HTM resources.

4.3.6 Commit Operations

The commit of a transaction in PART-HTM is straightforward. First-trial HTM transactions are committed directly in HTM, no additional operation is required [Line 7].

If the transaction is read-only (i.e., no writes occurred during the execution), it has been already validated before entering the commit phase, thus it can just commit [Line 28].

Even the case where the transaction performed at least a write operation is simple because it has been already validated by both the HTM-pre-commit-validation, invoked before committing the last sub-HTM transaction, and the in-flight-validation, called after the last sub-HTM transaction. In addition, its written objects are already applied to the shared memory and protected by locks. The only remaining tasks are related to the update of the global meta-data. The transaction adds its `write-set-signature` to the `global-ring` [Line 30] and increments the `global-timestamp` [Line 29], atomically. Finally, all transaction's write locks should be released [Line 31]. This operation is done by executing a bitwise `XOR` between the transaction's `write-set-signature` and the global `write-locks-signature`, as shown in Figure 4.4(c).

4.3.7 Aborting a Transaction

The abort of first-trial HTM transactions is handled by the HTM implementation itself. Sub-HTM transactions that fail the HTM-pre-commit-validation are explicitly aborted and retried for a limited number of times (5 in our implementation) before being handled by the software framework.

The abort of a global transaction requires to restore the old memory values of objects written by its committed sub-HTM transactions. This operation is done traversing the transaction's `undo-log` [Line 34]. After that, the transaction's owned write-locks are released from the global `write-locks-signature` [Line 35] and a retry is invoked after an exponential back-off time [Line 38-39]. Before to proceed with the next rerun, if the transaction has been aborted for a resource failure, it will be split again. After 5 aborts within the software framework, the transaction finally falls back to the GL-software path.

4.4 Compatibility with other HTM processors

The IBM Power8 HTM processor supports execution of non-transactional code inside an HTM transaction. Two new instructions `tsuspend` and `tresume` are provided such that a transaction can be suspended and resumed, respectively. Our algorithm can take advantage of this "non-transactional window" for executing the HTM-pre-commit-validation and for acquiring the `write-locks-signature`. This way, aborts due to false conflict on those

meta-data are avoided.

We designed PART-HTM to be *hardware friendly*, namely all the meta-data and procedures used can be implemented directly in hardware because they are limited in space and their size is small. Also, bloom-filter read/write signatures can be generated via hardware by [24]. As a result, these characteristics make PART-HTM a potential candidate for being implemented directly as hardware protocol, thus significantly improving its performance.

4.5 Correctness

PART-HTM ensures serializability [1] as correctness criterion. In this section, we show how PART-HTM is able to detect that a transaction conflicts with another concurrent transaction.

Three types of conflicts can invalidate the execution of a transaction: *write-after-read*, *read-after-write*, and *write-after-write*. Let T_r^x and T_w^y be the sub-HTM transactions reading and writing, respectively. T_r^x belongs to the global transaction T^x whereas T_w^y to T^y.

The *write-after-read* conflicts happen when T_r^x reads an object o that T_w^y will write. If the conflicting operations of T_r^x and T_w^y happen while both the transactions are running, the HTM conflict detection will abort one of them. Otherwise, it means that the write operation of T_w^y on o is executed after the commit of T_r^x. If so, T_r^x will detect this invalidation through the HTM-pre-commit-validation performed at the end of the sub-HTM transaction that follow T_r^x. If there is no sub-HTM transaction after T_r^x, it means that T^x commits before T^y, thus the conflict was not an actual conflict because T^y will be serialized after T^x. On the other hand, if T_w^y is the last sub-HTM transaction of T^y, T^y will be committed and its `write-set-signature` attached to the `global-ring`. In this case, the in-fligh-validation performed by T^x after the commit of T_r^x will detect the conflict and abort T^x.

The *read-after-write* conflicts happen when T_r^x reads an object o that T_w^y already wrote. As before, if the conflict is materialized while both are running, the HTM handles it. If T_r^x reads after the commit of T_w^y, but T^y is still executing, then T_r^x will be aborted before it could commit thanks to the HTM-pre-commit-validation, which detects a lock taken on o by T^y. If T^y commits just after T_w^y, this is not a problem because it means that T_r^x accessed to the last committed version of o.

The *write-after-write* conflicts are detected because, otherwise, a read operation on an object already written inside the same transaction could return a different value. Besides the trivial case where both the writes happen during the HTM execution, before committing, all HTM transactions perform the HTM-pre-commit-validation, which intersects the transaction's `write-set-signature` with the global `write-locks-signature`, thus detecting a taken lock.

Following the above rules, a transaction arrives at commit time having observed a state that is still valid, thus it can commit. Any possible invalidation that happens after the last in-

flight-validation is ignored because the transaction is intrinsically committing by serializing itself before the transaction that is invalidating (we recall that all objects are already into the shared memory and protected by locks).

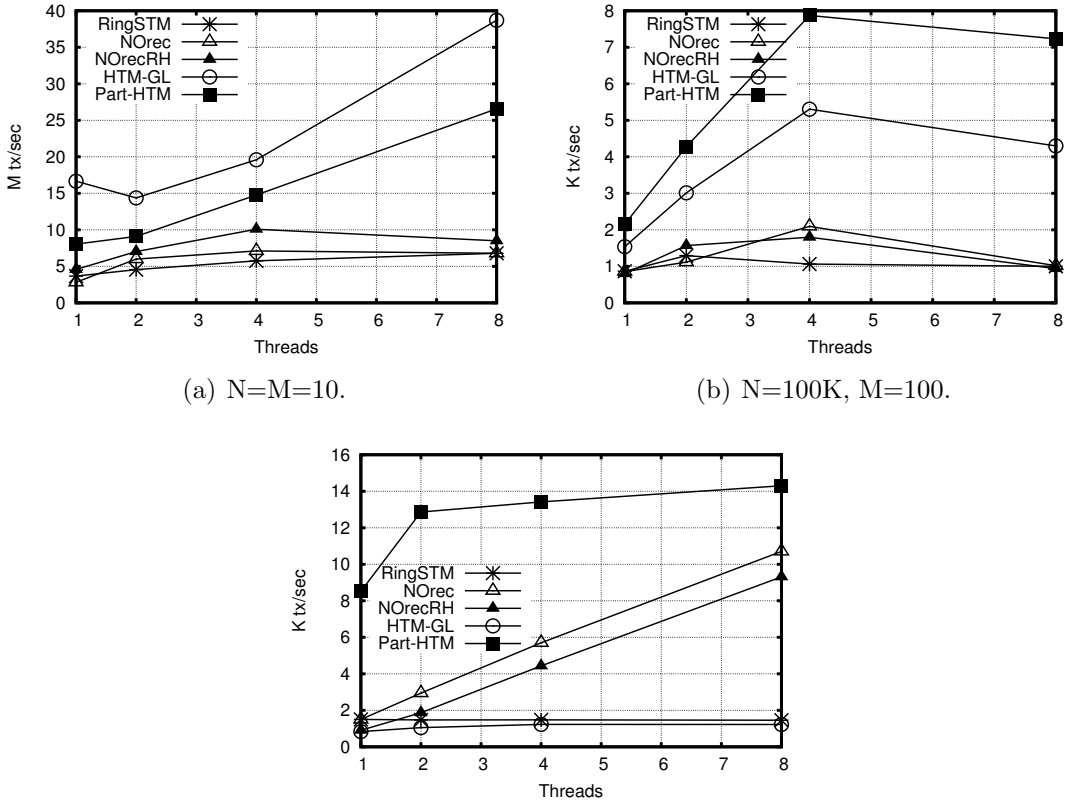
Considering that PART-HTM reads and writes only using HTM transactions, there is the possibility that doomed transactions (those that will be aborted eventually) could observe inconsistent states while they are running as HTM transactions. In fact, locks are checked only before committing the HTM transaction, thus a hardware read operation always returns the value written in the shared memory, even if locked. The return value of those inconsistent reads could be used by the next operations of the transaction, generating not predictable execution (e.g., infinite loops or memory exception). This behavior does not break serializability because aborted transactions are not taken into account by the correctness criterion. However, for in-memory processing, like TM, avoiding such scenarios is desirable [48]. A trivial solution for this problem consists of instrumenting every hardware read operation, but it would mean loosing the advantage of executing as HTM. Fortunately, the HTM provides a sandboxing feature, which eventually aborts those misbehaving HTM transactions, preventing the propagation of the error to the application. If no error happens, the doomed transaction is aborted using the HTM-pre-commit-validation.

4.6 Evaluation

PART-HTM has been in implemented in C++. To conduct an exhaustive evaluation, we used three benchmarks: *N-reads M-write*, a known and configurable application provided by RSTM [74]; linked-list, a data structure; STAMP [77], the popular suite of applications designed for TM-based systems and widely used for evaluating STM and HTM-related concurrency controls (e.g., [19, 41]).

As competitors, we included two state-of-the-art STM protocols, RingSTM [101] and NOrec [33]; one hybridTM, Reduced Hardware NOrec (NOrecRH) [75]; and one HTM with the GL-software path as fall back (HTM-GL). For both the STM algorithms we used the version integrated in RSTM. NOrecRH and HTM-GL retry a transaction 5 times as HTM before falling back to the software path. In this evaluation study we used Intel Haswell Core i7-4770 processor and we disabled the hyper-threading support to avoid sharing of HTM resources between threads running on the same core. All the data points reported are the average of 5 repeated execution.

N-Reads M-Writes. This benchmark is designed for allowing programmer to vary the transaction size by changing the N and M parameters. Each transaction reads N elements from one array and writes M to another. The contention level can be tuned by changing the array size but transactions can be also configured to access disjoint elements (i.e., no contention). We take advantage of this latter feature so that we can evaluate PART-HTM in scenarios where the conflict failures of HTM transactions are minimized. All other bench-



(a) N=M=10.

(b) N=100K, M=100.

(c) N=M=100.

Figure 4.5: Throughput using N-Reads M-Writes benchmark and disjoint accesses.

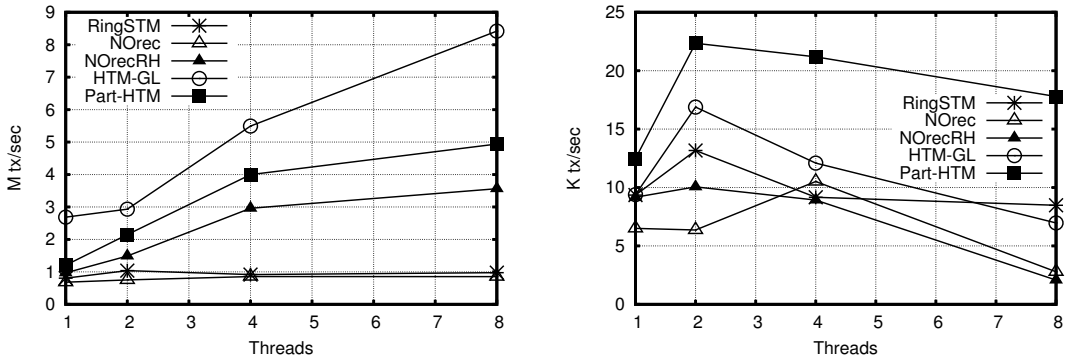
marks do not offer any disjoint access pattern thus, with them we will cover the conflicting scenarios. We report three workloads where we change the transactions' size and execution time.

Figure 4.5(a) shows the results of reading and writing only 10 disjoint elements. In this experiment, no transaction is aborted for resource failure, thus all commit as HTM. Thus, HTM-GL has the best throughput, followed by PART-HTM which is the closest to HTM-GL. This scenario does not represent the best case for PART-HTM but still, thanks to the lightweight instrumentation of first-trial HTM transactions, PART-HTM shows a slow-down limited to 45% over HTM-GL, whereas the best competitor (NOrecRH) is 91% slower than PART-HTM.

Figure 4.5(b) shows an experiment where 100,000 elements are read and 100 elements are written to a large array. This scenario reproduces large transactions in a read-dominated workload. Here, HTM-GL still performs good because the Haswell HTM implementation can go beyond the L1 cache capacity just for read operations [29], however most of HTM transactions fall back to the GL-software path. For this reason, the benefit of partitioning

is evident and transactions can be committed using sub-HTM transactions, which are much faster than falling back to the GL-software path. PART-HTM gains up to 50% over HTM-GL. STM protocols and NOrecRH suffer from excessive instrumentation cost due to the several operations per transaction.

In Figure 4.5(c), we configured the benchmark such that each transaction performs one read on an object o, then it does some floating point instructions on o before to write its new value back to the destination array. This sequence is repeated 100 times. This way we emulate transactions that could be committed as HTM in terms of size but, for time limitation, are likely aborted (e.g., by a timer interrupt). In this scenario, PART-HTM shows a significant speed-up compared to other competitors. HTM-GL executes all transactions using global locking. NOrecRH and NOrec perform similar but NOrecRH is slightly worst as it executes the transaction in hardware first.



(a) Linked-list with 1K elements and 50% writes.

(b) Linked-list 10K elements and 50% writes.

Figure 4.6: Throughput using Linked-List.

Linked-List. In this benchmark, we do operations on a linked list data structure. We change the size of the linked list and the percentage of write operations (insert and remove) against read operations (contains). Linked list transactions traverse the list from the beginning until the requested element. This increases the contention between transactions. Write operations are balanced such that the initial size of the list remains stable.

Figure 4.6(a) shows the results of a 1000 elements linked list using 50% of write operations. Linked list operations do a large amount of memory reads to traverse the data structure, and a few writes to insert or remove an element. Thus, almost all transactions commit in hardware and HTM-GL has the best throughput. However, following the same trend as Figure 4.5(a), PART-HTM places its performance closer to HTM-GL.

Figure 4.6(b) shows a larger linked list with 10,000 elements. Here, most of the transactions fail in hardware for resource failures. As for the case in Figure 4.5(c), PART-HTM's throughput is the best as sub-HTM transactions pay a limited instrumentation cost and fast execution in hardware. Here PART-HTM gains up to 74% over HTM-GL. At 8 threads, all

competitors degrade their performance due to the high number of conflicting traversing the list.

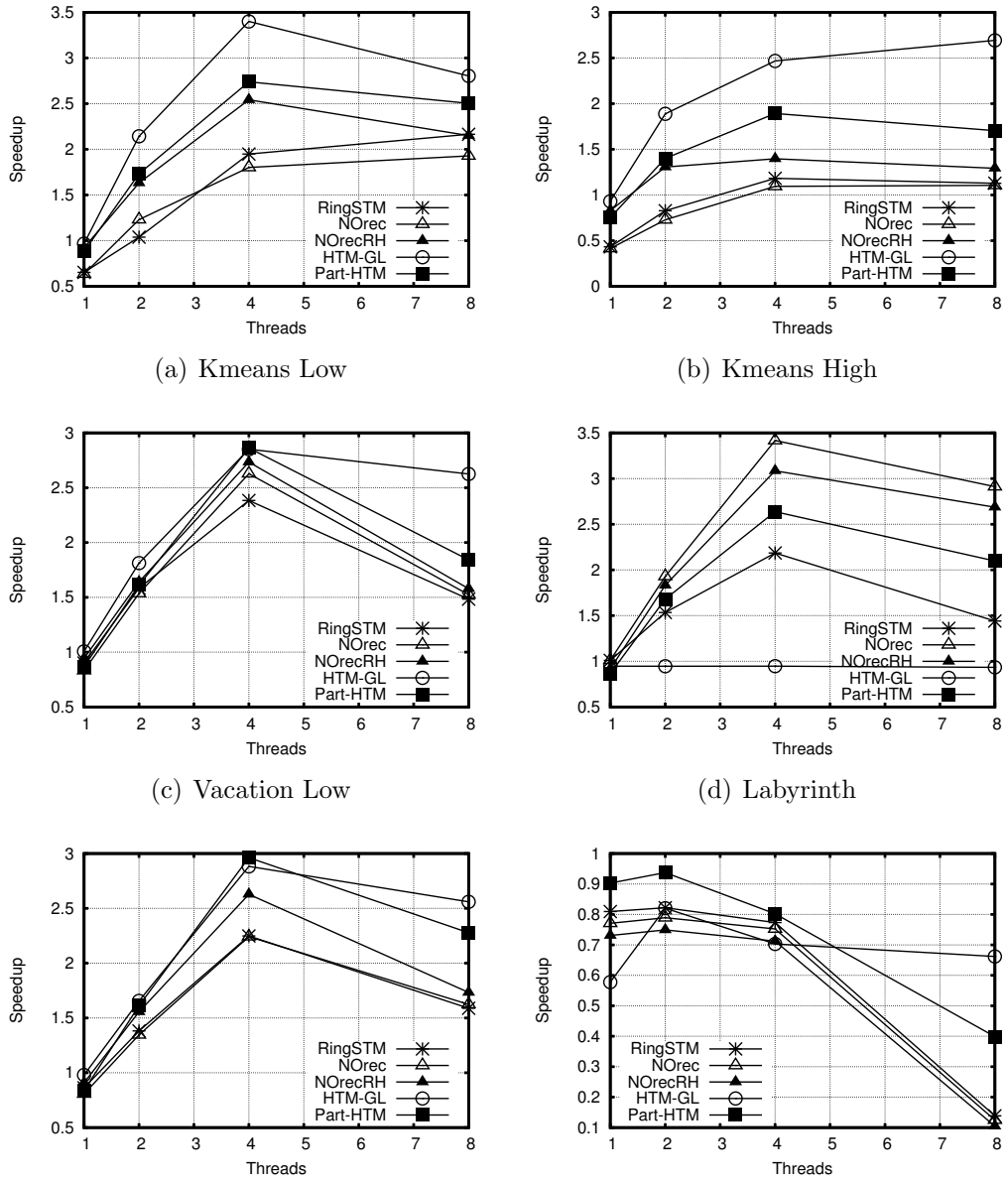
STAMP. Figure 4.7 shows the results of different STAMP applications. STAMP applications' transactions likely do not fail in HTM except for Labyrinth and Yada. However, most of the effort in the design of PART-HTM is focused on reducing overheads. In fact, STAMP applications's performance confirms the goodness of the design because PART-HTM is the closest to the best competitors in most of cases. All data points report the achieved speed-up with respect to the sequencial execution of the application.

Kmeans (Figure 4.7(a) and 4.7(b)), Vacation (Figure 4.7(c)) and Intruder (Figure 4.7(e)) are application where HTM transactions do not fail for resource limitations, but they are mostly short and conflict due to real conflicts. In all those application, HTM-GL is the best but PART-HTM is always the closest competitor. On the other hand, application like Labyrinth (Figure 4.7(d) and Table 4.1) are suited more for STM protocols than HTM. This is because, in Labyrinth more than half of the generated transactions are large and long (thus HTM cannot be efficiently exploited). In addition, they also rarely conflict each other, thus false conflicts are the main reason of performance degradation. As a result, NOrec and NOrecRH have the best speed-up over serial execution but PART-HTM is placed just after them. Interestingly, PART-HTM behaves better than RingSTM because, even though both suffer from false conflicts, the latter exploits faster sub-HTM transactions to commit and optimized signature-based meta-data.

	% of Aborts				% of committed tx per type		
	Conflict	Capacity	Explicit	Other	GL	HTM	SW
(A)	10.11%	70.76%	0.04%	19.09%	49.6%	50.4%	N/A
(B)	93.95%	1.09%	1.14%	3.82%	0.1%	50.3%	49.6%

Table 4.1: Statistics' comparison between HTM-GL (A) and PART-HTM (B) using Labyrinth application and 4 threads.

