On Optimizing Transactional Memory: Transaction Splitting, Scheduling, Fine-grained Fallback, and NUMA Optimization

Mohamed Mohamedin

Dissertation 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

July 30, 2015
Blacksburg, Virginia

Keywords: Transaction Memory, Hardware Transaction Memory (HTM), Best-efforts HTM, Transactions Partitioning, Transactions Scheduling, NUMA, NUMA Optimization, NUMA-aware STM, Fine-grained Fallback
Copyright 2015, Mohamed Mohamedin

On Optimizing Transactional Memory: Transaction Splitting, Scheduling, Fine-grained Fallback, and NUMA Optimization

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 generation. 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 and without deeply knowing the application logic. A new, easier, synchronization abstraction is therefore required and Transactional Memory (TM) is the concrete 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, optimistically. In the past decade, TM researchers worked extensively to improve TM-based systems. Most of the work has been dedicated to Software TM (or STM) as it does not requires special transactional hardware supports. Very recently (in the past two years), those hardware supports have become commercially available as commodity processors, thus a large number of customers can finally take advantage of them. Hardware TM (or HTM) provides the potential to obtain the best performance of any TM-based systems, but current HTM systems are best-effort, thus transactions are not guaranteed to commit in any case. In fact, HTM transactions are limited in size and time as well as prone to livelock at high contention levels.

Another challenge posed by the current multi-core hardware platforms is their internal architecture used for interfacing with the main memory. Specifically, when the common computer deployment changed from having a single processor to having multiple multi-core processors, the architects redesigned also the hardware subsystem that manages the memory access from the one providing a Uniform Memory Access (UMA), where the latency needed to fetch a memory location is the same independently from the specific core where the thread executes on, to the current one with a Non-Uniform Memory Access (NUMA), where such a latency differs according to the core used and the memory socket accessed. This switch in technology has an implication on the performance of concurrent applications. In fact, the building blocks commonly used for designing concurrent algorithms under the assumptions of UMA (e.g., relying on centralized meta-data) may not provide the same high performance and scalability when deployed on NUMA-based architectures.

In this dissertation, we tackle the performance and scalability challenges of multi-core architectures by providing three solutions for increasing performance using HTM (i.e., Part-htm, Octonauts, and Precise-tm), and one solution for solving the scalability issues provided by NUMA-architectures (i.e., Nemo).

- Part-htm is the first hybrid transactional memory protocol that solves the problem of transactions aborted due to the resource limitations (space/time) of current best-effort

HTM. The basic idea of Part-htm is to partition those transactions into multiple sub-transactions, which can likely be committed in hardware. Due to the eager nature of HTM, we designed a low-overhead software framework to preserve transaction's correctness (with and without opacity) and isolation. Part-htm is efficient: our evaluation study confirms that its performance is the best in all tested cases, except for those where HTM cannot be outperformed. However, in such a workload, Part-htm still performs better than all other software and hybrid competitors.

- 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 a performance improvement up to 60% when Octonauts is deployed in comparison with pure HTM with falling back to global locking.

- Precise-tm is a unique approach to solve the granularity of the software fallback path of best-efforts HTM. It provide an efficient and precise technique for HTM-STM communication such that HTM is not interfered by concurrent STM transactions. In addition, the added overhead is marginal in terms of space or execution time. Precise-tm uses address-embedded locks (pointers bit-stealing) for a precise communication between STM and HTM. Results show that our precise fine-grained locking pays off as it allows more concurrency between hardware and software transactions. Specifically, it gains up to 5× over the default HTM implementation with a single global lock as fallback path.

- Nemo is a new STM algorithm that ensures high and scalable performance when an application workload with a data locality property is deployed. Existing STM algorithms rely on centralized shared meta-data (e.g., a global timestamp) to synchronize concurrent accesses, but in such a workload, this scheme may hamper the achievement of scalable performance given the high latency introduced by NUMA architectures for updating those centralized meta-data. Nemo overcomes these limitations by allowing only those transactions that actually conflict with each other to perform inter-socket communication. As a result, if two transactions are non-conflicting, they cannot interact with each other through any meta-data. Such a policy does not apply for application threads running in the same socket. In fact, they are allowed to share any meta-data even if they execute non-conflicting operations because, supported by our evaluation study, we found that the local processing happening inside one socket does not interfere with the work done by parallel threads executing on other sockets. Nemo's evaluation study shows improvement over state-of-the-art TM algorithms by as much as 65%.