Finally, Figure 4.7(f) shows the results of Yada. This application has transactions that are long and large, generating a reasonable contention level. Thus it represents a favorable workload for PART-HTM. In fact, it is the best until 4 threads. PART-HTM becomes worse than HTM-GL using 8 threads because, in this case, the contention is high and executing transactions protected by a single global lock performs better than executing them in parallel.



(a) Kmeans Low

(b) Kmeans High

(c) Vacation Low

(d) Labyrinth

(e) Intruder

(f) Yada

Figure 4.7: Speed-up over serial (single-thread) execution using applications of the STAMP Benchmark.

Chapter 5

Octonauts

5.1 Problem Statement

Transactional Memory (TM) achieves high concurrency when the contention level is low (i.e., few conflicting transactions are concurrently activated). At medium and high contention level, transactions aborts each other more frequently and a contention manager or a scheduler is required. A contention manager (CM) is an encounter time technique: when a transaction is conflicting with another one, the module implementing the CM is consulted, which decides which transaction of them can proceed. As a consequence, the other transaction is aborted or delayed. A CM collects information about each transaction (e.g., start time, number of reads/writes, number of retries). Based on the collected information and the CM policy, CM decides priorities among conflicting transactions. This guarantees more fairness and progress and could prevent some potential live-lock situation. A CM can work either during the transaction's execution by using live (on the fly) information, or work prior the transaction's execution. Schedulers in the latter category use information about transaction's potential working-set (reads and writes) defined a priori in order to avoid the need of solving conflicts while transactions are executing.

In this chapter, we address the problem of CM in Hardware Transactional Memory (HTM). Current Intel's HTM implementation is a black-box because it is entirely embedded into the cache coherence protocol. The L1 cache of each core is used as a buffer for the transactional write and read operations. The granularity used for tracking accesses is the cache line. The eviction and invalidation of cache lines defines when a transaction is aborted (it reproduces the idea of read-set and write-set invalidation of STM). Since there is no way to change Intel's HTM conflict resolution policy (because it is embedded into the hardware cache coherence protocol), also the implementation of a classical CM cannot be trivially provided. Regarding this topic, the Intel's documentation says "*Data conflicts are detected through the cache coherence protocol. Data conflicts cause transactional aborts. In the initial implementation,*

the thread that detects the data conflict will transactionally abort.". As a result, we cannot know which thread will detect the conflict as the details of Intel's cache coherence protocol are not publicly available.

From an external standpoint, when a conflict is detected between threads accessing the same cache line, one of the transaction running on them is immediately aborted without giving the programmer a chance to resolve the conflict in a different manner. For example, when two concurrent transactions access the same cache line, and one access is a write, one HTM transaction will detect the conflict when it receives the cache coherence invalidation message. That transaction will immediately abort. The program will jump to the abort handler where it should handle the abort. Thus, when the program is notified with the abort, it is already too late to avoid it or decide which transaction was supposed to abort. In addition, Intel's HTM treat all reads/writes in a transaction as a transactional reads/writes, even if the accessed object is not shared (i.e., a local variable). Non-transactional accesses inside a transaction cannot be performed in the current HTM implementations.

Customizing the conflict resolution policy and control which transaction aborts means necessarily detecting the conflict before it happens. However, this also means repeating what HTM already does (i.e., conflict detection) with a minimal overhead because it is provided at the hardware level. In addition, every access to shared data (read/write) should be monitored for a potential conflict. In other words, every access to a shared object should be instrumented such that we know if other transactions are accessing that object concurrently. We also need to keep information about each object (i.e., meta data). That leads to another problem, having a shared meta data for each object and reading/updating such meta data will introduce more conflicts (we recall that HTM triggers an abort if any cache line is invalidated, even if in that cache line there is stored a non shared object). For example, if we will add a read/write lock for each object which indicates which transaction is reading/writing the object, then each transaction should read the lock status before reading/writing the object. From the semantics standpoint, if the lock is acquired by one transaction for read and another transaction reads the object, then it can proceed and acquire the lock for read too. However, at the memory level, the acquisition of the lock means writing to the lock variable. From the HTM standpoint, the lock is just an object enclosed in a cache line. Reading the lock status will add it to the transaction read-set, and acquiring (updating) the lock will add it to the write-set. Since all memory accesses in an HTM transaction are considered as transactional, once a transaction acquires the lock, it will conflict with all other transactions that read/wrote to the same lock. In order to solve this problem, we need a technique to collect information about each object without introducing more conflict.

On the other hand, adding a scheduler in front of HTM transactions is more appealing because it does not necessarily require live information to operate. Such a scheduler uses static information about incoming transactions, and based on these information, only those transactions that are non conflicting can be concurrently scheduled (thus no conflicting transactions can simultaneously execute in HTM). Thus, a scheduler can orchestrate transactions without introducing more conflict. In addition, considering the best-effort nature of HTM transac-

tions, a scheduler should handle also the HTM fallback path efficiently (i.e., HTM-aware scheduler). As an example, falling back to STM rather than global lock usually guarantees better performance. Thus, the scheduler should allow HTM and STM transactions to run concurrently without introducing more conflict due to HTM-STM synchronization. Finally, the scheduler should also be adaptive, namely if a transaction cannot fit in HTM or is irrevocable (thus cannot be aborted), then the scheduler should start it directly as an STM transaction or alone exploiting the single global lock.

5.2 Algorithm Design

We propose OCTONAUTS, an HTM-aware Scheduler. OCTONAUTS's basic idea is to use queues that guard shared objects. A transaction first declares its potential objects that will be accessed during the transaction (called *working-set*). This information is provided by the programmer or by a static analysis of the program. Before starting a transaction, the thread subscribes to each object queue atomically. Then, when it reaches the top of all subscribed queues (i.e., it is the top-standing), it can start the execution of the transaction. Finally, it is dequeued from the subscribed queues thus allowing the following threads to proceed with their transactions. Large transactions, which cannot fit as HTM, are started directly in STM with their commit phase executed as a reduced hardware transaction (RHT) [76]. To allow HTM and STM to run concurrently, HTM transactions runs in two modes. First mode is entirely HTM, this means that no operations are instrumented. The second mode is initiated when an STM transaction is executing. Here, a lightweight instrumentation is used to let the concurrent STMs know about executing HTM transactions. In our scheduler we managed to make this instrumentation transparent to HTM transactions, this way HTM transactions are not aware of concurrent STM transactions. In fact, in our proposal HTM transactions notify STM with their written objects signature. STM transactions uses the concurrent HTM's write signatures to determine whether its read-set is still consistent or an abort should be invoked. This technique does not introduce any false conflicts in HTM transactions, compared to other techniques such as subscribing to the global lock in the beginning of an HTM transaction or at the end of it. If a transaction is irrevocable, it is stated directly using global locking. In addition, if the selection of the adaptive technique turns out to be wrong, then an HTM transaction will fallback to STM, and an STM transaction will fallback to global locking.

Figure 5.1 shows how each thread subscribes to different queues based on their transaction working-set, wait for their time to execute, and execute transactions. In this figure, T1 and T2 are on the top of all queues required by their working-set. Thus, T1 and T2 started executing their own transactions. T4 cannot start execution since it is still waiting to be on the top of O2 queue. Once T2 finishes execution, it will be dequeued from O2 queue allowing T4 to proceed. T5 must wait for T2 and T4 to finish in order to be on top of O2, O5 and O6 queues and start execution. T3 just arrived and it is subscribing to O1, O3 and

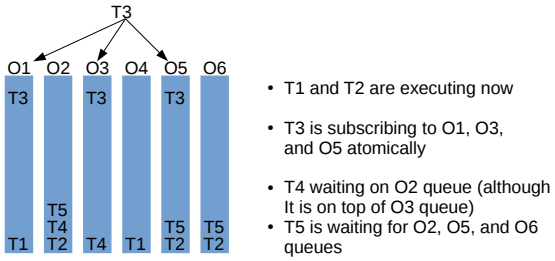


Figure 5.1: Scheduling transactions.

O5 by enqueueing itself to the corresponding queues atomically.

5.3 Algorithm Details

OCTONAUTS is an HTM-aware Scheduler. It is designed to fulfill the following tasks:

1. reduce conflicts between transactions;
2. allows HTM and STM transactions to run concurrently and efficiently;
3. analyze programs to detect potential transaction data size, duration and conflicts;
4. schedule large transaction immediately as STM transaction without an HTM trial.

5.3.1 Reducing Conflicts via Scheduling

As shown in Figure 5.1, every thread before starting a new transaction subscribes to each object's queue in its working-set. Each queue represents an object or a group of cache lines. Once subscribed, the thread waits until it is on the top of all subscribed queues. Finally, it executes the transaction and the thread is dequeue from all subscribed queues.

In order to implement this technique correctly and efficiently, we used a systems inspired by the synchronization mechanism where tickets are leveraged. We use two integers i.e., `enq_counter` and `deq_counter` and a lock to represent each queue. To subscribe to a queue, a thread atomically increments the `enq_counter` of that queue (i.e., acquire a ticket). Then, it loops on the `deq_counter` until it reaches the value of its own ticket. To prevent deadlock, a thread must subscribe to all required queues at the same time (i.e., atomically). To accomplish this task, the thread acquires all required queues' locks before incrementing `enq_counter`. When the thread finishes the execution of the transaction, it increments all subscribed queues' `deq_counter` and therefore next (conflicting) transactions are allowed to proceed.

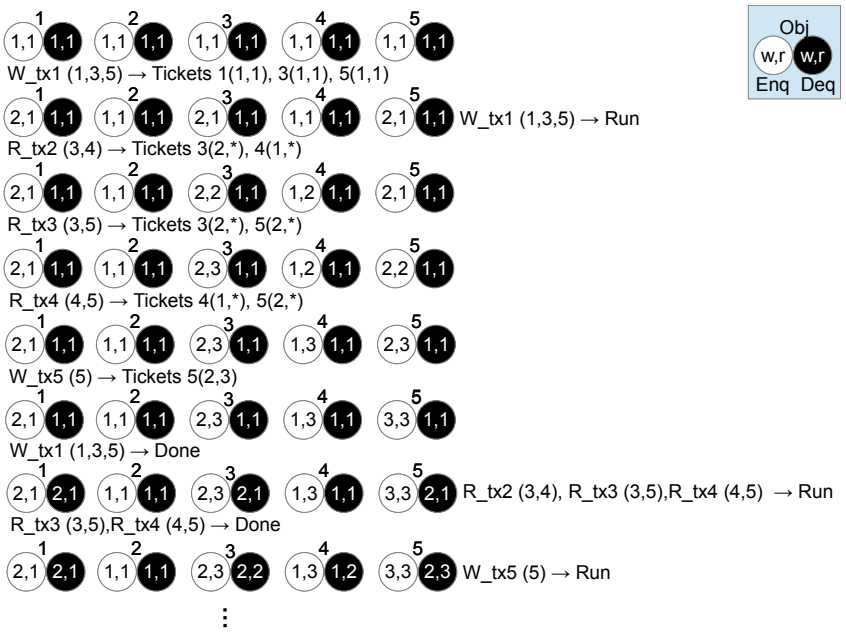


Figure 5.2: Readers-Writers ticketing technique.

Using the describe technique, two read-only transactions accessing the same object are not allowed to execute concurrently. However, such a read-only transaction cannot conflict with each other (because none of them writes) and serializing them affects the performance significantly, especially in read dominated workloads. To address this issue, we modified the aforementioned ticketing technique to accommodate reader and writer tickets. Readers ticket's owners can proceed together if there is no active writers. Rather, conflicting writers are serialized.

The readers/writers ticketing system works as follows. Instead of `enq_counter` and `deq_counter`, we have `w_enq_counter`, `w_deq_counter`, `r_enq_counter` and `r_deq_counter`. Each transaction now has two tickets. A writer transaction increments `w_enq_counter` and reads the current `r_enq_counter`, while a reader transaction increments `r_enq_counter` and reads the current `w_enq_counter`. A writer transaction waits for `w_deq_counter` and `r_deq_counter` to reach their tickets numbers. A reader transaction waits for `w_deq_counter` only. After executing the transaction, a reader transaction increments `r_deq_counter` only, while a writer transaction increments `w_deq_counter` only. Following the example in figure 5.2, we notice that one writer ticket can unlock many readers to proceed in parallel.

Figure 5.2 shows how reader and writer threads proceed using our readers-writers ticketing technique. Reader threads are only blocked by conflicting writers (their tickets include the `w_enq_counter` and any value for the `r_enq_counter` which is represented by * in the figure). Writers threads are blocked by both conflicting readers and writers. The figure also shows how multiple reader threads can proceed together.

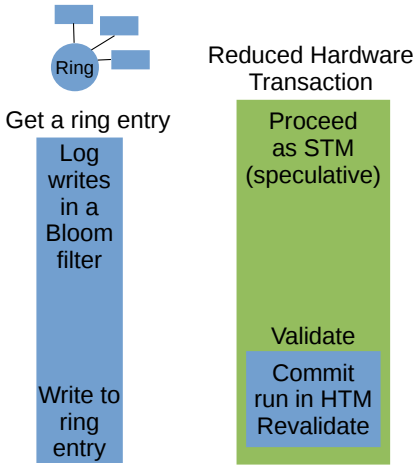


Figure 5.3: HTM-STM communication.

5.3.2 HTM-aware Scheduling

Intel's HTM is a best efforts HTM where transactions are not guaranteed to commit. A fallback path must be provided to ensure progress. In the previous sub-section, we showed how to prevent conflicting transaction from running concurrently, which solves the problem of aborts due to conflicts. However, HTM transactions can be also aborted for resource limitations reasons (i.e., space or time) if the transaction cannot fit into the HW transactional buffer or requires time larger than the OS scheduler time slice. This type of transactions cannot successfully complete in HTM, and the only way to commit them is to let them run alone using a global lock or run them as STM transaction. Acquiring a global lock reduces the concurrency level of the system, thus we use the STM fallback at first. To guarantee correctness, STM transaction must be aware of executing HTM transactions. As a result, STM and HTM should communicate with each other.

Figure 5.3 shows our new lightweight communication. It has a twofold aim: it eliminates HTM false conflicts due to HTM-STM communication and it priorities HTM transactions over STM ones. HTM transactions works in two modes; plain HTM and instrumented HTM. When the entire transactional workload runs in HTM, we use plain HTM. Once, an STM transaction wants to start, it sets a flag so that all new HTM transactions start in the instrumented HTM mode. The STM transaction waits until all plain HTM transactions finish, and then starts execution. When the system finishes all STM transactions, it returns back to plain HTM mode. Every STM transaction increments `stm_counter` before starting and decrements it when finishes to keep track of active STM transaction.

In instrumented HTM mode, we keep a circular buffer (the *ring*) which contains write-set signatures of each committed HTM transaction. An HTM transaction gets an empty entry from the ring before staring the transaction (i.e., non-transactionally using a CAS operation).

During the HTM transaction, every write operation to a shared object is logged into a local write-set signature (i.e., Bloom filter). Before committing the HTM transaction, the local write-set signature is written to the reserved ring entry.

This design eliminates false conflicts due to shared HTM-STM meta data (in our case, the ring). The ring entry is reserved before starting the HTM transaction and each HTM transaction writes to its own private ring entry. For STM transactions, they read only the ring entries so that they cannot conflict with HTM transactions.

STM transactions proceed speculatively until commit phase. Before committing, it validates its read-set against concurrent HTM transactions. If it is still valid, it starts an HTM transaction where it commits its write-set (i.e., reduced hardware transaction (RHT) [76]). Before starting the commit phase of RHT, it checks the ring again to confirm that the ring itself is not changed since last validation (which was executed outside the RHT). If the ring is unchanged, then it the transaction can commit, it abort and re-validate. If the re-validation fails, then the entire transaction is restarted.

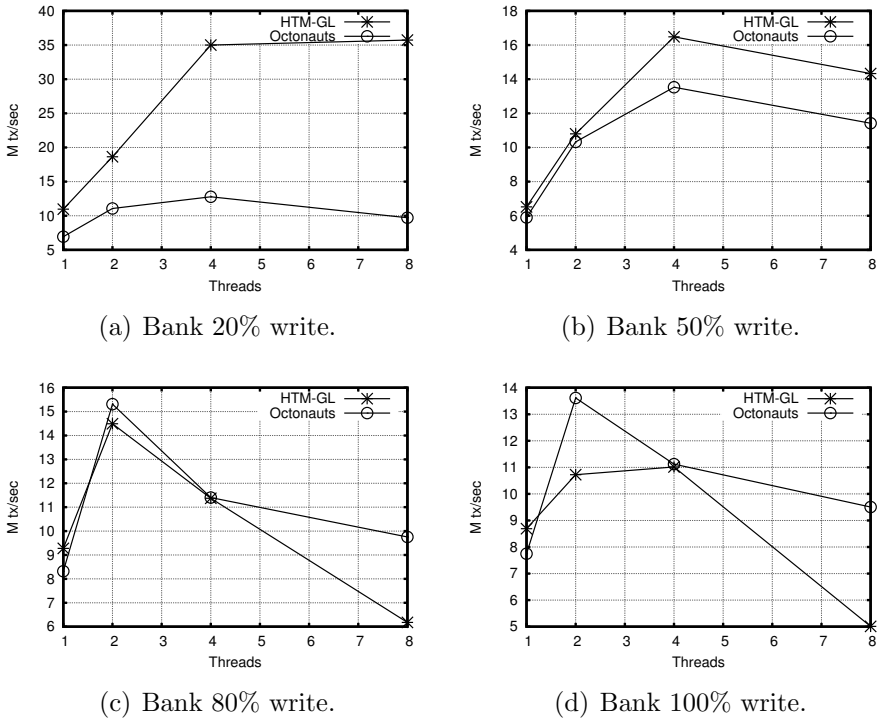
This technique seems to favor HTM transactions, but since both HTM and STM transactions subscribe to the same scheduler queues, HTM and STM transactions can only conflict due to inaccurate determination of the working-set or due to Bloom filters false conflicts. Thus, STM transaction cannot suffer from starvation.

For those transactions that cannot fit also as RHT due to their large write-set size or due to some irrevocable call (e.g., system call), the global locking path has been introduced. We implemented this path by simply adding a global lock before letting the scheduler work on the queues. A transaction that should execute in mutual exclusion, first acquires the global lock, which blocks all incoming transactions, then waits until all queues are empty before starting the execution.

5.3.3 Transactions Analysis

OCTONAUTS works based on the a priori knowledge of the transaction's working-set, which in our implementation is provided by the programmer at the time the transaction is defined. Besides the working set, there is a number of additional parameters that are useful to better characterize the transaction execution, especially having HTM as runtime environment. Our analysis estimates the transaction size, duration, number of accessed cache lines, and irrevocable action invoked. These information are used by the scheduler to adaptively start a transaction with the best fitting technique (i.e., hardware, software or global lock), as described in the following subsection.

Transaction's size and duration are estimated statically at compile time, given the underlying hardware model as input. Clearly this analysis can make mistakes, however if the estimation is wrong the transaction will be aborted but eventually will be correctly committed as software of global lock transaction. Finally, a transaction is marked as irrevocable if it call



(a) Bank 20% write.

(b) Bank 50% write.

(c) Bank 80% write.

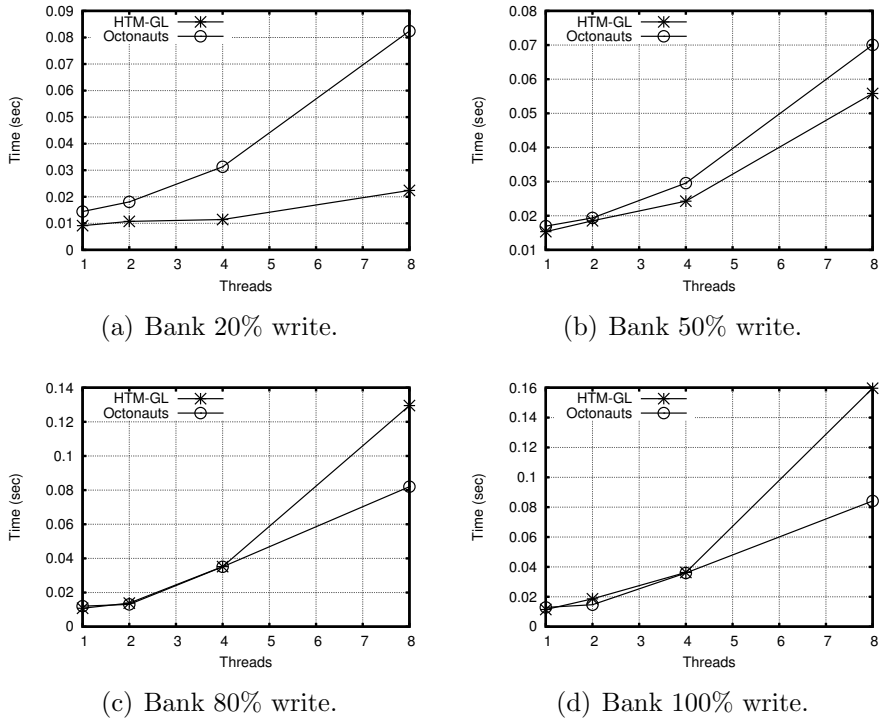
(d) Bank 100% write.

Figure 5.4: Throughput using Bank benchmark.

any irrevocable action or system call.

5.3.4 Adaptive Scheduling

The adaptivity in our scheduler is the process of selecting the right starting path for a transaction according its characteristics (e.g., data size and duration). If we know from the program analysis that a transaction does not fit in an HTM transaction, then it is started as an STM transaction from the very beginning without first trying in HTM. The same for transactions that call irrevocable operations, which are started directly using a single global lock, without trying alternative paths. We also disable scheduling queues when the contention level in the system is very low. In fact, at low contention level, scheduling queues overhead can overcome its performance benefits and slowdown the system. When the scheduling queues are disabled, a transaction starts immediately its execution without the ticketing system. However the adaptivity module is always active because it uses offline informations thus its overhead is minimal.



(a) Bank 20% write.

(b) Bank 50% write.

(c) Bank 80% write.

(d) Bank 100% write.

Figure 5.5: Execution time using Bank benchmark (lower is better).

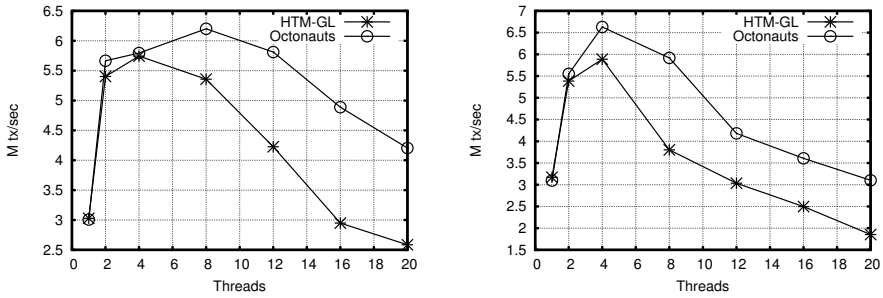
5.4 Evaluation

OCTONAUTS has been implemented in C++. To conduct our evaluation, we used two benchmarks: *Bank*, a known micro-benchmark that simulate monetary operations on a set of accounts, and TPC-C [31], the famous on-line transaction processing (OLTP) benchmark which simulates an ordering system on different warehouses. TPC-C includes five transaction profiles, three of them are write transactions and two are read-only.

We compared OCTONAUTS results to plain HTM with global locking fallback (HTM-GL). In HTM-GL, a transaction is retried 5 times before falling back to global locking. In this evaluation study we used Intel Haswell Core i7-4770 processor with hyper-threading enabled. All the data points reported are the average of 5 repeated execution.

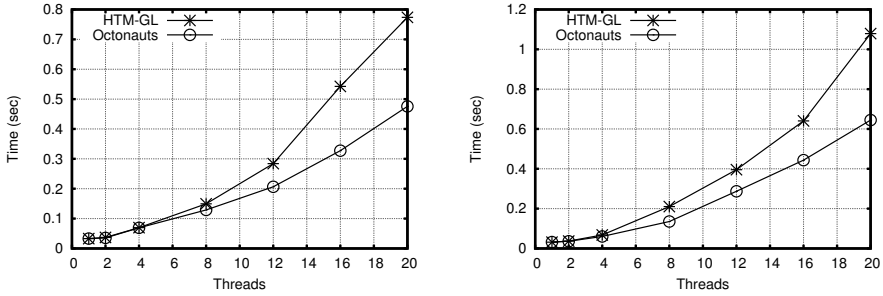
5.4.1 Bank

This benchmark simulates monetary operations on a set of accounts. It has two transactional profiles: one is a write transaction, where a money transfer is done from one account to another; the other profile is a read-only transaction, where the balance of an account is checked. The accessed accounts are randomly chosen using a uniform distribution. When



(a) TPC-C 20 warehouses. (b) TPC-C 10 warehouses.

Figure 5.6: Throughput using TPC-C benchmark.



(a) TPC-C 20 warehouses. (b) TPC-C 10 warehouses.

Figure 5.7: Execution time using TPC-C benchmark.

the number of accounts is small, the contention level is higher. For this experiment, we used 10 accounts to produce a high level of contention. Each account on a unique cache lines to guarantee that transactions accessing different account do not conflict. We changed the ratio of write transactions (20%, 50%, 80%, and 100%). Increasing the percentage of write transactions increases the contention level as well.

Figure 5.4 shows the results of Bank benchmark. From the experiments, we notice that OCTONAUTS overhead is high in low contention cases (Figure 5.4(a)). As the contention level increases (Figure 5.4(b)), the gap between OCTONAUTS and HTM-GL decreases. At high contention levels (Figures 5.4(c) and 5.4(d)), OCTONAUTS started to perform better than HTM-GL. Specially at 8 threads and 100% writes, when OCTONAUTS is 1× better than HTM-GL.

5.4.2 TPC-C

This benchmark is an on-line transaction processing (OLTP) benchmark, which simulates an e-commerce system on different warehouses. TPC-C transactions are more complex than

Bank's tarnsactions (i.e., larger in data size and longer in duration). The contention level of TPC-C benchmark can be controlled by the number of warehouses in the system. In our experiments, we used 10 and 20 warehouses to achieve a high and medium level of contention, respectively. We also used the standard TPC-C settings as a mix of transaction profiles.

Figure 5.6 shows the results of TPC-C benchmark. The conflict level in TPC-Cis high, hence OCTONAUTS is particularly effective, being able to reduce the conflicts ratio significantly. OCTONAUTS performs better than HTM-GL starting from 4 threads. This experiment shows the benefits of scheduling on workloads similar to real applications. In addition, when number of threads is larger than cores, OCTONAUTS is still able to scale. This is due to the fact that scheduling the execution of transactions properly can lead to more concurrency than leaving contending transactions to abort each other.

Chapter 6

Shield

6.1 Problem Statement

Data are becoming more precious everyday and protecting data integrity is critical to many business applications. Moreover, keeping the system alive is also critical. For example, automatic teller systems, stock trading, banking systems and many other business critical applications require both safety and liveness. The proliferation of multi/many core architectures defines the current technological trend. In such systems, the problem of tolerating/detecting data corruption is becoming more complex due to the nature of the underlying hardware [11].

As an example, *soft-errors* [16] belong to the category of hardware-related errors and they are very difficult to detect or expect/predict. Specifically, they are transient faults that may happen anytime during application execution. They are caused by physical phenomena [11], e.g., cosmic particle strikes or electric noise, which cannot be directly managed by application designers or administrators. As a result, when a soft-error occurs, the hardware is not affected by interruption, but applications may crash or behave incorrectly.

The trend of building smaller devices with increasing number of transistors on the same chip is allowing designers to assemble powerful computing architectures – e.g., multicore processors. Although the soft-error rate of a single transistor has been almost stable over the last years, the error rate is growing in current and emerging multicore architectures due to rapidly increasing core counts [11]. Also, as an increasing number of enterprise-class applications, especially those that are business-critical (e.g., transactional applications), are being built for, or migrated onto such architectures, the impact of transient failures can be significant.

Shield focuses on those transient faults that can occur inside processors (e.g., multicore CPU), called hereafter *CPU-tfauls*. Specifically, we center our attention on soft-errors as good representative of CPU-tfauls because they are random, hard to detect, and can cor-

rupt data – e.g., a soft-error can cause a single bit to flip in a CPU register due to the residual charge of a transistor, which inverts its state. Most of the time, such an event is likely to be unnoticed by applications because they do not use that value (e.g., an unused register). However, sometimes, the register can contain a memory pointer or a meaningful value. In those cases, the application behavior can be unexpected. An easy solution for recovering from transient faults is a simple application-restart, but such solutions are unacceptable for transactional applications due to availability/reliability constraints, and also due to performance and business reasons.

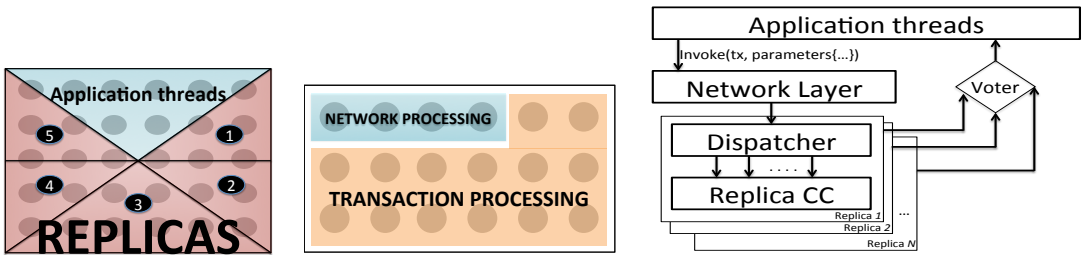
A famous example for data corruption effect in production systems is the major outage of Amazon S3 service[1]. In this incident, a single-bit flip corrupted a message and the effect propagated from one server to another. They managed to fix it by taking the system down, clearing the system state, and restoring the system back. This part of the system was not protected against data corruption errors. The downtime was over seven hours and initiated by a single CPU-tfault.

Motivated by these observations, we propose SHIELD, a software layer for tolerating CPU-tfauls in transactional systems running on multicore architectures. SHIELD's key idea is to partition available resources and run transactions on all the partitions in parallel, according to a novel scheme inspired by the state-machine replication paradigm [85]. SHIELD's most concern is to prevent any data stored in CPU registers from moving to the system's main memory where the shared state is saved, without being verified. We restrict to processor's data integrity because we assume the memory is *reliable* due to several well-known mechanisms (mainly hardware) (e.g., Error-Correcting Code), which protect memory chips from corruption. Similar mechanisms are not adopted in customer-level CPU because of their high cost of deployment.

SHIELD is designed for transactional systems, which are applications where threads generate transactions consecutively upon clients' requests. In those applications, threads are usually stateless because their state is stored into the shared data. SHIELD does not protect threads' execution outside transactions because we consider it as an orthogonal problem, which can be solved employing alternative solutions. As an example, a thread is spawned for handling a client request and its goal is to activate a transaction with user-defined parameters. SHIELD considers the transaction, together with its parameters, as an input and it cannot verify if a CPU-tfaul corrupted the values of those parameters before issuing the transaction, because it is not aware of the original client's request. However, these events (e.g., the client request) are usually logged into stable storage by any transactional system, thus a sanity check traversing logged events backward can detect those transactions activated with erroneous parameters and compensating their actions accordingly. Rather, SHIELD guarantees that the objects belonging to the shared data-set are never undesirably corrupted by CPU-tfauls that happen in the physical processor where the transaction runs.

Technically, as in state-machine replication, SHIELD wraps transaction requests into network

[1] http://status.aws.amazon.com/s3-20080720.html



(a) System (b) Replica (c) Software

Figure 6.1: System, replica and software architecture.

messages that are submited to an ordering layer. This layer ensures a global serialization order (GSO) among partitions. Each partition is thus notified with the same sequence of transactions and is able to process those transactions independently from the other partitions. Objects accessed by transactions are fully replicated so that each partition can access its local copy, avoiding further communications. Transactions are processed according to the GSO. Additionally, we use a *voter* for collecting the outcome of transactions and delivering the common response (i.e., majority of the replies) to the application. If one or more replies differ, actions are triggered to restore the consistent status of data in those faulty partitions. Assuming the majority of partitions are correct (i.e., non-faulty), the voter just selects the proper transaction reply to deliver back to the application thread and uses the collected replies that differ from the one delivered to the application for identifying faulty partitions.

SHIELD embeds optimizations for reducing network protocol latency and delivering requests in one communication step. Also it increases concurrency in request processing in order to maximize the multicore systems' utilization.

We implemented SHIELD in C++. All its components share a synchronized clock, which is available in hardware in all modern (multicore) architectures. We use in-memory transactional applications (e.g., [100]) as a candidate for evaluating our proposal. Such applications are good candidates as they typically use multicore architectures and do not use stable storage to log their actions, both for achieving high performance.

Our evaluation centers on shared-memory multicore architectures spanning from those based on bus (i.e., 48-core x86 AMD machine) to recent message-passing solutions (36-core Tilera TILE-Gx family [103]). Results, using well-known transactional benchmarks such as TPC-C [31] and STAMP [21], reveal that SHIELD ensures fault-tolerance with competitive performance of centralized (i.e., non fault-tolerant) systems, without paying the cost (both in terms of performance degradation and financial cost) of deploying a distributed infrastructure.

SHIELD makes the following contributions: *i)* A new network protocol for ordering requests, which is optimized for centralized systems and supports optimistic delivery; *ii)* An innovative concurrency control algorithm that supports the activation of parallel transactions while preserving a pre-defined commit order; *iii)* A comprehensive evaluation on message-passing-

based and bus-based shared memory architecture.

6.2 Is Byzantine Fault Tolerance the Solution?

CPU-tfaults are transient faults, and those class of faults belongs by itself to the category of *Byzantine Faults* (BF) [67]. A BF is an arbitrary fault that can generate incorrect response or corrupt the system state. BFs include commission and omission faults.

Solutions targeting BFs, named also *Byzantine Fault Tolerant* (BFT) protocols (e.g., [22, 28]), are usually designed for minimizing the assumptions on the correctness and trustiness of components composing the execution environment, as well as for being resilient to malicious behaviors. As a result, their impact on system's performance could be much higher than what is actually needed for solving the problem of CPU-tfault which, indeed, can be seen as a small part of the bigger picture of BFT. As an example, our solution does not target any malicious application behavior, security hazard or node crash. In addition, BTF solutions often require a physical multi-node distributed system to isolate nodes each other for avoiding the propagation of faults. However, replicating centralized systems for tolerating faults results in significantly degraded performance (e.g., 10–100×) [95]. This is primarily due to the costs for remote synchronization and communication that are incurred to ensure node consistency. Also, a distributed architecture comprising of multicore nodes may not often be cost-effective.

BFT systems handle malicious client behavior and unreliable networks that can reorder, drop, or corrupt messages. In addition, most BFT systems require $3f+1$ nodes to reach agreement and $2f+1$ nodes for the transaction execution in order to tolerate up to f faulty nodes [112]. In a centralized system, clients can be trusted since they are local application threads. Also, a reliable network (e.g., the bus or most message passing architectures [106, 103]) is intrinsically available or can be easily built.

SHIELD is meant to be a transparent software layer that can be plugged into a classical centralized transactional system without deploying a distributed infrastructure or significantly impacting the performance of the original (i.e., non fault tolerant) system. We believe adopting a BFT solution means equipping the transactional system with a number functionalities for tolerating faults, which negatively impact its performance without being actually exploited for tolerating faults like CPU-tfaults. In order to support this claim, in our evaluation we included a reference BFT protocol [22] and showed that its performance are much slower than SHIELD.

6.3 System Model and Assumptions

We consider a synchronous system based on the message-passing abstraction, where a set of nodes $\{N_1,N_2,\ldots, N_i\}$, installed on the same physical hardware, communicate each other. Each node is a group of computing cores on a multicore system. We assume a shared physical clock available for the entire system (see Section 6.6 for details). Message delivery time is assumed to be bounded. The communication infrastructure is assumed reliable such that no messages are lost. On x86 systems, we use reliable shared memory queues to implement the communication infrastructure.

Application's shared data objects are replicated such that each node accesses its own copy. This way, storing a value from a CPU-register to a memory location does not interfere with the work of other nodes, which prevents the propagation of a possible fault to other nodes. When a fault is detected on a node, a recovery mechanism is in charge of replacing that node's corrupted objects with consistent objects from another non-faulty node.

Our approach has a major drawback, it limits the overall system's resources to the application due to the partitioning. In particular, SHIELD reduces the amount of memory available, as well as the number of cores reserved per application thread. Regarding the former, we believe the amount of memory available nowadays on a single node is large enough to satisfy most of application needs. Regarding the latter, we assume enough cores available per application thread such that the application can still run without affecting its performance significantly. We recall that we target transactional applications, which have often scalability issues due to the logical contention of accessed objects. As a result, they are often not able to fully exploit the underlying hardware parallelism, therefore, even reducing the amount of cores available per application thread, it should not mean a necessarily reduction in performance.

To tolerate f faults (i.e., CPU-tfaults), our system requires $2f+1$ nodes, such that a majority can always being formed and the voting procedure can take place. Our system can tolerate f faults over the lifetime of the system as long as no more than f faults occur during the bounded recovery time window.

6.4 Fault Model

SHIELD targets CPU-tfault, which are transient faults that occur inside the processor and are not still propagated to the memory. SHIELD does not prevent any value corrupted from a CPU-tfault to be stored in memory, rather it cares only of those values that will affect the shared data-set. As an example, if a node is affected by a CPU-tfault, it can still write the corrupted value in its private memory space, but it will never overwrite the location storing that value in the shared data-set. In such a case, SHIELD uses the outcome of other nodes to select a majority and writing back the proper value to the shared data-set.

SHIELD works on a single machine, thus it cannot tolerate a crash or a permanent hardware failure of the machine. Asynchronous checkpointing to stable storage can be used to tolerate such failures. Similar to other replication-based techniques, SHIELD cannot tolerate deterministic software bugs or incorrect configurations. A deterministic bug will occur on all replicas and generate the same results. Thus, the parallel execution and voting will not detect it. This problem could be solved using other techniques, like diversity[2](e.g., [28, 114]).

Exploiting its capability, SHIELD can also detect some random software bugs (i.e., non CPU-tfault), e.g., race condition and other concurrency bugs, that happen on one node, due to the particular schedule of operations generated, but not on others. However, SHIELD cannot ensure that these errors are caught because if they happen on a majority of nodes, no common decision can be taken by the voter and system's resiliency could be broken. If those errors are less than the maximum number of faults admitted by SHIELD (f as in Section 6.3), then it is also capable to detect and solve them.