This dissertation is supported in part by US National Science Foundation under grant CNS 1217385, and by AFOSR under grants FA9550-14-1-0163, and FA9550-14-1-0187.

Dedication

To my father, you always dreamed of seeing one of your children (especially me) holding a PhD degree. Unfortunately, you died few months before your dream comes true. I know you are happy now. Your spirit is always around me. I pray for you everyday that God forgives all your sins and bestow Heaven upon you. You are my hero, role model, friend, and the best father. I love you so much and I will always remember you.

To my mother, you give me all kinds of support, care and love. No matter how old I am, I will always be your kid and will always need you. You are always there when I need you. May God give you a long, healthy, and happy life, and may God provide me the strength to serve you and make you happy.

And to my soul mate, my wife, Germin. Without you I would not be here.

I cannot find the right words to describe how grateful I am to all of you.

Acknowledgments

First and before all, I praise God, the almighty, for giving me the strength and perseverance to accomplish this dissertation. He helped me jumping over all the obstacles, and provided me with the best people to help and support me.

I would like to thank my advisor, Dr. Binoy Ravindran, for all his endless help and support. Without his help and patience during my dark start I could not make it. I am really grateful for him. I learned a lot from him. Also, I would like to thank my mentor, co-advisor and friend, Dr. Roberto Palmieri, for guiding me closely throughout every detail. He exerted huge efforts with me, and everyone else in the team. I am also really grateful for him.

I would also like to thank my committee members: Dr. Leyla Nazhandali, Dr. Mohamed Rizk, Dr. Paul Plassmann, and Dr. Robert P. Broadwater, for their suggestions and guidance. I'm lucky and honored for having them serving in my committee.

Special thanks to Dr. Roberto Palmieri, Ahmed Hassan, and Dr. Sebastiano Peluso. I believe that together we are a great research team. I'm grateful to everyone of them. I enjoyed brainstorming with them, our fruitful discussions, our arguments, and of course their continuous help. Ahmed Hassan deserves a special “special” thanks. He is like a brother to me. Also, thanks to Mohamed Saad Ibrahim, whom I started my research career with him. He is my partner since my undergraduate study in everything.

I would like to thank my wife, Germin, and my kids Amgad, Jude, and Sidra for their endless love. They made my life joyful and gave me a spiritual charge to continue. Without them, my life would be hollow and I would never be able to make it. And without Germin especially, there would be no me. Also, I would like to thank my parents. Their encouragement and spiritual support are always essential for my success. “My Lord! bestow on them thy Mercy even as they cherished me in childhood”. Special thanks to my brother Mahmoud who took care of my parents and me in all aspects. And thanks to my sisters Amal, Suhair, and Abir for all their support and remembering me in their prayers.

I would like to thank my colleagues Sachin Hirve, Dr. Antonio Barbalace, Dr. Vincent Legout, Anthony Carno, Robert Lyerly, Duane Niles, Alex Turcu, and everyone in the Systems Software Research Group (SSRG) for their help and support in addition to the friendly atmosphere in the group.

Finally, I would like to thank my friends, in no particular order, Mohammed Magdy Farag, Amr Hilal, Haithem Ezzat Taha, Mohamed Medhat Abdel-Raheem, Mohamed Azab, Amr Abed, Ahmed Said Eltrass, Bassem Mokhtar, Mohammed El-Shambakey, Mohammed El-henawy, Mostafa Ali, Abdullah Awaysheh, Mohamed Zein, Karim Said, Nader Shehata, Islam Ashry, Ishac Kandas, Abdelrahman Eldosouky, Mohammed Fawzy Seddik, Mohamed Handosa, Ahmed Ghanem, Hassan Mahsoub, Atia Eisa, Dr. Alamir, Hosam Shahin, Haitham Elmarakeby, Amr Nabil, Hassan Eldib, Ahmed Khalifa, Mostafa Taha, Samir Alghool, Mohammed Shafae, and Hamdy Fayez Mahmoud for their encouragement, help, support, sincere friendship, and being the best community I ever lived in. They made my life much easier and enjoyable. Special thanks to Dr. Sedki Riad, Dr. Yasser Hanafy, Dr. Mustafa ElNainay, and all VT-MENA program team for all their continuous efforts. And Thanks to Dr. Sedki again for being like a father to me. He shed his love and care all over me. He always listened to me and gave me great advices.