6.5 Shield Design

6.5.1 Overview

SHIELD's basic idea for tolerating faults is to logically partition computational resources into groups. Each group is composed of a subset of available cores, and is responsible for processing transactional requests issued by application threads. We use the state-machine replication paradigm for processing transactions. A transaction request is sent to all the groups (or *replicas* hereafter), relying on a group communication system (GCS). (Applications do not execute transactions in the same requesting thread.) GCS, implementing a total order service (e.g., Atomic Broadcast [37]), is responsible for delivering the same set of transaction requests to all the replicas, in the same order. Processing the same sequence of transactions allows replicas to reach the same state.

Unfortunately, total order protocols involve the exchange of several network messages for each transactional request to agree on a common delivery order, which is often a performance bottleneck [95]. To alleviate this, we rely on an optimistic version of the total order specification, as proposed in [59]. This new abstraction includes an additional delivery, called *optimistic delivery*, which is notified by the total order layer to the replicas before the (*final*) notification of the message, along with its order. This early delivery has a twofold benefit. First, early delivery notifies replicas that a new message has been previously broadcast and it is currently in the coordination phase for defining its final order. Second, early delivery defines an implicit optimistic order. Although this order can be used for executing transactions speculatively, overlapping their processing with their coordination, it is not reliable

[2]Each replica produces the same results via different approaches (e.g., different versions of the same application.)

and cannot be considered for the transaction commit phase. As a result, when the optimistic order and the final order coincide, the transaction can be validated and committed (if completely executed) without paying the cost of its re-execution from scratch.

An ordering-based concurrency control (ObCC) protocol, running locally on each replica, is responsible for committing transactions following the order defined by the sequence of final deliveries issued by the GCS. A subset of cores in each replica is dedicated for the execution of ObCC. In order to maximize the overlapping of (speculative) transaction processing with the total order protocol, transactions are processed in parallel (enforcing a pre-defined order) as soon as they are optimistically delivered. When a conflict arises, ObCC resolves it in order to meet the order defined by the request timestamp (optimistic order). Whenever the optimistic order is the same as the final order, or the transaction does not conflict with others, the transaction can be safely committed after validation. The entire shared data-set is replicated on each replica. This enables ObCC to process transactions completely locally, making each replica independent of the others.

ObCC's effectiveness mostly depends on the likelihood of having an optimistic order that is *equivalent* to the final order. Consider four transactions. Suppose $\{T_1,T_2,T_3,T_4\}$ is their optimistic order and $\{T_1,T_4,T_3,T_2\}$ is their final order. Assume that the transactions are completely executed when their respective final deliveries are issued. When T_4 is final-delivered, ObCC observes that T_4's optimistic order is different from its final order. However, if T_4 does not conflict with T_3 and T_2, then its serialization order, realized during execution, allows the transaction to commit without re-execution (we call T_4's serialization order equivalent to its final order). On the other hand, if T_4 conflicts with T_3 and/or T_2, then T_4 must be aborted and restarted in order to ensure replica consistency. If conflicting transactions are not committed in the same order on all the replicas, then replicas could end up with different states of the shared data-set.

When a fault occurs on a replica, the other replicas are able to serve the transaction request without additional coordination i.e., the failure is *fully masked*. A *voter* is in charge of collecting outcomes from the transactions processed on all the replicas, and returning the majority outcome to the application. Even though this approach potentially increases the end-to-end transaction latency, all the replicas are part of the same architecture, running at the same clock speed, and are exposed to the same workload. As a consequence, the transactions' results are received almost simultaneously by the voter, without affecting the performance. The voter also identifies the replicas that have faults (i.e., presenting an outcome that differs from the one delivered to the application), and restores a consistent state of their memory from a correct (i.e., non-faulty) replica. Using Transactional Memory (TM) simplifies the voter comparison procedure. TM keeps track of all written objects in the *write-set*. We do not use write-set hash-based signatures because hash collisions (when two values map to the same hashed value), can hide some faults. If the voter is faulty, the shared data's integrity and consistency are not affected. In the worst case, the voter will start a data restoration for a correct replica, which has no effect on shared data.

When the voter restarts a faulty replica, it enters in recovery mode and starts copying the state of a correct non-faulty replica. The copy is incremental. The non-faulty replica keeps track of all objects modified during the copying process so that it can still serve new transactions. This incremental state is then pushed to the faulty replica for finalizing the copy.

6.5.2 Limitations

Though SHIELD overcomes CPU-tfaults transparently from the application's point of view, it has three main drawbacks:

- First is the reduced number of cores available for transaction processing. However, SHIELD targets massively multicore architectures, where the number of cores is sufficiently large for exploiting application concurrency. Additionally, transactional applications typically do not consume all the resources available because of logical contentions (e.g., locks). In SHIELD, the number of replicas can be tuned according to the degree of fault resilience desired, and also according to the expected resource utilization.
- The second drawback is the increased memory consumption. In order to support SHIELD's architecture, each group replicates all the shared objects, increasing the total memory utilized. However, this is not a major restriction to many applications because memory cost is rapidly decreasing and many medium-level servers are now equipped with 16/32/64GB. Moreover, to protect data from corruption, we need more than one isolated copy of the data.
- The third drawback is the energy consumption. Replication increases consumption, but it also allows error detection and recovery [109]. Effective error detection and recovery mechanisms have often a negative impact on energy because they entail redundancy. Reducing energy's overhead is still an open issue which we plan to address as a future work.

6.6 Network Layer

We propose an optimized approach that is tailored for centralized systems, where the interconnection medium for node-to-node communication is fast and reliable. As stated in Section 6.3, we assume a hardware infrastructure that is equipped with a monotonically increasing synchronized clock, called *clock-service* hereafter. This assumption does not limit the applicability of SHIELD, because all modern multicore architectures provide this feature. SHIELD targets multicore systems based on both message passing and bus as communication infrastructure. In the former, the cycle counter register in each core serves as clock-service. In the latter, we adopt a technique for building a clock-service starting from the physical clocks (we postpone this discussion to Section 6.8.2).

Our network protocol assumes a reliable and FIFO physical communication infrastructure.

In other words, if a node N_a sends two messages m_1 and m_2 to a node N_b in that order, N_b delivers m_2 only after m_1. No message can be lost. We believe this assumption is not stringent in centralized architecture.

We design a protocol for establishing a total order of messages (i.e., transaction requests) issued by application threads, in the presence of CPU-tfaults. Our protocol is inspired by Lamport's algorithm [64]. We name the component that is responsible for implementing this protocol as the *network layer*. The network layer is decentralized, i.e., each replica reserves one core for executing its part of the work. SHIELD network protocol is client-centric where clients are the application threads. The agreement on the order of messages relies on a synchronized clock-service available at each client and replica. It provides optimistic delivery in one communication step only. Final order is determined in two steps in low contention execution or in one step in high contention execution. This enables the algorithm's usage in centralized settings without significant performance degradation with respect to the execution of standalone, non fault-tolerant applications.

Each application thread sends its requests (`tx_request`) to all the replicas and sends associated acknowledgment-requests (`ack_request`) to other application threads. According to our programming model, a `tx_request` message contains: the application thread ID; the name of the transaction to execute along with its parameters (if any); and the current local timestamp (i.e., the current value of the clock-service). An `ack_request` message contains the ID and the current timestamp only.

When a replica receives a new `tx_request` message, it immediately triggers the optimistic delivery for that message using the message's timestamp as the optimistic order. Since messages cannot be lost in our architecture, the replicas do not reply with an *ack* to other replicas or application threads. However, a node N_a receiving a message m_x does not know whether m_x can be final-delivered or if there is another message m_y that was sent before m_x (from a different application thread) that is currently in transit in the message-passing infrastructure. m_y can only be sent by an application thread different from the one that sent m_x. Otherwise, due to the thread-FIFO property, m_y would have been delivered before m_x.

For this reason, before issuing the final delivery for a message m_x with timestamp ts_x, a replica waits until all other application threads notify their current local timestamp. Exploiting the thread-FIFO property, if a replica receives a notification from all other application threads with a timestamp greater than ts_x, it means that all the previous sent messages have been delivered. If some application threads send a notification with a timestamp lesser than ts_x, then the final delivery cannot be issued and the replica waits until new notifications for m_x are received from those application threads.

At this stage, each replica simply selects the message with the minimum timestamp and triggers the final delivery for that message. In order to minimize the messages sent through the message-passing system, we further optimize the notification sent by application threads after receiving a new `ack_request` message from another application thread. Since replicas need only the timestamp of each application thread, the total number of messages sent can

be reduced by merging this notification with the broadcast of a new `tx_request` message. In other words, when an application thread receives an `ack_request` message, if it has a new `tx_request` message to broadcast, it does not just send the timestamp and then perform the broadcast, but rather, directly broadcasts. When a replica that is waiting for the notification from that application thread receives a new `tx_request` message broadcast from that thread, it extracts the timestamp associated with the new message and considers it as the notification. This reduces the delay of the final delivery when the system is under high-load (from two steps to one step) and reduces also the load on the underlying physical network infrastructure.

Since all clocks are synchronized and monotonically increasing, and the network is reliable, the same scenario occurs in all other replicas, resulting in the same final delivery order.

The logic executed by the network layer can be split in two: one executed at the application side (Algorithm 6.2), and one executed at the replica side (Algorithm 6.3).

```
while(true) {
   bool ack_requested = false;
   while(app_queue.dequeue())
       ack_requested = true;
   if (!app_requests.empty()) {
      ack_requested = false;
      for (i=0; i < replica_count; i++)
         send_tx_request(replica[i], app_requests.dequeue());
      for (i=0; i < application_count; i++)
         if (i != id)
            send_ack_request(app[i]);
   }
   if (ack_requested)
      for (i=0; i < replica_count; i++)
         send_ack_msg(replica[i]);
}
```

Figure 6.2: Network Layer Protocol (Application side).

6.6.1 Tolerating CPU-tfaults

If a CPU-tfault affects the special register that holds the clock value, SHIELD will detect that error and resynchronize the clocks again. A bit flip in the clock register can corrupt the clock value, but the new value will continue to increase after that error as before since this register is incremented with every cycle. We can identify three cases for the clock register's soft-error:

- The first case is a large shift to the past or future. This is easily detected by the replica by comparing the request timestamp with the last timestamp from the same client.
- The second case is a small shift to the past or future. In this case, the protocol continues to order requests as before, but some acknowledgments will be dropped. As long as all application threads are sending requests, the shift will not be detected and the system will

```
while (true) {
      msg = receive_msg();
   max_timestamp_seen[msg.source] = msg.timestamp;
   if (msg.data == DATA_MSG) {
       msgs_queue[msg.source].enqueue(msg);
       opt_deliver_msg(msg);
   }
   found = true;
   while (found) {
      min = INFINITY;
      min_index = 0;
      for (i=0; i < application_count; i++)
         if (!msgs_queue[i].empty())
            if (msgs_queue[i].top().timestamp < min) {
               min = msgs_queue[i].top().timestamp;
               min_index = i;
            }
      if (min == INFINITY) {
         found = false;
         break;
      } else
         for (i=0; i < application_count; i++)
            if (max_timestamp_seen[i] < min && i != min_index) {
               found = false;
               break;
            }
      if (found) {
         head_msg = msgs_queue[min_index].dequeue();
         final_deliver_msg(head_msg);
      }
   }
}
```

Figure 6.3: Network Layer Protocol (Replica side).

continue to order requests uninterrupted. When requests stop, some of the last requests will likely be stuck in the replica queues. A timeout is used to detect this case. The timeout is set to the maximum network delay.

- The third case is a soft-error that occurs while copying the clock value to the request message. In this case, one request will be sent with a different timestamp to one of the replicas. This error is detected by a timeout in the voter when a replica response is delayed.

6.7 Replica Concurrency Control

We propose a new concurrency control algorithm, called ObCC. The main goals of ObCC is to process transactions in-order, leveraging the optimistic order, and to reduce the instrumentation overhead as much as possible for the next-to-commit transaction (also called the *committer* transaction). ObCC does not involve remote interactions (it is fully local).

Transactions are activated as soon as they are optimistically delivered (*opt-del* hereafter). The timestamps of opt-dels define the optimistic order. When two or more opt-del trans-

actions conflict, it is resolved using the optimistic order. Consider two transactions T_1 and T_2. Let their opt-del order be T_1 followed by T_2. When a conflict occurs, T_2 is aborted and restarted, allowing T_2 to access data written by T_1. Non-conflicting opt-del transactions are processed without incurring any aborts because their processing order is equivalent to any other order.

Transactions are classified as *speculative* and *non-speculative*. A transaction is speculative when it is activated but its final order is still undefined (i.e., the network layer has not yet triggered its final delivery). The transaction becomes non-speculative when its final delivery has been issued, thereby defining its final order. Transactional read and write operations are locally buffered in private structures called read-set and write-set, respectively.

ObCC uses write-locks that are acquired at encounter time before performing the actual write operation. A transaction T_w writing object X must acquire the write-lock on X[3]. If X is locked by another transaction T_k that is older than T_w (according to the optimistic order), then T_w waits until T_k commits. If T_k follows T_w in the optimistic order, then T_k is aborted. Exploiting this mechanism, conflicting transactions are serialized according to the optimistic order.

When a transaction T_r reads an object Y, locked by a transaction T_j and T_j precedes T_r, then T_r waits until that transaction finishes. In the opposite case, T_r ignores the lock and reads the object because it is serialized before T_j, therefore T_j's written object is not visible to T_r.

Only one thread is allowed to commit at a time. This is because ObCC must enforce the final order defined by the network layer as the commit order. No concurrency is allowed. Say three transactions are optimistically delivered in the order $\{T_1, T_2, T_3\}$ and final-delivered in the order $\{T_3, T_1, T_2\}$. The transaction that receives the permission to commit first is therefore T_3, even though it is the last according to the optimistic order. Suppose the final delivery for T_1 arrives before T_3 commits. T_3's commit phase cannot start because T_1 executed speculatively on a different serialization order. Therefore, objects accessed by T_3 could be invalid and its written objects could be inconsistent. Starting T_1's commit after T_3 successfully finalizes its commit guarantees a transaction history that is compliant with the final order.

When the final order for a transaction is defined, and it corresponds to the next transaction to commit, that transaction has the highest priority in the system and must be committed fast because the application is waiting for its reply. ObCC detects when a speculative transaction becomes non-speculative by simply checking whether its final order has been defined. If the final order corresponds to the next transaction to commit, that transaction's priority is made the highest. We call this execution status: *committer mode*.

The main advantage of processing a transaction in the committer mode is that the transac-

[3]A memory location can either be an object or a memory address. We use the memory address, but SHIELD is independent.

tion executes with very low instrumentation (e.g., it does not compete with other transactions while accessing objects). The goal of the committer mode is to commit the highest priority transaction as soon as possible. In state-machine replication, transactions cannot be committed before the delivery of their final order. However, when the order is defined, all the extra time spent in processing before the transaction is committed must be minimized to avoid increasing the transaction's latency, and thus degrading application performance.

The committer thread (i.e., the thread processing a transaction in the committer mode) writes directly to shared memory, and reads without instrumentation. A thread can be shifted to the committer mode at anytime, while operations are executing, or when the entire transaction has completed and the thread is waiting for its final commit order. If a thread becomes the committer after executing all the transactional operations, it validates its read-set and commits its written objects to shared memory. No additional operations are required because this thread already has all the write-locks necessary for changing the shared objects. After this, it updates the timestamps of all the locks and release them.

If a thread's promotion to committer happens when the transaction is still executing, the thread validates the current status of its read-set, commits its written objects to shared memory and keeps executing. For each new write operation, the committer thread acquires a new type of lock, called *super-lock*. This lock is implemented such that no other threads can compete with it. Therefore, no CAS (Compare-And Swap) or atomic operation is required, allowing the committer thread to proceed without blocking its execution on any synchronization point. Due to the serial execution of commit, there is only one committer thread that is active in the system. For this reason, the super-lock does not need an atomic operation for its acquisition. Clearly, when a super-lock is acquired on an object, no other transactions are allowed to write on that object. Determinism is guaranteed by committing all transactions with the same order on each replica. Moreover, non-deterministic operations like `random` and `time` are not allowed on replicas. Clients send these values with a transaction request so that all replicas use the same non-deterministic value.

There is only one committer thread at a time that executes with the highest priority following the total order already established. No concurrency is allowed on the commit process and speculative transactions are always validated before to enter in the committer mode. Therefore, it is straightforward to prove that SHIELD guarantees one copy serializability [13].

Begin. When a transaction begins, it records the current timestamp (taken from clock-service) in `readTS`, which is used for validating the transaction read-set, and the opt-del timestamp in `timestamp`, which is used for ordering. The transaction sets the final order (`TOorder`) to infinity. During transaction execution, `TOorder` is set to the final delivery order. Setting `TOorder` during transaction execution is different from changing the transaction to committer mode. Since a transaction that is final-delivered is serialized before any other speculative transaction, `TOorder` overcomes timestamp-based ordering (i.e., the optimistic order). Many transactions may know their final order, but only one is in the committer mode (i.e., the next transaction permitted to commit).

```
word txRead(address, tx) {
      wLock = Orecs.get(address);
      if (wLock.owner == tx)
             return writeSet.get(address);
      do {
      while(wLock.isSuperLocked || (wLock.isLocked && wLock.owner (precedes) tx))
          waitForUnlock();
      v1 = wLock.timestamp;
      value = memory[address];
      v2 = wLock.timestamp;
      } while(v1 != v2);
             if (v1 > tx.readTS)
                    validate(tx);//or abort if validation failed
             readSet.add(address);
             return value;
}
word committer_txRead(address, tx) {
       return memory[address];
}
```

Figure 6.4: Transaction's read procedure.

Read. When a transaction T_r reads an object X, ObCC first checks whether X is already in the write-set (i.e., the write-lock on X has been acquired by T_r) and, if so, returns X's value (Figure 6.4). Otherwise, ObCC checks whether the committer thread or another thread preceding it has acquired the super-lock or the write-lock respectively. In both cases, T_r waits until X is unlocked. When no other preceding thread has locked X, the read operation can be performed. In order to guarantee that no transaction wrote to the object during the read operation, X's timestamp is checked before and after the read. When the read is complete, X's timestamp is compared to T_r's timestamp. If X's timestamp is greater than T_r's timestamp, then it means that other transactions in the system have been committed after T_r began executing. Therefore, ObCC validates T_r's entire read-set to be sure that all the objects accessed are still consistent. If validation fails, T_r is aborted and restarted. The read operation ends by adding the object to the read-set.

A read operation performed by the committer thread (`committer_txRead`) does not follow the previous rules; it directly accesses the value in shared memory.

Write. When a transaction T_w writes to an object X, ObCC first checks whether X is already in the write-set (Figure 6.5). If so, ObCC just updates the value in the write-set. Otherwise, if X is locked by the committer thread or by a transaction serialized before T_w, then T_w waits until X is unlocked. At this stage, as already described in the read operation, ObCC validates T_w's entire read-set because the transaction that released the write-lock could have invalidated some objects read by T_w. If validation fails, T_w is aborted and restarted.

If X is locked by a transaction serialized after T_w, then the current lock holder is aborted and restarted, because its execution order is not compliant with the optimistic order. The write completes when the lock is successfully acquired by T_w and the object's value is added to the write-set.

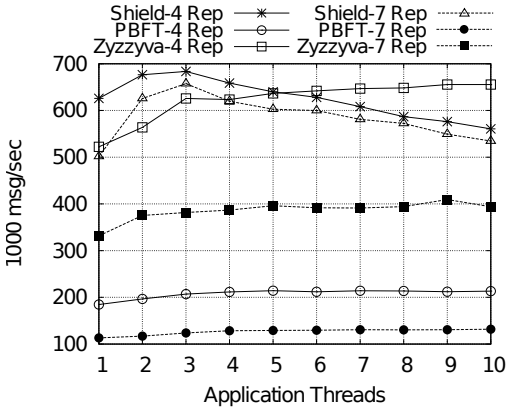
```
void txWrite(address, value, tx) {
      wLock = Orecs.get(address);
      if (wLock.owner == tx) {
             writeSet.update(address, value);
             return;
      }
      do {
             if (wLock.isLocked)
                if(wLock.isSuperLocked || wLock.owner(precedes)tx){
            waitForUnlock();
            validate();
                } else
            abort(wLock.owner);
      } while( CAS(wLock.lock, UNLOCKED, LOCKED));
      wLock.owner = tx;
      writeSet.add(address, value);
}
void committer_txWrite(address, value, tx) {
       wLock = Orecs.get(address);
       if (wLock.owner != tx) {
              abort(wLock.owner);
              wLock.superLock = LOCKED;
              aquiredLocks.add(address);
              wLock.owner = tx;
       }
       memory[address] = value;
}
```

Figure 6.5: Transaction's write procedure.

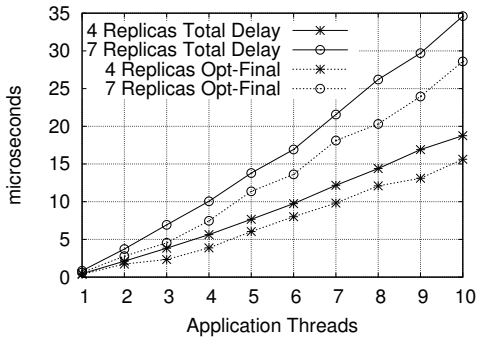
The committer thread doing a write operation (`committer_txWrite`) writes directly to shared memory if it has acquired a lock on X. If X is locked, the transaction holding the lock is immediately aborted and restarted, because the committer thread has the highest priority to execute and commit. The super-lock is acquired on all the unlocked written objects.

Validation, Commit and Committer. ObCC performs the validation of a transaction by relying on timestamps. The procedure compares the timestamps of all objects in the read-set with the transaction's timestamp (`readTS`). If validation succeeds, i.e., all the object timestamps are smaller than transaction's `readTS`, then the transaction's `readTS` is advanced to the current system timestamp. This minimizes the number of invocations of the validation procedure in the future.

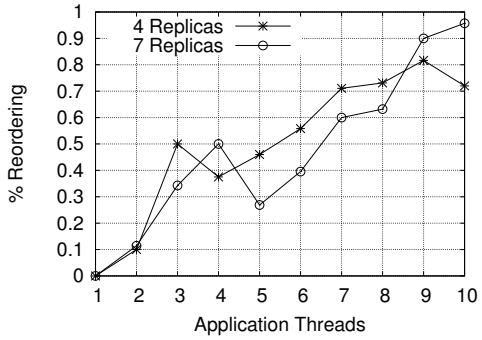
When a transaction is final-delivered and it is the next to commit, it calls `transitionToCommitter` for entering the committer mode. ObCC triggers a validation and if validation succeeds, then the thread commits its written objects to shared memory (if any) and enters the committer mode. A read-only transaction also validates in order to follow the final delivery order. Only the committer thread can commit a transaction. The commit phase is lightweight. The transaction releases all the held write-locks and super-locks, and update each lock's timestamp.



(a) Throughput of network layer



(b) Latency of network layer



(c) Reordering between opt-del and final-del

Figure 6.6: Performance of network layer on Tilera.

6.8 Evaluation

All the modules of SHIELD are implemented in C++. The network layer was designed and implemented to be platform-independent: it can be configured to run on both message-passing as well as shared memory architectures. For space constrains, we report the full evaluation using the message-passing architecture in Section 6.8.1 and we briefly discuss results obtained on the x86 machine in Section 6.8.2.

6.8.1 Tilera TILE-Gx family

Our test-bed for message-passing architecture consists of a 36-core machine of the Tilera TILE-Gx family [103]. This hardware is commonly used as an accelerator or an intelligent network card. Each core is a full-featured 64-bit processor (1.0 GHz), with two levels of caches, 8 GB DDR3 memory, and has a non-blocking mesh that connects cores to the Tilera 2D-interconnection system. For x86 shared memory architecture, we used a 48-core machine.

The machine has four AMD Opteron™ Processors (6164 HE), each with 12 cores running at 1700 MHz, and 16 GB of memory.

We implemented ObCC in the Rochester STM (RSTM) library [74]. RSTM uses platform-specific assembly instructions. Therefore, we ported its original implementation to our test-bed. As a result, ObCC is compliant with all the platforms already supported by RSTM and the Tilera TILE-Gx family.

Network Layer. In order to evaluate the performance of the network layer without transactional workloads, we conducted an experimental study by varying the number of application threads and fixing the number of replicas to {4, 7}. We implemented two well known BFT systems: PBFT [22] and Zyzzyva [62]. PBFT is one of the state-of-the-art BFT protocols. PBFT uses a three-phase ordering protocol and a request is totally ordered after completing the three phases. Zyzzyva is a client-centric protocol that uses only two-hop communication in non-faulty operation of the system. Both PBFT and Zyzzyva use $3f + 1$ replicas to tolerate f faults. Thus, we selected 4 and 7 replicas in our experiments although SHIELD requires only $2f + 1$ replicas to tolerate the same number of faults. Our PBFT and Zyzzyva implementations are simplified without message authentication which is not required in centralized systems where messages are originating from a local application thread.

In SHIELD, each replica reserves one core for running its own instance of the network layer. We configured the whole system such that each thread runs on its own core. A dispatcher (per replica) is used for managing network messages and triggering ObCC upon the receipt of optimistic and final deliveries. In this experimental study, dispatchers do not activate ObCC, but they only log network messages' meta-data for collecting statistics. SHIELD's network layer implementation does not batch messages for reducing transaction latency.

We varied the number of application threads and measured the network throughput (i.e., number of messages ordered per second). We also measured the average latency between broadcast and final delivery of a message, as well as the one between optimistic and final delivery. We also measured the mismatch probability between optimistic and final order. Since all replicas execute at the same clock speed, the gap between the execution times of the fastest and slowest is negligible[4].

Figure 6.6(a) shows the network throughput. Two factors affect the network throughput: the total number of messages per request and the number of hops/stages until final order agreement. The number of messages per request depends on the number of replicas only in both PBFT and Zyzzyva. In SHIELD, it depends on three factors: number of replicas, number of clients, and the system load. PBFT's request-ordering takes three stages and the total number of messages per request is 28 and 91 for 4 and 7 replicas respectively. Zyzzyva's request-ordering takes two hops but the number of messages per request is small and equals the number of replicas. SHIELD, when under high load, takes one hop (a client does not need to send an acknowledgment when it has an application message ready for sending) and the

[4]Automatic CPU frequency scaling is disabled.

total number of messages per request equals $num\_of\_replicas + num\_of\_clients - 1$. This clarifies why PBFT throughput is low compared to Zyzzyva and SHIELD. Zyzzyva is slightly better than SHIELD at 4 replicas and a high number of clients. But the two hops' effect is clear at 7 replicas. SHIELD performance is slightly affected by increasing the number of replicas. With 4 replicas, SHIELD was similar to Zyzzyva in the range of +20% to -14%. At 7 replicas, SHIELD is on average 52% better than Zyzzyva. Compared to PBFT, SHIELD is 2× and 3.6× better with 4 and 7 replicas respectively.

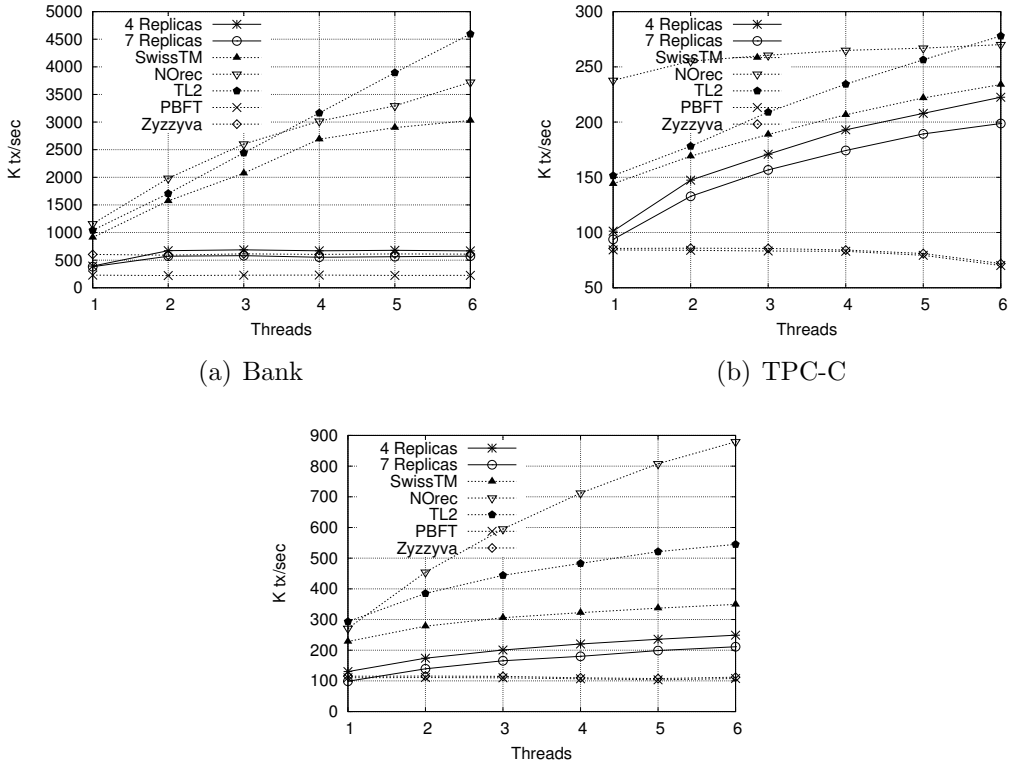
Figure 6.6(b) shows how the average time between the broadcast and final-delivery of a message ("Delay" in the plot) increases when the number of clients increases, following the same trend as that of throughput's slight degradation. Figure 6.6(b) reports also the average time between an optimistic and its respective final delivery. This time includes, on average, 76% of the total delay for ordering a message thus this time can be effectively exploited by replicas' concurrency control to speculate before the arrival of the final delivery.

Figure 6.6(c) evaluates the probability of mismatch between optimistic and final order. The plot shows an almost perfect match ($\approx 99\%$). This result is particularly important for the replica concurrency control, as it shows the effectiveness of processing transactions speculatively.

Concurrency Control. We evaluated SHIELD using three well-known benchmarks for transactional systems including Bank, TPC-C [31], and Vacation from the STAMP benchmark suite [21], and one micro-benchmark, the List data structure. These benchmarks have different transaction execution times: Bank transactions are very short; TPC-C involves more computation, resulting in longer execution time; Vacation represents a real-world workload by emulating a travel reservation system using an in-memory database; List operations pay the cost of traversing the data structure, increasing the read-set size and thus the execution time.

We experimented with 4 and 7 replicas, and varied the number of ObCC threads accordingly. Besides SHIELD's overall performance, we also measured the performance of PBFT and Zyzzyva with 4 replicas, TL2 [40], NOrec [34], and SwissTM [44]. PBFT and Zyzzyva run requests sequentially and increasing the number of application threads does not result in better performance. SHIELD's ObCC is close to SwissTM as it uses read/write locks and is Orec-based (i.e., for each shared object, a memory space is defined where object's meta-data are stored).

Figure 6.7(a) shows the Bank benchmark's throughput. SHIELD's performance with 4 and 7 replicas are very close to the network layer's throughput. This is because Bank's transactions are very short and the speculative processing that is done before issuing the final order allows the transaction execution to significantly overlap with the network ordering process. In this case, the performance degradation between SHIELD and the non fault-tolerant competitors is from 2.4× to 3×. This significant difference is largely due to the very short execution time of Bank transactions. In this setting, the impact of ordering the message before transaction execution on the transaction's latency is evident. Most real-world applications' transactions



(a) Bank

(b) TPC-C

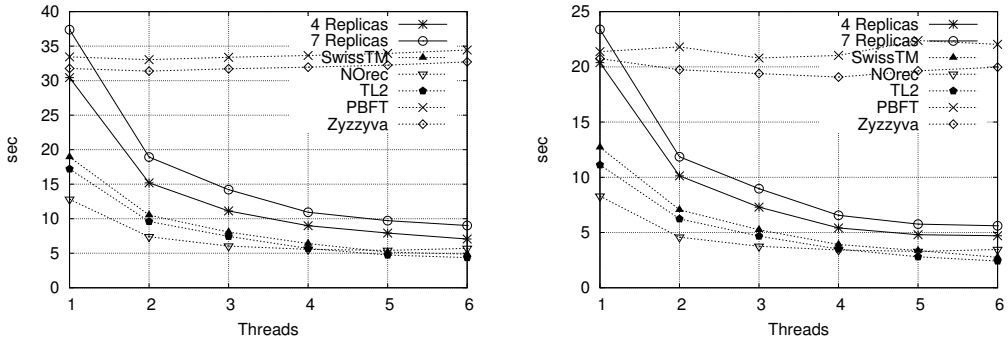
(c) List

Figure 6.7: Transactional throughput of SHIELD on Tilera.

are much longer and network throughput does not represent a bottleneck in that case.

In Figure 6.7(b) SHIELD's performance using TPC-C benchmark is plotted. The benchmark is configured using the standard percentage of transaction profile as suggested by the original specification, with 100 warehouses available in the system. In TPC-C, transactions are much longer than Bank's. Thus, the total throughput is lower. However, the gap between the non fault-tolerant protocols (i.e., SwissTM, TL2) and SHIELD is limited to 9% and 22.5%, respectively. NOrec's behavior on this benchmark is different from others as it does not use Orecs. But at a large number of cores, it is similar to TL2. The impact of the network layer's delay is annulled by standalone transaction execution. SHIELD is 1.14× and 1.19× faster on average than Zyzzyva and PBFT respectively. The benefits of concurrent execution of SHIELD are evident. At one thread, all have similar performance, then SHIELD performance increases as the number of threads increases.

Results with the List benchmark indicate the same trend as that of TPC-C for SHIELD and SwissTM. The performance of TL2 and NOrec is much better as they do not use write locks. We configured the list with 256 items. Each transaction selects an operation to execute (i.e., read, insert, and delete) using a uniform distribution. Here, the gap between SwissTM



(a) Vacation - High Contention (b) Vacation - Low Contention

Figure 6.8: Vacation execution time on Tilera.

and SHIELD is about 48%. SHIELD is 0.8× and 0.87× faster on average than Zyzzyva and PBFT respectively. Contention in list operations is high due to traversing all nodes up to the required one. Thus, the read-set is large, which increases the conflict rate when a node in the read-set is modified by another transaction. This limits the speculative execution's benefits.

Figure 6.8(a) shows results for the Vacation/STAMP benchmark with high contention configurations. Note that, in this experiment, we plot the execution time (in contrast to all previous benchmarks, which plot the throughput), thus, lower is better. Vacation represents a real-world application and has a long transaction execution time. It emulates an online travel reservation system, with several clients (application threads) performing three types of transactions (reservations, cancellations, and updates) that interact with an in-memory travel system database. Starting from 3 application threads, the differences between SHIELD and SwissTM, NOrec, and TL2 are 44%, 54%, and 58% respectively. SHIELD is 2× and 2.2× faster on average than Zyzzyva and PBFT respectively.

Figure 6.8(b) shows results for Vacation for low contention configurations. At low contention, the number of conflicting transaction decreases and more transactions can run concurrently. Thus, the execution time is shorter. In this figure, the differences between SHIELD and SwissTM, NOrec, and TL2 are 47%, 59%, and 68% respectively. SHIELD performance is better at higher contention levels compared to previous experiments. SHIELD is 1.9× and 2.2× faster on average than Zyzzyva and PBFT respectively.

Summarizing, the general trend observed in these results shows that SHIELD does not significantly degrade system performance with respect to standalone (i.e., non fault-tolerant) execution, especially in non-trivial benchmarks (e.g., TPC-C, Vacation), where this gap is very limited. SHIELD overhead is in the range from 9% to 60%. SHIELD concurrent execution allows it to get much better performance compared to non-concurrent BFT systems. SHIELD is 1.54× on average better than non-concurrent BFT systems.

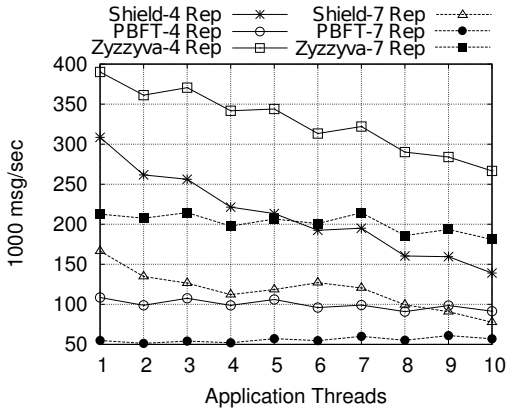


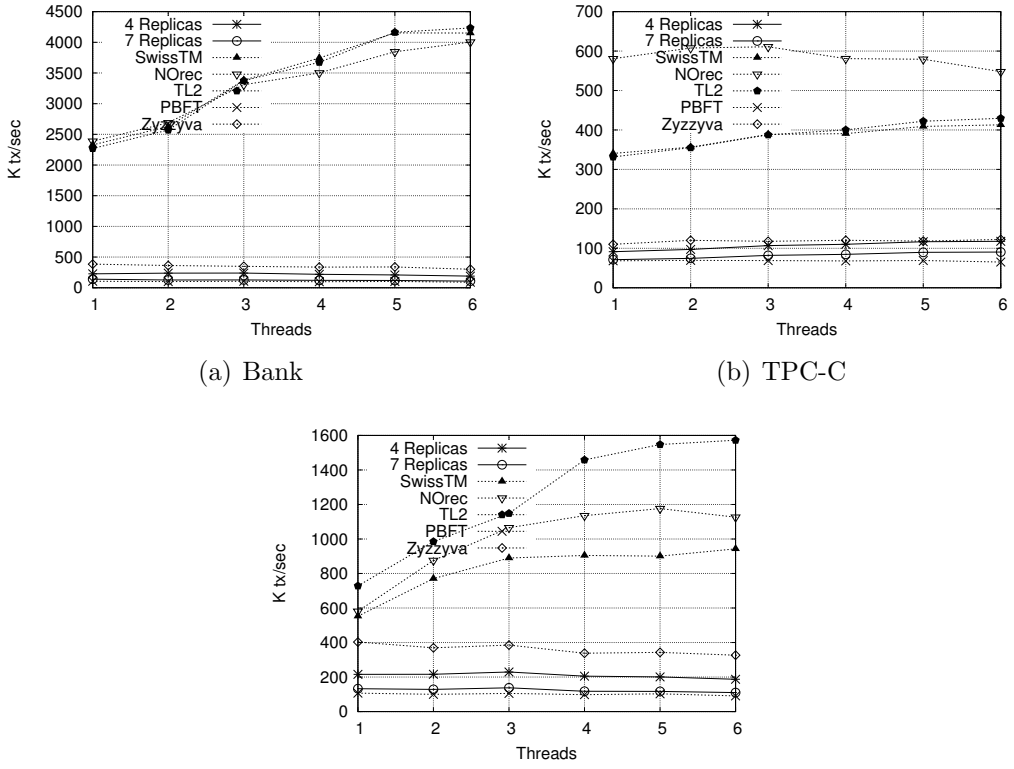
Figure 6.9: Network layer throughput on x86.