Contents

1 Introduction **1**

1.1 Summary of Dissertation Contributions 3

1.1.1 Improving Multi-core Performance Exploiting HTM 3

1.1.2 Scalable NUMA-aware TM 7

1.2 Dissertation Outline 9

2 Related Work **10**

2.1 Performance Improvement Using HTM 10

2.2 Transactional Memory Scheduling 13

2.3 Solution for NUMA architectures 14

2.3.1 Database 15

2.3.2 Transactional Memory 15

3 Background **17**

3.1 Parallel Programming 17

3.2 Transactional Memory 18

3.2.1 TM Design Classification 18

3.2.2 Version Control 19

3.3 Transactional Memory Algorithms 20

3.3.1 TL2 20

3.3.2 RingSTM 21

3.3.3 NOrec 22

3.3.4 Reduced Hardware NOrec (RH-NOrec) 23
3.3.5 TLC 23
3.4 Intel's Haswell HTM processor 23

4 Part-HTM **26**

4.1 Problem Statement 26
4.1.1 Intel's HTM Limitations 29
4.2 Algorithm Design 29
4.3 Algorithm Details 33
4.3.1 Protocol Meta-data 33
4.3.2 Begin Operation and Partitioning 35
4.3.3 Transactional Read and Write Operation 35
4.3.4 Validation, Commit and Abort 36
4.3.5 Ensuring Opacity 40
4.4 Compatibility with other HTM processors 42
4.5 Correctness 42
4.6 Evaluation 44

5 Octonauts **50**

5.1 Problem Statement 50
5.2 Algorithm Design 52
5.3 Algorithm Details 53
5.3.1 Reducing Conflicts via Scheduling 53
5.3.2 HTM-aware Scheduling 55
5.3.3 Transactions Analysis 56
5.3.4 Adaptive Scheduling 57
5.4 Evaluation 58
5.4.1 Bank 58
5.4.2 TPC-C 59

6 Precise-TM **61**

6.1 Problem Statement 61

6.1.1 Drawbacks of Using a Single Global Lock as *Slow-path* 62

6.1.2 On Reducing the Effect of Global Locking 62

6.2 Precise-TM Design Principle: Fine-Grained Embedded Locks 64

6.3 Algorithm Details 66

6.3.1 Precise-TM-V1: Precise Monitoring in the *Fast-Path* 66

6.3.2 Precise-TM-V2: Precise Locking in the *Slow-Path* 69

6.4 Evaluation 71

6.4.1 Bank 72

6.4.2 EigenBench 74

6.4.3 Linked-list 75

7 Nemo **77**

7.1 Problem Statement 77

7.2 Non-Uniform Memory Access: architecture, characteristics, and performance using atomic operations 80

7.3 Algorithm Design 83

7.4 Algorithm Details 86

7.4.1 Nemo-TS 86

7.4.2 Nemo-Vector 88

7.4.3 NUMA Memory Allocator 91

7.4.4 Nemo and multi-sockets HTM architectures 92

7.5 Evaluation 92

7.5.1 Effect of Inter-NUMA zones Transactions 96

7.5.2 Summary 97

8 Conclusions **99**

8.1 Summary of Contributions 100

8.2 Future Work 101

Bibliography **102**

List of Figures

4.1 Part-htm's Basic Idea. 30
4.2 Comparison between HTM, STM, and Part-htm. 33
4.3 Part-htm's pseudo-code. Procedures marked as * are executed in software. 34
4.4 Acquiring write-locks (a). Detecting intermediate reads or potential overwrites (b). Releasing write-locks (c). 37
4.5 Part-htm-o's pseudo-code. Procedures marked as * are executed in software. 39
4.6 Throughput using N-Reads M-Writes benchmark. 45
4.7 Throughput using Linked-List. 45
4.8 Speed-up over sequential (non-transactional) execution using applications of STAMP Benchmark. 47
4.9 Speed-up over sequential (non-transactional) execution using Labyrinth as in [84]. 48
4.10 Speed-up over sequential (non-transactional) execution using EigenBench. . . 49