6.8.2 x86 Architecture

The framework presented in this chapter can be configured for working also in modern x86 multicore architectures. As a clock-service, the `rdtscp` instruction can be used [96] instead of the cycle counter register of Tilera. ObCC is implemented relying on RSTM. RSTM naively supports execution on modern x86 architectures. Regarding the network layer, we defined APIs for decoupling the functionality offered from the underlying implementation of basic primitives like send or receive. In a message passing system, sending information to a replica means traversing the hardware interconnection system. In shared memory architecture, we implemented it using reliable in-memory queues between replicas.

Figure 6.9 shows the network throughput on x86. SHIELD results on x86 architecture show low network throughput compared to Tilera (1× to 3× lower). Scalability on x86 is worse than Tilera and increasing the number of replicas degrades performance. SHIELD broadcast a client request to all replicas and other clients. This puts more pressure on the shared bus compared to Zyzzyva where client sends the request to the primary replica only. Then the primary replica broadcasts to other replicas. In shared-bus architecture, bus bandwidth is the major bottleneck and it is directly related to number of sent messages. Sending messages to different entities (as SHIELD does) does not reduce the pressure on the network as all messages end on the same shared bus. In Tilera, the network is a mesh network. Thus, sending the messages into two directions (replicas and clients) is done in parallel using two different links.

The limited network throughput on x86 affects all benchmarks results (Figure 6.10). The network layer is a bottleneck to ObCC. ObCC is idle most of the time waiting for a request from the network layer. Figure 6.11(a) shows Vacation benchmark results in high-contention settings. As Vacation transactions are large enough to overcome the network layer bottleneck, SHIELD overhead is in acceptable range and is about 1.3× on average. Figure 6.11(b) shows Vacation benchmark results in low-contention settings. It follows the same trend of high-



(a) Bank

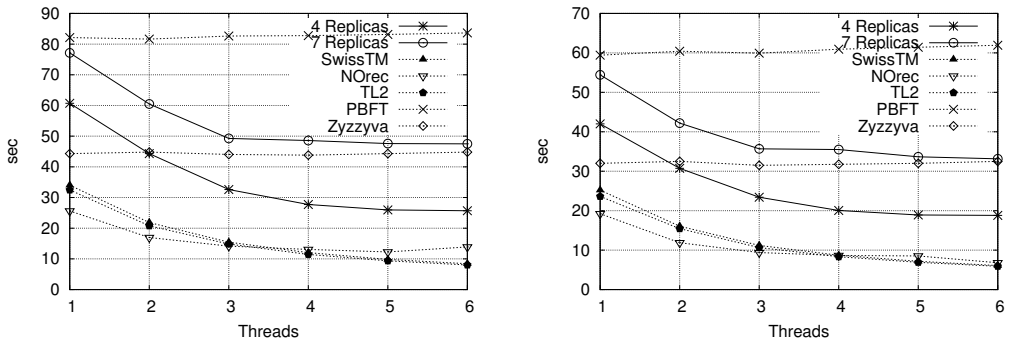
(b) TPC-C

(c) List

Figure 6.10: Transactional throughput of SHIELD on x86.

contention but with lower execution time.

It is clear from x86 results that using reliable shared memory queues adds a significant performance hit. A more optimized inter-process communication technique is required. We plan to address that as a future work.



(a) Vacation - High Contention (b) Vacation - Low Contention

Figure 6.11: Vacation execution time on x86.

Chapter 7

SoftX

7.1 Problem Statement

Multicore architectures are the current reality for improving parallelism, scalability and performance of applications. In particular, this is the case of transactional applications that are inherently parallel due to the abstraction of transaction. A single thread application performs transactions that are executed concurrently with others invoked by parallel application threads.

When those applications need to be resilient to faults, then a fault-tolerance protocol is needed for managing the concurrency among transactions, replicating objects on multiple sites or writing them on multiple storage systems. Both the solutions implicate additional costs for acquiring a distributed infrastructure but, beyond this, they have a negative impact on application performance because they introduce additional overhead such as network communication, distributed synchronization and interaction with stable storage, that can degrade overall performance even by orders of magnitude.

Faults can be roughly classified in transient and permanent. Transient faults can cause data corruption. They have many sources due to software bugs and hardware errors. In addition, some software bugs are difficult to detect and reproduce like race condition bugs, which only appear in some schedules under certain conditions. Soft-errors [16] belong to the category of hardware-related errors and they are very difficult to detect or expect/predict. Specifically, soft-errors may happen anytime during application execution. They are caused by (external) physical phenomena [11], e.g., cosmic particle strikes, electric noise, which cannot be directly managed or predicted by application designers or administrators. As a result, a soft error is silent, the hardware is not affected by interruption, but applications may crash or behave incorrectly.

This kind of faults are nowadays becoming a concrete problem to face with because of

the proliferation of multicore architectures. In fact, the trend of building smaller devices with increasing number of transistors on the same chip is allowing designers to assemble these powerful computing architectures. However, although the soft error rate of a single transistor has been almost stable over the last years, the error rate is now growing due to rapidly increasing core counts [11]. A soft error can cause a single bit in a CPU register to flip (i.e., residual charge inverting the state of a transistor). Most of the time, such an event is likely to be unnoticed by applications because they do not use that value (e.g., unused register). However, sometimes, the register can contain an instruction pointer or a memory pointer. In those cases, the application behavior can be unexpected. In SoftX we focus on soft-errors as a good representative of transient faults that are random, hard to detect, and can corrupt data.

Widely used replication solutions [98, 87, 108] are usually more suited for permanent failures, where the entire node goes down and it can stop or restart according to the adopted failure model, than transient faults. Byzantine behaviors are usually connected with transient faults because, when an application has a transient fault, it can arbitrarily misbehave. However, byzantine solutions (such as [22, 27]) are more general because they are designed for minimizing assumptions on execution's behavior and for malicious behavior in general. As a result, their impact on system's performance could be much higher than what is actually needed for solving the problem of transient faults that, indeed, can be seen as a small part of the bigger picture of byzantine fault-tolerance. As an example, our solution does not target any malicious application behavior or security hazard.

A famous example for data corruption effect in production systems is the major outage of Amazon S3 service[1]. In this incident, a single-bit flip corrupted a message and the effect propagated from one server to another. They managed to fix it by taking the system down, clearing the system state, and restoring the system back. This part of the system was not protected against data corruption errors. The downtime was over seven hours and initiated by a single transient fault.

We tackle the challenge of solving transient faults by building a low-intrusive software infrastructure that guarantees reasonable good performance, even though worse than the original, non resilient to transient faults, version of the application, but better than typical replication-based solutions. Our proposed solution guarantees both safety and liveness. The targeted system should always return correct results without any downtime.

In this chapter we present SoftX, our framework for making a transactional application resilient to transient faults. SoftX's main goal is preventing data corruption caused by a transient fault from propagating in main memory, without a significant impact on application's performance (i.e., 10× worse). This goal is reached by SoftX leveraging on speculative processing [84, 83, 56]. When a transaction is activated, SoftX captures the transaction and forks a number of threads that replicate the execution of the transaction in parallel, speculatively on different cores of the system. We assume an underlying hardware

[1] http://status.aws.amazon.com/s3-20080720.html

architecture equipped with an enough core count such that the speculative execution does not hamper the application's progress. The outcome of each thread is then compared with the others using dedicated *committer threads*. If it corresponds to the majority outcome, then the transaction can commit and the control is returned to the application. This approach masks transient faults happened during the execution on some core with the speculative execution on other cores.

A straightforward solution to transient faults problem that supports both safety and liveness is state-machine replication (SMR) [86]. But SMR approach has several sources of overhead. Request must be wrapped in a network message, totally ordered, executed in-order, and finally uses voting to compare results and accepts the majority result. SMR requires also a high communication bandwidth. In order to map SMR approach which is designed for distributed system to a centralized multicore machine, computational resources, as well as memory, should be partitioned into replicas [78].

In SoftX, computational resources and memory are not partitioned and no ordering protocol is required before to process transactions (e.g., [56, 84]) or after for validating transactions (e.g., [88]). Moreover, hardware cores are reused by other transactions after a commit event; they are not preassigned to certain application threads such that the overall system's load is more balanced.

As a practical design choice, SoftX follows the same abstraction as Software Transactional Memory (STM) [100] where transactions are simply enclosed in atomic blocks and the burden of managing concurrency and guaranteeing isolation and atomicity is on the STM protocol. This decision does not represent a limitation for our framework. It can be easily integrated with other transaction abstractions. We decided to use the STM's because this framework is appealing and transparent from the programmer standpoint.

In order to assess the performance of SoftX, we implemented it on top of the Rochester STM (RSTM) library [74] a famous framework for developing and running STM algorithms. As competitors, we selected: a well known STM system such as NOrec [34] as representative of protocols that are not resilient to transient faults; a modified version, for being resilient to transient faults, of the classical SMR approach [78], where transactions are ordered before their execution; and PBFT [22] as a representative of Byzantine fault-tolerant systems. This way we attempt to provide an overview about SoftX's cost and its performance against fault-tolerant, as well as non fault tolerant approaches.

Results on a 48-cores AMD server, using well-known transactional benchmark such as List, Bank and TPC-C [31], reveal that SoftX has an overhead with respect to NOrec that is limited to 1.1× except for Bank which is characterized by very small transactions. Thus, synchronization overhead is more evident. Real applications usually have larger transactions. Compared to other fault-tolerant approaches, SoftX gains against the SMR approach by an average of 1.6× and gains against PBFT by an average of 4.5×.

Hardware message-passing communication is an important emerging trend in multicore ar-

chitectures. Given that SoftX targets multicore architectures, we believe it is important to study our proposal on both: current shared-bus architectures, as well as emerging message-passing architectures. In order to reach this goal, we ported SoftX to the Tilera TILE-Gx [103] board, a message-passing architecture. Message-passing architectures do not suffer from the limited bandwidth of the shared-bus architectures. Results show that SoftX still outperforms replication-based approaches. SoftX's overhead compared to non-fault-tolerant approach (i.e., NOrec) is reduced to about 40% on average. Replication-based approaches also take advantage of the hardware message-passing network. Their results are now closer to SoftX.

SoftX is optimized for transactional applications because we believe they fulfill a key role in the space of transient faults especially because their nature does not allow any kind of unexpected and unmanageable data corruption.

To the best of our knowledge, SoftX is the first attempt to reduce the scope of data corruption to transactional processing without the overhead of classical replication-based solutions. We propose a solution that reduces overhead and increases resources utilization while keeping safety and liveness.

SoftX's full implementation, with sources and test-scripts, is available at `www.hyflow.org/downloads/softx.zip`.

7.2 Assumptions and Applicability

SoftX does not replicate data in memory therefore, when a hardware error that affects memory (e.g., soft errors) occurs, it relies on the detection and correction techniques, like ECC (Error-Correcting Code), to avoid propagation of such a error into registers. SoftX protects data by allowing only committer threads to write in memory. Speculative threads (groups) have read-only access to shared data. Writes are buffered and sent to committer threads to decide if they can be committed safely or a conflict/error happened. Moreover, each committer thread keeps an undo log of its writes to shared data so that writes can be validated and if a corruption is detected, memory can be restored using the undo log.

SoftX works on a single machine, thus it cannot tolerate a crash or a permanent hardware failure of the machine. Asynchronous checkpointing to stable storage can be used to tolerate such failures. Similar to other replication-based techniques, SoftX cannot tolerate deterministic software bugs or incorrect configurations. A deterministic bug will occur on all replicas and generate the same results. Thus, voting will not detect it. This problem could be solved by using diversity[2](e.g., [27, 114]). SoftX can still detect a wide range of transient faults that corrupt data during transaction execution. These include hardware

[2]Each replica produces the same results via different approaches (e.g., different versions of the same application.)

transient faults and random software bugs which are difficult to find/reproduce (e.g., race condition and other concurrency bugs).

7.3 Design and Implementation

SoftX is a software transactional memory system that provides programmers the traditional TM constructs (e.g., atomic block, read, write) for building transactional applications. At the same time, SoftX's architecture ensures resiliency to transient faults. In SoftX, an application thread that starts a transaction, forks a group of threads (on different cores). Each thread in the group executes the same transaction speculatively and independently from others. At the beginning of a transaction, they synchronize their starting points such that all will observe the same initial state.

At commit time, instead of proceeding with the commit operations, the first thread completing its execution contacts a group of dedicated threads, called *committer threads*, for deciding whether the transaction can be committed depending on the outcome of each speculative execution, or must be aborted. The transaction is committed only when the threads' validation succeeds and majority of the speculative executions reach the same stage (i.e., no conflicts and no unmasked faults); otherwise, an abort signal is sent back to the threads.

Committer threads as a whole can be considered as a voter component. Instead of allowing the speculative threads to synchronize with each other for finalizing the commit, relying on the committer threads for the commit operation offloads work from the speculative threads. In addition, having dedicated committer threads reduces cache misses and invalidations (as the overhead of CAS operations is avoided), improving performance, as also shown in [71]. Algorithm 1 and 2 show SoftX's pseudo code[3].

SoftX's design is inspired by NOrec [34]. This choice was made because NOrec uses a single global lock, thus the synchronization between the speculative threads and the committer threads is simple and efficient. Speculative threads proceed similar to NOrec until commit time. Upon reaching the commit stage, each thread alerts committer threads through setting a shared flag for accomplishing the commit procedure (i.e., validation and memory write-back if the transaction is valid). Subsequently, speculative threads wait until the notification of committer threads, and either commit or restart. During the execution, speculative threads log their read and written objects into private memory areas called read-set and write-set, respectively (line 1.17, 1.20).

Speculative threads in charge of executing the same transaction are logically grouped. A group's size can be configured according to the degree of resiliency desired i.e., number of transient faults that can potentially occur in parallel. SoftX decides to commit a transaction if the majority of the speculative executions' outcome coincides. For example, using

[3]For the identification of lines in the pseudo-codes, we use the notation Algorithm.Line.

Algorithm 1 Transaction speculative execution

```
1: procedure txBegin(groupId)
2:     myIndex ← IncAndGet(groupIndex[groupId])
3:     if myIndex = 1 then
4:         SendSignal(No_Commit)
5:     end if
6:     txStartTime ← timestamp
7:     if myIndex = GROUP_SIZE then
8:         SendSignal(Proceed_Commit)
9:         groupIndex[groupId] ← 0
10:    end if
11: end procedure
12: function txRead(addr, groupId)
13:    if WriteSet.contains(addr) then return WriteSet.get(addr)
14:    end if
15:    val ← Memory[addr]
16:    while txStartTime ≠ timestamp do
17:        txStartTime ← timestamp
18:        if ¬Validate() then
19:            AbortGroup(groupId)
20:        end if
21:        val ← Memory[addr]
22:    end while
23:    ReadSet.put(addr, val)
24:    return val
25: end function
26: procedure txWrite(addr, val, groupId)
27:    WriteSet.put(addr, val)
28: end procedure
29: procedure txCommit(groupId)
30:    RequestCommit(myIndex, groupId)
31:    loop
32:        WaitForResponse
33:    end loop
34:    if GetResponse() = ABORT then
35:        Restart()
36:    else
37:        FinishGroup()
38:    end if
39: end procedure
40: function Validate
41:    if CommittersInWriteBack() then
42:        Wait()
43:    end if
44:    for all entry in ReadSet do
45:        if entry.val ≠ Memory[entry.addr] then                ▷ Value-based
46:            return false
47:        end if
48:    end for
49:    return true
50: end function
```

three speculative threads per group masks a single transient fault without re-execution. If more faults occurred, the speculative execution is restarted to mask these faults.

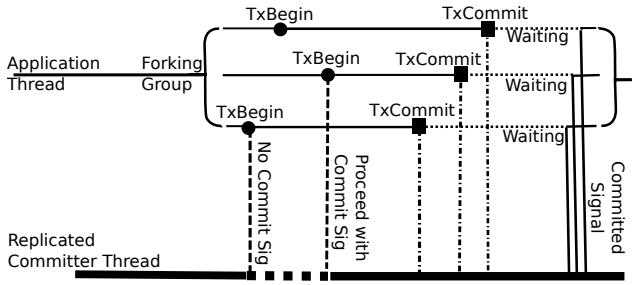


Figure 7.1: Processing transaction with SoftX.

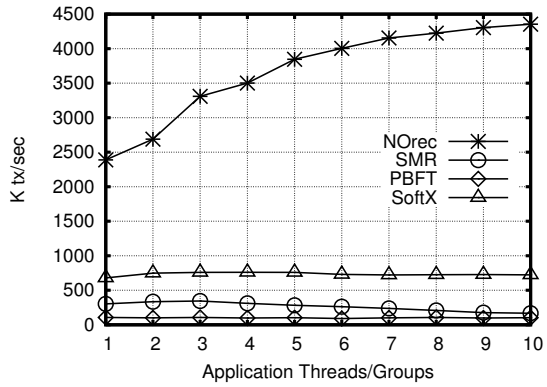
Figure 7.1 shows the communication between the speculative threads of a group and the committer threads (or "committers"). In this case, a group has three threads. Each thread executes the same sequence of operations independently from other threads. At the beginning of a transaction, all threads of the group must observe the same initial state (i.e., same starting timestamp). Thus, the first thread, before beginning a new transaction, sends a signal `No_Commit` to committers to pause any commit procedure (line 1.4). This way, the global timestamp cannot change until all threads in the group successfully start a new transaction (line 1.7). Committers resume the processing of commit requests when the last thread in the group sends the signal, `Proceed_with_Commit` (line 2.20).

In order to detect possible invalidations during transaction processing, any member of a group sends an abort signal to other group members to abort as soon as a previously read object becomes invalid (line 1.15). For this reason, speculative threads validate their read-set after each read operation (line 1.12 - 1.16). This way, all group members restart their execution from the same initial state after an abort.

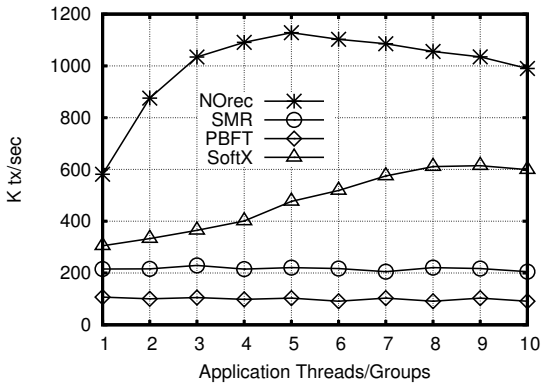
At commit time, each thread in the group sends a commit request to the committer (line 1.22) and waits for the decision (line 1.24). The committers consist of replicated threads such that they can mask transient faults during the commit procedure. As a result, they act as a "voter." They wait for a commit request from each speculative thread and starts the commit procedure when requests from all group members are received (line 2.4). Then, committers need to agree on which group to commit together (line 2.5). Then, each committer independently starts to validate the read-set generated by the speculative execution of a thread in the group (line 2.9). If the majority of speculative executions is valid (line 2.11), committers compare group members' write-set to ensure that the majority is identical (line 2.14). At this stage, each committer thread has reached a decision on whether to commit or abort its speculative execution, independently. After that, another coordination among committer threads is required to compare their final decisions (line 2.12, 2.17, 2.27). If a majority is reached, one committer thread executes the agreed decision (commit or abort). During write-back operation, an undo log is used to store current values in the memory be-

Algorithm 2 Committer threads

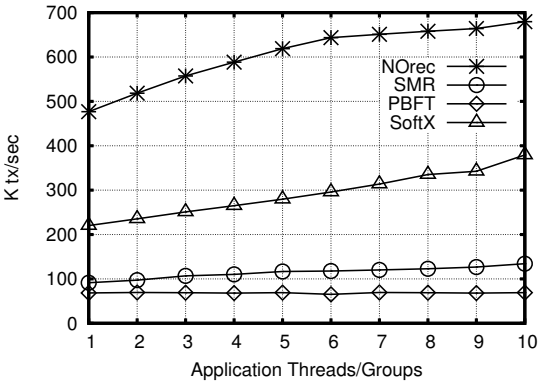
```
1: loop
2:     for group ← 1 to activeGroupsNum do
3:         doneMembers ← GetDoneMembersNum(group)
4:         if doneMembers = GROUP_SIZE then
5:             MySelectedGroup(myIndex, group)
6:                                          ▷ Agree on selected group between committer threads
7:             abortNum ← 0
8:             for i ← 1 to GROUP_SIZE do
9:                 if ¬Validate(GetContext(group, i)) then
10:                    abortNum ← abortNum + 1
11:                end if
12:            end for
13:            if abortNum > GROUP_SIZE/2 then
14:                SendResponse(ABORT, group)
15:            else
16:                majority ← Vote(group)
17:                                          ▷ Compare group's write-sets
18:                if ¬ majority then
19:                    SendResponse(ABORT, group)
20:                else if myIndex = CUR_COMMITTER then
21:                    if NO_COMMIT_FLAG then
22:                        WaitForProceedCommitSig(timeout)
23:                    end if
24:                    DoWriteBack(majority, group)
25:                    Notify(WRITEBACK_DONE)
26:                else
27:                    WaitForWriteBackSig(timeout)
28:                    if ¬ValidateWriteback() then
29:                        FixCommitters()
30:                    end if
31:                    SendResponse(COMMIT, group)
32:                end if
33:            end if
34:        else if doneMembers > 0 then
35:            StartGroupTimeout(group)
36:        end if
37:    end for
38: end loop
```



(a) Bank



(b) List



(c) TPC-C

Figure 7.2: Transactional throughput of SoftX on x86.

fore overwriting them. Then, in order to tolerate transient faults during this stage, the other committer threads confirm that the values written to memory match the original write-set's values (2.24, 2.25). Otherwise, the undo log is used to restore original memory state (line 2.26).

A transient fault can occur in an application thread, causing it to produce incorrect results or stalling it indefinitely (i.e., it becomes a zombie thread). On the one hand, incorrect results are detected when write-sets are compared and read-sets are validated by the committers. On the other hand, a zombie thread is detected by a timeout while waiting for the commit request from all threads in a group (line 2.29). Speculative threads do not write to shared memory; only the committers can change it. Memory protection mechanism is used to enforce read-only access for speculative threads. For making the committer threads resilient to transient faults, they execute the same steps in an independent manner. Then they compare their outcomes during the coordination phases. Even in this case, a timeout mechanism is used to detect possible transient faults (e.g., line 2.24).

SoftX requires that speculative threads in a group have identical inputs to produce identical

behavior. In other words, their actions must be deterministic [84, 56]. To do that, we scope out any form of non-determinism that could result in possible false transient faults. For example, if a transaction calls `random` function, we extract the generation of the random number from the transaction's body, and make this number available for all speculative threads.

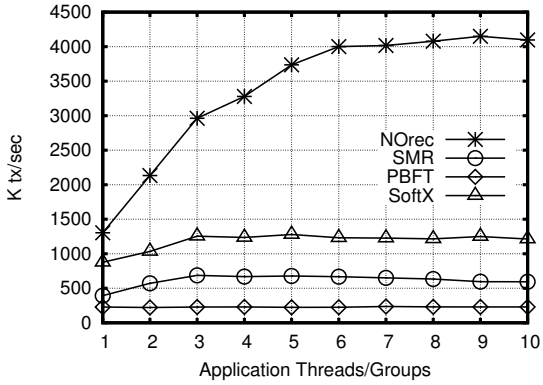
Another issue that SoftX has to cope with is the so called *timestamp extension* [34]. This consists of extending the original timestamp of a transaction when it observes that its read-set is still valid but the timestamp has been increased by other commits. This way, subsequent read operations could save the overhead of re-validation if there is no change in the global timestamp (i.e., no transactions committed in between those two read operations). In SoftX, speculative threads act independently, thus some threads could extend the timestamp rather than others. Extending the timestamp synchronously significantly impacts performance. Therefore, we made the decision of extending the timestamp asynchronously. In case some speculative thread arrives at its commit phase with a timestamp different from others and different results, we consider it as faulty execution and we abort the entire group. We adopt this "conservative" behavior because the committers cannot distinguish between the case in which the timestamp has been corrupted by a transient fault from the case in which the speculative thread missed extending its timestamp. (We also explored the solution removing the timestamp extension feature. However, the resulting performance was worse than keeping the timestamp extension and aborting the entire group.)

Another important aspect is the methodology for allocating memory slots within a transaction. When a new memory slot is allocated, each speculative thread acquires a different memory location because its area differs from areas allocated by other threads, even though those areas correspond to the same logical object. This results in different values (i.e., memory addresses) in the group's write-sets, and therefore, all members will never have identical write-sets. To overcome this problem and make the validation procedure feasible, we mark each new memory location with a logical reference, and we compare the value of the location instead of the address. Then we verify that the memory address is correct and if it points to the same value. On the contrary, memory deallocation is managed by committers, instead of speculative threads.

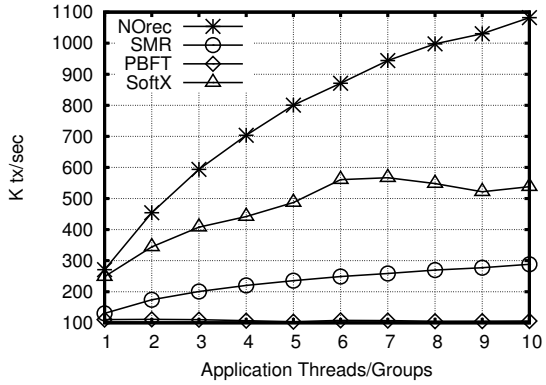
7.4 Experimental Results

SoftX is implemented on two architectures: a shared-bus architecture, represented by common x86 compliant multicore systems, and a message-passing architecture, represented by the Tilera TILE-Gx board [103].

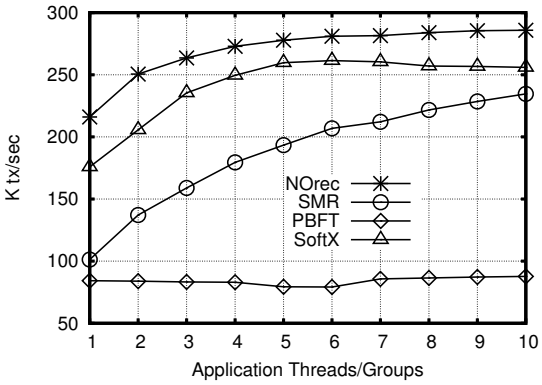
Regarding the former, we conducted our experiments on a 48-core machine, which has four AMD Opteron^{TM} Processors (6164 HE), each with 12 cores running at 1700 MHz, and 16 GB of memory. The machine runs Ubuntu Linux Server 12 LTS 64-bit. Regarding the latter,



(a) Bank



(b) List



(c) TPC-C

Figure 7.3: Transactional throughput of SoftX on Tilera.

we used a 36-core board of the Tilera TILE-Gx family. This hardware is commonly used as an accelerator or an intelligent network card. Each core is a full-featured 64-bit processor (1.0 GHz), with two levels of caches, 8 GB DDR3 memory, and has a non-blocking mesh that connects cores to the Tilera 2D-interconnection system.

We implemented our solution using C++ in Rochester STM (RSTM) library [74]. RSTM uses platform-specific assembly instructions. Therefore, we ported its original implementation to support Tilera TILE-Gx family.

In order to assess the performance of SoftX, we used three competitors. The first is the original, non transient faults tolerant, version of NOrec [34]; the second is an SMR approach [78], typically used in transactional replication; and the third is PBFT [22] which represents a byzantine fault-tolerant system.

We implemented an SMR system in a centralized setting inspired by [78]. In this system, each client (i.e., application thread) reserves a core and sends its requests to replicas (or nodes) via shared memory queues or hardware message passing channels. A replica represents

a partition of resources in terms of memory and cores. Specifically, each reserves one core for network communications and a variable number of cores to execute transactional requests in parallel. According to the SMR paradigm, clients send their transactional requests to an ordering layer, which is responsible for ordering those requests and delivering them to replicas. With the purpose of keeping consistent the states on all replicas, each must execute transactional requests in the order defined by the ordering layer. We implemented this order by tagging all client requests with the value of a single shared counter using the atomic fetch-and-increment instruction. For respecting the order at the level of the concurrency control, usually, in SMR systems, each replica executes requests sequentially. We overcome this lack, developing a replica concurrency control that supports parallel execution but, at the same time, it enforces the requests' order. Each client also acts as a voter that collect replies from replicas. We added the voter because also this approach has been designed for avoiding transient fault. We only assume that a transient fault cannot happen when the atomic fetch-and-increment instruction is called. PBFT is implemented similarly but it executes transactions sequentially and has a higher overhead in the ordering layer according to PBFT original design.

We tested SoftX on three well-known benchmarks in transaction processing such as Bank, List and TPC-C [31]. Bank mimics operations of a monetary applications. List is a micro benchmark that represents the list data structure. List operations pay the cost of traversing the data structure, increasing the read-set size and thus the execution time. TPC-C is the classical representative of an online transaction processing system based on warehouses and orders that clients perform on shared items. These benchmarks expose different transaction execution time: in Bank transactions are very short; in List transactions are longer and conflict rate is higher; in TPC-C transactions are the longest because they involve more computation. They differ also in terms of transaction profiles. Bank has one profile for writing bank accounts (i.e., the write transaction) and one for retrieving informations from bank accounts (i.e., the read-only transaction). List has three operations representing the list data structure insert, delete and contains (read-only) operations. TPC-C has five transaction classes: three of them are write transactions while two are read-only transactions. In this evaluation study, we configured Bank for generating 65% of write transactions, List with uniform distribution between the three operations, and TPC-C with its default configuration. All data points plotted are the average of five repeated samples.

As for the STM configuration, in the SMR approach we fixed the number of replicas to three because it matches the size that we used in SoftX as groups' size. Moreover, having only three replicas does not reduce significantly the resource available per replica. For PBFT, we used four replica which is the minimum number of replica required. BFT systems requires $3f + 1$ replicas to tolerate f faults. In the following plots we measure the transactional throughput while we increase the number of groups serving transactions. As a result, when we report the results on 10 groups, it means that the system is running 30 threads. Finally, the number of committer threads is always fixed at three.

7.4.1 Shared-bus architecture

Figure 7.2 shows the results on x86 shared-bus architecture. In all benchmarks, the shared-bus represents a bottleneck for replication-based systems. SoftX minimal synchronization improved its performance compared to replication-based systems. It also reduced the overhead of SoftX compared to non faults tolerant system.

Figure 7.2(a) shows the results with Bank benchmark. Performance of NOrec is 4× better than SoftX. This large overhead is due to Bank's very small transactions, which magnify the synchronization overhead of SoftX. In addition, NOrec does not have any mechanism for being resilient to transient faults. However, when compare to the SMR approach, SoftX does not suffer from the overhead of global ordering and in-order processing, thus outperforming it by 1.96× on average. Compared to PBFT, SoftX outperforms PBFT by 6.3× on average. PBFT has higher overhead to reach consensus and also executes requests sequentially.

Figure 7.2(b) shows the results with the List application. As the length of list's transactions is longer and the contention is higher, the additional overhead due to multiple synchronization points, needed for tolerating possible transient faults, is less evident than in Bank. Thus, SoftX overhead is reduced to 1.15× on average compared to NOrec. When the contention in the system increases due to multiple threads working, performance becomes more comparable (65% overhead) because SoftX exploits the presence of committer threads, which reduces also hardware contention and increase system's scalability. As a general trend, SoftX performs better than SMR. The reason is mostly related to the in-order processing and also the final voting that clients perform at the end of each transaction. SoftX optimizes this process: speculative threads process in parallel to committer threads. This way the transaction execution in the system is overlapped with the commit phase, thus reducing the stalls that happen on the SMR approach. SoftX outperforms SMR by 1.23× on average. SoftX outperforms PBFT by 3.89× on average.

Figure 7.2(c) shows the results with TPC-C benchmark. Here the benchmark is complex and transactions are the longest compared to List and Bank. In terms of contention, TPC-C has lower contention than List as we use 100 warehouses whereas in List, all transactions traverse the same nodes of the list, which increase the conflicts. NOrec outperforms SoftX by 1.1× on average. SoftX gains up to 1.54× compared to SMR and 3.27× compared to PBFT.

Summarizing, SoftX overhead is acceptable for application that has high contention (e.g., List) and/or long transactions (e.g., TPC-C).

7.4.2 Message-passing architecture

Figure 7.3 shows the results on the Tilera message-passing architecture. With the more communication bandwidth, all fault-tolerant approaches improve their performance.

Figure 7.3(a) shows the results with Bank benchmark. Performance of NOrec is now 1.8× better than SoftX compared to 4× in shared-bus results. The message-passing network increased the communication bandwidth between different cores and now synchronization overhead is distributed over multiple links. For the same reason, replication-based approaches overhead is decreased as it is designed for exploiting networking capabilities. SoftX outperforms SMR by 0.94× on average. And SoftX outperforms PBFT by 4.18× on average.

Figure 7.3(b) shows the results with the List benchmark. The overhead of SoftX is farther decreased given List's high contention nature and message-passing's higher bandwidth. SoftX overhead is reduced to 61% on average compared to NOrec. SoftX outperforms SMR by 1.02× on average. SoftX outperforms PBFT by 3.37× on average.

Figure 7.3(c) shows the results with TPC-C benchmark. Given the TPC-C's long transactions, SoftX overhead is minimal to 12% only on average compared to NOrec. SoftX is 33% better compared to SMR and 1.88× better compared to PBFT.

In summary, having a message-passing architecture, synchronization and communication overhead is significantly reduced. Now, the network bandwidth does not represent a bottleneck in most of the benchmarks. The benefits of SoftX's and SMR's concurrent execution are evident compared to PBFT, which does not scale with increased number of threads. SoftX still outperforms replication-based approaches.

SoftX performs better than replication-based approaches since it requires less data transfer between system components. State-machine replication is designed for distributed systems which have a network. In x86 centralized system, using memory queues as a network saturates the shared bus and reduces the performance, significantly. Moving the implementation to a message-passing architectures reduces replication-based systems' overhead however, even in this case, SoftX outperforms the other competitors.

Chapter 8

Conclusions

In this dissertation proposal, we made several contributions aimed at alleviating multi-core challenges in terms of performance and dependability. We exploited HTM as one of the best candidates for improving multi-core performance and easing parallel programming. Our target is to overcome most of best-efforts HTM limitations. We identified three major limitations: resource limitations; lack of an advanced contention manager; and lack of advanced nesting support. We addressed the resource limitations problem in PART-HTM. We addressed the lack of an advanced contention manager by an HTM-aware scheduler (OCTONAUTS). And we will address advanced nesting support in our post-preliminary work. Regarding dependability, we targeted transactional systems dependability where safety and liveness are crucial. Transactional systems also represent a large section of critical industrial applications. SHIELD addressed the problem using state-machine replication paradigm exploiting the fast message-passing hardware. SOFTX addressed the problem using redundant execution without splitting computational resources or ordering transactions. SOFTX lightweight synchronization make it suitable for both shared-bus based multi-core architectures and message-passing based ones.

We presented PART-HTM, a hybrid TM, which aims at committing those HTM transactions that cannot be fully executed as HTM due to space and/or time limitation. The core idea of PART-HTM is splitting hardware transactions in multiple transactions and run them in hardware with a minimal instrumentation. PART-HTM's performance is appealing, in our evaluation it is the best in favorable workloads, and it is always the closest to the best competitor in other workloads. PART-HTM represents one of the first efforts in solving the resources limitation problem of best-efforts HTM.

OCTONAUTS represents the first HTM-aware scheduler, to the best of our knowledge. It depends on a priori knowledge of transactions working-set. Using that knowledge, it prevents the activation of conflicting transactions simultaneously. Being HTM-aware, it supports concurrent execution of HTM and STM transaction with minimal added overhead. OCTONAUTS is also adaptive, based on the transaction characteristics (i.e., data size, duration,

irrevocable calls) it selects the best path among HTM, STM, or global locking. OCTONAUTS results shows performance improvement at high contention levels which confirms the need for an advanced contention manager for HTM systems.

SHIELD confirms that the state-machine replication paradigm is an effective solution for making transactional systems resilient to data corruption faults, especially in emerging message passing-based multi-core infrastructures. The cost of total-ordering transactions can be effectively mitigated by overlapping speculative transaction processing with ordering. In addition, when the transaction execution time is not minimal, the ordering cost is mitigated by local transaction execution, resulting in performance comparable with non fault-tolerant system, but with the advantage of making the application resilient to data corruption faults.

SHIELD is completely transparent to the underlying hardware: no instrumentation or wrapping of specific assembly instructions is needed. Also, the approach is independent of the underlying software platform (i.e., OS or virtual machine), as well as modular – it can be easily plugged into an existing platform, making it resilient to data corruption faults.

SOFTX confirms that it is possible providing a concurrency control protocol that makes transactional applications resilient to transient faults, without giving up high performance. We leveraged the huge amount of computational resources available in recent multi-core architectures, as well as a protocol that limits the overhead by combining speculative execution and dedicated committer threads. Our results show lower overhead compared to replication-based approaches even with an optimized state-machine replication protocol. SOFTX is also suitable for current shared-bus architectures, as well as the emerging message passing architectures.

8.1 Proposed Post-Prelim Work

8.1.1 HTM Advanced Online Contention Manager

Based on OCTONAUTS results, a contention manager (CM) for HTM is required to solve HTM high contention problem. OCTONAUTS gained more performance using a priori knowledge of transactions working-set. Given that a transaction static information is an overestimating of all potential accessed objects, we expect to gain more performance by using accurate online information that is collected at encounter time. Accurate online information allows more concurrency as it is a subset of the static analysis working-set.

We are working on a solution to the challenge of sharing online information without introducing more aborts due to HTM conflicts. Our potential solution is based on having a circular buffer meta data for each object. Where a transaction reserve a location for adding its own access information about the object. A transaction writes to the reserved location once only. Thus, it will not conflict with other transaction that will read that information

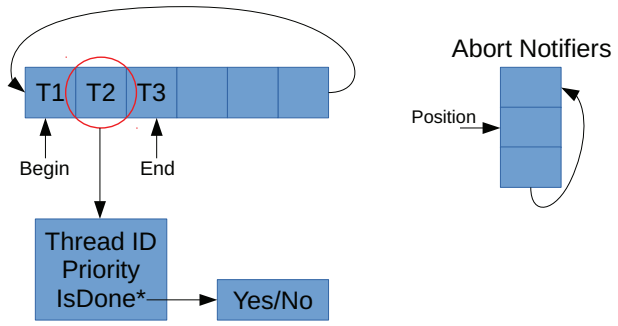


Figure 8.1: The proposed object's circular buffer meta data and abort notifiers.

later. This design gives a time window for each object meta data where it is not changed. The time window is the duration of time until the circular buffer rolls over.