5.1 Scheduling transactions. 53
5.2 Readers-Writers ticketing technique. 54
5.3 HTM-STM communication. 55
5.4 Throughput using Bank benchmark. 57
5.5 Execution time using Bank benchmark (lower is better). 58
5.6 Throughput using TPC-C benchmark. 59
5.7 Execution time using TPC-C benchmark. 59

6.1 Execution time using Bank benchmark. 72

6.2 Execution time using Bank benchmark with disjoint accesses to accounts. . . 73
6.3 Execution time using EigenBench benchmark for two special cases. 74
6.4 Execution time using linked-list benchmark of 5K elements and 20% write operations. 75

7.1 Hardware architecture of an AMD Opteron 64-cores (4 sockets and a 16 cores processor per socket. 81
7.2 Bank benchmark configured for producing a scalable workload (disjoint transactional accesses). 82
7.3 Average cost of 100k increment operations of a centralized vs local timestamp. 83
7.4 Nemo-TS's pseudo-code. 86
7.5 Nemo-Vector's pseudo-code. 89
7.6 Lock-free version of Nemo-Vector's pseudo-code. 91
7.7 Comparison of a centralized global lock and a global NUMA-lock. 93
7.8 Throughput using Bank benchmark. 94
7.9 Throughput using NUMA-Linked-list benchmark. 95
7.10 Throughput using TPC-C benchmark. 96
7.11 Throughput using Bank benchmark under different inter-NUMA-zone transactions percentage. The number of threads is 48. 97
7.12 Zooming Figure 7.11 in between the 0% datapoint and the 10% datapoint. . 97

List of Tables

4.1 Statistics' comparison between HTM-GL (A) and Part-htm (B) using Labyrinth application and 4 threads. 48

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 higher performance when deployed on an upgraded hardware with more cores and same clock speed as before. Unfortunately, multi-core programming is an advanced topic, often suited for expert programmers only. 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 vast majority of programmers to do concurrent programming, efficiently and correctly.

An effective abstraction is Transactional Memory (TM) [51, 48, 89, 58, 30]. TM programming model is as easy as coarse-grained locking, such that 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 operation. TM also guarantees atomicity, consistency, and isolation. Thus, a transaction is executed all or nothing, always observing a consistent state, and works in isolation from other transactions.

In terms of performance, TM is likely to have comparable performance to fine-grained locking, which in principle uses the minimal number of locks needed for preserving the correctness of the program. Unfortunately, fine-grained locking cannot be composable, e.g., two atomic blocks implemented using fine-grained 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) [35, 29], which can be executed without any trans-

actional hardware support; *hardware* (HTM) [51, 48, 24], which exploits specific hardware facilities; and *hybrid* (HyTM) [58, 30], which mixes both HTM and STM.

TM has been studied extensively for the past decade [51, 48, 89, 52, 92, 79, 57, 49, 16, 90, 35, 29]. 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 [54], followed by inclusion of TM constructs in C++11/C1X new standards that were finally adopted by the famous GCC compiler starting from version 4.7.

Hardware TM is available commercially now in commodity CPUs (i.e., Intel Haswell) and HPC (i.e., IBM Blue Gene/Q and IBM Power 8 [17]). Both Intel's and IBM's HTMs are *best-effort* 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, 2, 20, 69, 68, 37, 1]. Unfortunately, the well studies STM techniques cannot be ported directly to HTM or Hybrid TM. With many high performance STM algorithms (e.g., TL2 [35]), the performance is highly degraded compared to both plain HTM and plain STM [81, 69]. Thus, Hybrid TM algorithms requires careful design to balance the added overhead on both HTM and STM components.

Besides the great programmability, TM algorithms should provide their best performance when deployed on multicore architectures. Those architectures are mostly multi-sockets, which means that more than one multicore chip is deployed on the same hardware platform, and they coordinate with each other in order to handle application requests. The de-facto standard for the memory management of such a multicore architecture involves non-uniform communication latencies between the chip where the thread is executing and the physical socket where the memory location is stored, also called as Non-Uniform Memory Access (or NUMA) [65, 8, 96, 25].

Even though it has been proved to be an effective solution to handle memory accesses of such a high number of parallel threads [65], it imposes new challenges in designing TM algorithms. In particular, when the concurrent application generates a workload with partitioned accesses, where a tight locality relation between data and executing threads can be established, we found that most of the approaches in literature fail in ensuring high performance given the constraints of the underlying hardware. As an example of that, they rely on shared meta-data, updated anytime a writing transaction is committed, even if no conflict happened during its execution. In this case, the hardware itself introduces a cost (hidden to the programmer) in updating a meta-data that is possibly located in a memory socket

different from the one mostly used by the executing thread. This cost can (and should) be avoided when the committing transaction does not observe any conflict during its execution. Most existing and well-known TM protocols suffer from this weakness, thus exposing serious scalability bottlenecks.

Motivated by these observations, this dissertation focuses on two aspects of concurrent computation: improving multi-core performance by exploiting best-efforts HTM while overcoming its limitations; providing a scalable solution for maximizing the effectiveness of locality-aware applications by exploiting the NUMA organization.

1.1 Summary of Dissertation 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 tuned 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, fine-grained locking requires much higher level of expertise to avoid the problems of deadlocks, livelocks, lock-convoying, 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, if we have two such hash tables and we want to atomically remove an entry from one table and add it to the other, the individual 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-grained locking technique, it is algorithm specific and hard to generalize.

Transactional Memory (TM) brings down the parallel programming interface to the level of coarse-grained locking while still 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 it being best-effort HTM. An HTM transaction is not guaranteed to commit even if it runs alone with no contention, since it has resource limitations (e.g., hardware buffer size, hardware interrupts) which forces the transaction to abort. 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. Due to these lim-

itations, a fallback path is required by best-effort 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. Another problem facing current HTM is livelocks under medium/high contention levels. When the contention level increases, transactions can abort each other repeatedly. In addition, providing a fine-grained HTM fallback path is a challenge that can alleviate the GL-software path bottleneck.

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 limitations to fall back to the GL-software path, and commit them still exploiting the advantages of HTM. As a result, Part-htm limits the number of 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 each transaction as HTM and if the transaction aborts 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 have minimal 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 at 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 better than 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 the best trade-off between STM and HTM transactions.

Opacity [45] is the reference correctness criterion for TM implementations because it avoids any inconsistency during the execution, independently from the final transaction outcome (either commit or abort). However, ensuring opacity in Part-htm is challenging because its overhead could nullify Part-htm's benefits. While acknowledging the importance of an opaque hybrid-TM protocol, in this dissertation we present two versions of Part-htm. One aims at obtaining the best performance by relaxing opacity in favor of serializability [13], the well-known consistency criterion for online transaction processing, and by relying on the HTM protection mechanism (i.e., sandboxing), which protects from faulty computations (e.g., division by zero). In the second version, we enriched Part-htm for ensuring opacity but, at the same time, we present a set of innovations (e.g., *address-embedded* write locks) for reducing the transaction's memory footprint so that the overhead is kept limited (less than the achievable gain).

We evaluated Part-htm on a wide range of benchmarks including a micro-benchmark, a data structure, the STAMP suite [70], and EigenBench [53]. As competitors, we selected a pure HTM with GL-software path as a fallback, two state-of-the-art STM protocols (RingSTM [90], NOrec [29]) and a recent HybridTM (RH-NOrec [69]). Results show that Part-htm is the best in almost all the tested cases, except those where HTM outperforms all (and therefore no competitor can perform better than that). In these workloads, Part-htm still represents the best among other STM and HybridTM alternatives. The combination of these two cases gives Part-htm the unique characteristic of being an effective trade-off, independently from the application workload.

Octonauts

Part-htm tackled the problem of HTM resource limitations, but another problem afflicts TM in general i.e., efficiently handling conflicts. At high contention levels, transactions keep aborting each other and can lead to a 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 among 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 either provided by the programmer or gathered 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) [68]. 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 STM transactions 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 started directly using global locking.

We evaluated Octonauts using two benchmarks; Bank and TPC-C [27]. At high-contention levels, Octonauts showed its advantages specially at higher number of threads ($1\times$ better than HTM-GL at 8 threads). Using TPC-C, where transactions are more complex and contention is high, Octonauts results are better than HTM-GL starting from 4 threads. Octonauts also managed to handle multi-programming efficiently. Instead of letting threads fight for the limited number of available cores, Octonauts orchestrate them and prevented most of the conflicts between the concurrently scheduled transactions.