Figure 8.1 shows a sketch of the object's circular buffer meta data. Each entry includes the thread id, its priority, and a pointer to its status (i.e., isDone). The isDone pointer idea is to allow the slot owner thread to notify other thread of its completion without aborting other readers. For example, when a transaction conflicts with another transaction in the object meta data and it has a lower priority, instead of immediately aborting itself. It first checks the isDone pointer value. Other higher priority threads will not read that pointer and will not be affected when it is updated. Low priority thread gets a better chance of living when it reads the pointer in case the thread is finished. Otherwise, it should abort itself anyway.

In order to allow a thread to abort another one, we use abort notifiers. Each transaction reads its abort notifier at the beginning, which adds the notifier to its read-set. Other transactions just need to write to a thread abort notifier in order to abort it. With each new transaction, a new abort notifier is used to nullify the possibility of an old transaction aborting a fresh transaction on the same thread.

8.1.2 HTM Advanced Nesting Support

HTM advanced nesting support can increase HTM concurrency level and performance. Current Intel HTM implementation supports only flat nesting. We plan to extend HTM nesting support by including closed and open nesting in order to alleviate HTMs flat nesting limitations. Closed nesting improves the performance by increasing the concurrency level [80] and providing a lightweight partial abort mechanism. In fact, with closed nesting when a conflict happens during the execution of a child transaction, only that transaction is aborted and re-

tried, rather than the entire transaction. Open nesting improves the concurrency further [80] because children transactions are allowed to commit their changes to the main memory before the commit of the parent transaction. When the parent transaction is aborted, an undo action is required for compensating children transactions changes. Open nesting removes children transactions read-set and write-set from the parent, which reduces conflicts chances.

Our plan is to extend PART-HTM to support nesting. PART-HTM supports committing parts of the original HTM transaction before committing the original transaction. In addition, it is possible to retry a partition without retrying the entire transaction. We need to create logical partitions groups that represent each child transaction. We need also to handle nesting semantics on each group of partitions. Each group should hold its own read-set and write-set and only share it with the parent transaction according to the nesting paradigm. For example, in closed nesting, a child transaction can access the parent transaction write-set but it should also keep its own write-set private until a successful commit. In open nesting, a child transaction will commit to the main memory and will not merge its write-set with the parent transaction. Moreover, a separate undo log is needed to undo committed children transaction in case the parent aborts.

8.1.3 Hybrid TM Framework

We plan to merge all our HTM enhancements and build a comprehensive framework for Hybrid TM algorithms. Hybrid algorithms is gaining more traction as it promises better performance than falling back to global locking. In addition, hybrid TM algorithms shares many building blocks. Our proposed framework makes embedding and testing new hybrid TM algorithms easy. It aims also at helping the research community in developing more hybrid TM algorithms, easily comparing them, and adaptively selecting the best one for a certain situation.

In order to do that, our proposed framework will support multiple HTM-STM communication primitives as well as giving the programmer the ability to define a new HTM-STM communication primitive. Each HTM-STM communication primitive defines how HTM is instrumented. For HTM transactions, the framework provides Pre and Post commit handlers, global locking with eager or lazy subscription, and reduced hardware transactions support. The framework supports adaptive fallback paths and/or multiple consecutive fallback paths. For STM transactions, the framework provides the following handlers: transaction begin; transactional read; transactional write; transaction abort; and transaction commit. Finally, the framework provides statistics information about transactions (e.g., committed in HTM, committed in STM, HTM abort reasons, percentage of each HTM abort reason).

Bibliography

[1] Philip A. ‘, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[2] Yehuda Afek, Amir Levy, and Adam Morrison. Software-improved hardware lock elision. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC ’14, pages 212–221, New York, NY, USA, 2014. ACM.

[3] Yehuda Afek, Alexander Matveev, and Nir Shavit. Reduced hardware lock elision. In *6th Workshop on the Theory of Transactional Memory*, WTTM ’14, 2014.

[4] Dan Alistarh, Patrick Eugster, Maurice Herlihy, Alexander Matveev, and Nir Shavit. Stacktrack: An automated transactional approach to concurrent memory reclamation. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys ’14, pages 25:1–25:14, New York, NY, USA, 2014. ACM.

[5] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS ’67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[6] C.S. Ananian, K. Asanovic, B.C. Kuszmaul, C.E. Leiserson, and S. Lie. Unbounded transactional memory. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 316 – 327, feb. 2005.

[7] C.S. Ananian, K. Asanovic, B.C. Kuszmaul, C.E. Leiserson, and S. Lie. Unbounded transactional memory. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 316–327, Feb 2005.

[8] Mohammad Ansari, Mikel Lujn, Christos Kotselidis, Kim Jarvis, Chris Kirkham, and Ian Watson. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In Andr Seznec, Joel Emer, Michael OBoyle, Margaret Martonosi, and Theo Ungerer, editors, *High Performance Embedded Architectures and Compilers*, volume 5409 of *Lecture Notes in Computer Science*, pages 4–18. Springer Berlin Heidelberg, 2009.

[9] Hagit Attiya and Alessia Milani. Transactional scheduling for read-dominated workloads. *Journal of Parallel and Distributed Computing*, 72(10):1386 – 1396, 2012.

[10] João Barreto, Aleksandar Dragojevic, Paulo Ferreira, Ricardo Filipe, and Rachid Guerraoui. Unifying thread-level speculation and transactional memory. In *Middleware '12*.

[11] R.C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *Device and Materials Reliability, IEEE Transactions on*, 5(3):305–316, 2005.

[12] D. Bernick, B. Bruckert, P.D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. NonStop advanced architecture. DSN, 2005.

[13] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[14] Geoffrey Blake, Ronald G. Dreslinski, and Trevor Mudge. Proactive transaction scheduling for contention management. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 156–167, New York, NY, USA, 2009. ACM.

[15] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13:422–426, July 1970.

[16] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Micro, IEEE*, 25(6), 2005.

[17] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, 2006.

[18] Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. Robust architectural support for transactional memory in the power architecture. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 225–236, New York, NY, USA, 2013. ACM.

[19] Irina Calciu, Justin Gottschlich, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. Invyswell: A hybrid transactional memory for haswells restricted transactional memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*, PACT '14, 2014.

[20] Irina Calciu, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. Improved single global lock fallback for best-effort hardware transactional memory. In *9th Workshop on Transactional Computing*, TRANSACT '14, 2014.

[21] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08*.

[22] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM TOCS*, 20(4), 2002.

[23] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, ISCA '06, pages 227–238, Washington, DC, USA, 2006. IEEE Computer Society.

[24] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, pages 227–238, Washington, DC, USA, 2006. IEEE Computer Society.

[25] D. Christie, J.W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, et al. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *Proceedings of the 5th European conference on Computer systems*, pages 27–40. ACM, 2010.

[26] Dave Christie, Jae-Woong Chung, Stephan Diestelhorst, Michael Hohmuth, Martin Pohlack, Christof Fetzer, Martin Nowack, Torvald Riegel, Pascal Felber, Patrick Marlier, and Etienne Rivière. Evaluation of amd's advanced synchronization facility within a complete transactional memory stack. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 27–40, New York, NY, USA, 2010. ACM.

[27] Byung-Gon Chun, Petros Maniatis, and Scott Shenker. Diverse replication for single-machine byzantine-fault tolerance. In *USENIX Annual Technical Conference*, pages 287–292, 2008.

[28] Byung-Gon Chun, Petros Maniatis, and Scott Shenker. Diverse replication for single-machine byzantine-fault tolerance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC, 2008.

[29] Intel Coorporation. Intel 64 and IA-32 architectures optimization reference manual (Section 12.1.1), 2014. URL: `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf`.

[30] Miguel Correia, Daniel Gómez Ferro, Flavio P Junqueira, and Marco Serafini. Practical hardening of crash-tolerant systems. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference, USENIX ATC*, 2012.

[31] TPC Council. TPC-C benchmark. 2010.

[32] Luke Dalessandro, François Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. Hybrid NOrec: A case study in the effectiveness of

best effort hardware transactional memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 39–52, New York, NY, USA, 2011. ACM.

[33] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. Norec: Streamlining stm by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 67–78, New York, NY, USA, 2010. ACM.

[34] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *PPoPP '10*, 2010.

[35] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 336–346, New York, NY, USA, 2006. ACM.

[36] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 336–346, New York, NY, USA, 2006. ACM.

[37] X. Defago, A. Schiper, and P. Urban. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36, 2004.

[38] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In Shlomi Dolev, editor, *Distributed Computing*, volume 4167 of *Lecture Notes in Computer Science*, pages 194–208. Springer Berlin Heidelberg, 2006.

[39] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *In Proc. of the 20th Intl. Symp. on Distributed Computing*, 2006.

[40] David Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC '06*.

[41] Nuno Diegues and Paolo Romano. Self-tuning intel transactional synchronization extensions. In *11th International Conference on Autonomic Computing*, ICAC '14. USENIX Association, 2014.

[42] Nuno Lourenco Diegues and Paolo Romano. Time-warp: lightweight abort minimization in transactional memory. In *PPOPP*, pages 167–178, 2014.

[43] Shlomi Dolev, Danny Hendler, and Adi Suissa. CAR-STM: Scheduling-based Collision Avoidance and Resolution for Software Transactional Memory. In *Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing*, PODC '08, pages 125–134, New York, NY, USA, 2008. ACM.

[44] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. In *PLDI '09*.

[45] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 155–165, New York, NY, USA, 2009. ACM.

[46] Aleksandar Dragojević, Rachid Guerraoui, Anmol V. Singh, and Vasu Singh. Preventing versus curing: Avoiding conflicts in transactional memories. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC '09, pages 7–16, New York, NY, USA, 2009. ACM.

[47] Justin E. Gottschlich, Manish Vachharajani, and Jeremy G. Siek. An efficient software transactional memory using commit-time invalidation. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 101–110, New York, NY, USA, 2010. ACM.

[48] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM.

[49] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional Memory Coherence and Consistency. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 102–, Washington, DC, USA, 2004. IEEE Computer Society.

[50] T. Harris, J. Larus, and R. Rajwar. Transactional memory. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, 2010.

[51] Tim Harris, James Larus, and Ravi Rajwar. Transactional memory, 2nd edition. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, 2010.

[52] Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. Remote invalidation: Optimizing the critical path of memory transactions. In *IPDPS*, 2014.

[53] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *ACM SIGPLAN Notices*, volume 41, pages 253–262. ACM, 2006.

[54] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[55] B. Hindman and D. Grossman. Atomicity via source-to-source translation. In *Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 82–91. ACM, 2006.

[56] Sachin Hirve, Roberto Palmieri, and Binoy Ravindran. Hipertm: High performance, fault-tolerant transactional memory. In *ICDCN*, pages 181–196, 2014.

[57] Intel Corporation. Intel C++ STM Compiler 4.0, Prototype Edition. `http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/`, 2009.

[58] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about Eve: Execute-verify replication for multi-core servers. OSDI, 2012.

[59] Bettina Kemme, Fernando Pedone, Gustavo Alonso, Andre Schiper, and Matthias Wiesmann. Using optimistic atomic broadcast in transaction processing systems. *IEEE TKDE*, 15(4), 2003.

[60] Sangman Kim, Michael Z. Lee, Alan M. Dunn, Owen S. Hofmann, Xuan Wang, Emmett Witchel, and Donald E. Porter. Improving server applications with system transactions. EuroSys, 2012.

[61] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with Java STM. In *Third Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*, 2010.

[62] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. SOSP, 2007.

[63] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 209–220, New York, NY, USA, 2006. ACM.

[64] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[65] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[66] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[67] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4:382–401, 1982.

[68] Yossi Lev and Jan-Willem Maessen. Split hardware transactions: true nesting of transactions using best-effort hardware transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 197–206, New York, NY, USA, 2008. ACM.

[69] Yossi Lev, Mark Moir, and Dan Nussbaum. PhTM: Phased transactional memory. In *2nd Workshop on Transactional Computing*, TRANSACT '07, 2007.

[70] S. Lie. Hardware support for unbounded transactional memory. Master's thesis, Massachusetts Institute of Technology, 2004.

[71] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. USENIX ATC'12, 2012.

[72] Guoming Lu, Ziming Zheng, and Andrew A. Chien. When is multi-version checkpointing needed? In *Proceedings of the 3rd Workshop on Fault-tolerance for HPC at Extreme Scale*, FTXS '13, 2013.

[73] Walther Maldonado, Patrick Marlier, Pascal Felber, Adi Suissa, Danny Hendler, Alexandra Fedorova, Julia L. Lawall, and Gilles Muller. Scheduling support for transactional memory contention management. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 79–90, New York, NY, USA, 2010. ACM.

[74] V.J. Marathe, M.F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W.N. Scherer III, and M.L. Scott. Lowering the overhead of nonblocking software transactional memory. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, TRANSACT '06, 2006.

[75] Alexander Matveev and Nir Shavit. Reduced hardware NOrec. In *5th Workshop on the Theory of Transactional Memory*, WTTM '13, 2013.

[76] Alexander Matveev and Nir Shavit. 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*, SPAA '13, pages 11–22, New York, NY, USA, 2013. ACM.

[77] Chi Cao Minh, Jaewoong Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 35–46, Sept 2008.

[78] Mohamed Mohamedin, Roberto Palmieri, and Binoy Ravindran. Managing soft-errors in transactional systems. In *Proceedings of the 19th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems*, DPDNS '14, 2014.

[79] K.E. Moore, J. Bobba, M.J. Moravan, M.D. Hill, and D.A. Wood. LogTM: log-based transactional memory. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 254 – 265, feb. 2006.

[80] J.E.B. Moss and A.L. Hosking. Nested transactional memory: Model and architecture sketches. *Science of Computer Programming*, 63(2):186–201, 2006.

[81] ObjectFabric Inc. ObjectFabric. `http://objectfabric.com`, 2011.

[82] Palmieri, Quaglia, and Romano. Osare: Opportunistic speculation in actively replicated transactional systems. In *SRDS*, 2011.

[83] Roberto Palmieri, Francesco Quaglia, and Paolo Romano. Aggro: Boosting STM replication via aggressively optimistic transaction processing. In *NCA*, pages 20–27, 2010.

[84] Roberto Palmieri, Francesco Quaglia, and Paolo Romano. Osare: Opportunistic speculation in actively replicated transactional systems. In *SRDS*, pages 59–64, 2011.

[85] Fernando Pedone, Rachid Guerraoui, and André Schiper. The database state machine approach. *Distrib. Parallel Databases*, 14(1):71–98, July 2003.

[86] Fernando Pedone, Rachid Guerraoui, and André Schiper. The database state machine approach. *Distributed and Parallel Databases*, 14(1), 2003.

[87] Sebastiano Peluso, Paolo Romano, and Francesco Quaglia. Score: A scalable one-copy serializable partial replication protocol. In *Middleware*, 2012.

[88] Sebastiano Peluso, Pedro Ruivo, Paolo Romano, Francesco Quaglia, and Luís Rodrigues. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *ICDCS*, pages 455–465, 2012.

[89] Jesse Pool, Ian Sin Kwok Wong, and David Lie. Relaxed determinism: Making redundant execution on multiprocessors practical. In *HotOS*, 2007.

[90] Laura L Pullum. *Software fault tolerance techniques and implementation*. Artech House, 2001.

[91] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, pages 494 – 505, june 2005.

[92] James Reinders. Transactional synchronization in haswell. *Intel Software Network. URL: http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/*, 2012.

[93] T. Riegel, P. Felber, and C. Fetzer. TinySTM. `http://tmware.org/tinystm`, 2010.

[94] Torvald Riegel, Patrick Marlier, Martin Nowack, Pascal Felber, and Christof Fetzer. Optimizing hybrid transactional memory: The importance of nonspeculative operations. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 53–64, New York, NY, USA, 2011. ACM.

[95] Paolo Romano, Carvalho, and Luís Rodrigues. Towards distributed software transactional memory systems. In *LADIS '08*, 2008.

[96] Wenjia Ruan, Yujie Liu, and Michael Spear. Boosting timestamp-based transactional memory by exploiting hardware cycle counters. In *TRANSACT '13*.

[97] B. Saha, A-R. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 185–196, Dec 2006.

[98] Fred B. Schneider. *Replication management using the state-machine approach*. ACM Press/Addison-Wesley Publishing Co., 1993.

[99] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213. ACM, 1995.

[100] N. Shavit and D. Touitou. Software transactional memory. In *PODC '95*, 1995.

[101] Michael F. Spear, Maged M. Michael, and Christoph von Praun. RingSTM: Scalable transactions with a single atomic instruction. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 275–284, New York, NY, USA, 2008. ACM.

[102] J.M. Stone, H.S. Stone, P. Heidelberger, and J. Turek. Multiple reservations and the Oklahoma update. *Parallel Distributed Technology: Systems Applications, IEEE*, 1(4):58 –71, nov 1993.

[103] Tilera Corporation. *TILE-Gx Processor Family*. `http://www.tilera.com`.

[104] Alexandru Turcu, Binoy Ravindran, and Roberto Palmieri. Hyflow2: a high performance distributed transactional memory framework in scala. In *PPPJ*, pages 79–88, 2013.

[105] University of Rochester. Rochester Software Transactional Memory. `http://www.cs.rochester.edu/research/synchronization/rstm/index.shtml`, `http://code.google.com/p/rstm`, 2006.

[106] Rob F. van der Wijngaart, Timothy G. Mattson, and Werner Haas. Light-weight communications on intel's single-chip cloud computer processor. *SIGOPS Oper. Syst. Rev.*, 45(1), 2011.

[107] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. SOSP, 2007.

[108] Matthias Wiesmann, André Schiper, Fernando Pedone, Bettina Kemme, and Gustavo Alonso. Database replication techniques: A three parameter classification. In *SRDS*, pages 206–215, 2000.

[109] Gulay Yalcin, Anita Sobe, Alexey Voronin, Jons-Tobias Wamhoff, Derin Harmanci, Adrian Cristal, Osman Unsal, Pascal Felber, and Christof Fetzer. Combining error detection and transactional memory for energy-efficient computing below safe operation margins. In *Parallel, Distributed, and Network-Based Processing (PDP)*, PDP, 2014.

[110] L. Yen, J. Bobba, M.R. Marty, K. Moore, H. Volos, M.D. Hill, M.M. Swift, and D.A Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 261–272, Feb 2007.

[111] L. Yen, J. Bobba, M.R. Marty, K.E. Moore, H. Volos, M.D. Hill, M.M. Swift, and D.A. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 261 –272, feb. 2007.

[112] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP, 2003.

[113] Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 169–178, New York, NY, USA, 2008. ACM.

[114] Yuanyuan Zhou, Darko Marinov, William Sanders, Craig Zilles, Marcelo d'Amorim, Steven Lauterburg, Ryan M. Lefever, and Joe Tucek. Delta execution for software reliability. In *Proceedings of the 3rd Workshop on on Hot Topics in System Dependability*, HotDep, 2007.