Precise-TM

One problem that is orthogonal to Part-htm and Octonauts is how to optimize the final global lock software path without slowing down the HTM fast-path. A global lock stops (i.e., aborts) all HTM transactions and allows only one transaction to proceed in software at a time. It favors the slow-path over the fast-path to guarantee the correctness of the execution of those transactions that repeatedly fail in hardware due to resource constraints or invocation of special instructions that cannot be performed as a part of an HTM transaction. Fine-grained locking fallback path can solve this problem but it adds (again) complexity on the programming model, which goes against the original purpose of TM, namely to

simplify the implementation of concurrent applications. One solution is to use a fine-grained TM algorithm like TL2, but the overhead of monitoring object's meta-data in the HTM fast-path would slow down the application performance and also consume precious HTM resources (e.g., additional cache lines).

Precise-tm is designed to solve this problem. Its core idea is to replace the global lock, which is acquired in the slow-path and monitored in the fast-path, with fine-grained locks without incurring in the high overhead that characterized the previous fine-grained proposals.

To avoid such a high overhead, Precise-tm uses the following innovations: *1)* locking/monitoring memory references by using the concept of *address-embedded-locks* (i.e., bit stealing from the memory address itself); *2)* using the traditional global lock for scalar variables; *3)* HTM transactions can ignore monitoring the global lock if the critical sections of the application contain only references. Embedded locks are used to notify HTM transactions of read and write operations performed by transactions executing in the software path.

We design two versions of Precise-tm. The first version (Precise-tm-v1) uses a global lock in the *slow-path* but HTM transactions do not necessarily monitor it in the *fast-path*. Rather, the monitoring of the global lock begins when the *fast-path* reaches a read/write operation that cannot be monitored using the *address-embedded-locks* technique (e.g., on a scalar variable). The second version (Precise-tm-v2) uses the *address-embedded-locks* technique as fine-grained locks in the slow-path instead of the global lock.

Results of Precise-tm show that the precise fine-grained locking (in the *slow-path*) and monitoring (in the *fast-path*) pays off as it allows more concurrency between transactions, reduces the false conflicts, and minimizes the added meta-data.

1.1.2 Scalable NUMA-aware TM

Most of current TM algorithms do not scale when deployed on NUMA architectures. Such algorithms use some centralized global meta-data that is frequently updated. This general scheme generates a high traffic on the shared bus that connect the different memory sockets (or NUMA-zones), with a high performance penalty resulting in slowing down the system. From our preliminary evaluation, NUMA architectures can handle atomic operations inside each NUMA-zone efficiently, without affecting the performance of threads operating on other NUMA-zones. In addition to that, there are many workloads that exhibit capability to be partitioned, namely where transactions mostly access local data (i.e., stored in the same NUMA-zone where the thread is directly connected to) and sometimes access some "remote" data (i.e., stored in another NUMA-zone). Existing TM algorithms fail in providing scalable performance.

Nemo

Nemo is a scalable NUMA-aware STM algorithm that is designed based on the following two principles: *1)* Within a single NUMA-zone, we can use the simple and efficient centralized meta-data conflict detection; *2)* The inter-NUMA-zones interactions are limited to the cases where the application itself explicitly requests an access to that shared object. The first principle is based on the fact that the workload is NUMA-local, and the atomic operations within a single NUMA-zone are fast and do not affect other NUMA-zones operations. With such a design, the common case becomes fast and efficient to handle, without unnecessary aborts. On the other hand, the second principle guarantees correctness of inter-NUMA zones transactions without affecting intra-NUMA (or NUMA-local) transactions operation. Thus, the uncommon case is correct and fast enough to not affect the high performance of the common case.

Nemo uses a single shared timestamp for synchronizing operations of threads executing within a NUMA-zone. This timestamp is updated every time a writing transaction commits, and it is also used as a means for detecting a (possibly dangerous) transactional access to a shared object created after the transaction begins. When a transaction conflicts with a transaction executing on another NUMA-zone, an additional synchronization is needed to preserve the protocol's correctness. To do that, each thread keeps a cached version of the timestamps of other NUMA-zones. The cached copies are updated once a transaction requests for an object located in a different NUMA-zone and it finds that the object is associated with a timestamp greater than the one of the cached copy. When such a case happens, the transaction undergoes an additional check, which reveals whether an abort/restart is needed or not. After that, the transaction updates its cached value of the other NUMA-zone's timestamp and can proceed.

In order to further reduce the overhead of unnecessary aborts due to outdated cache, we designed a version of Nemo that makes all the cached copies of the other NUMA-zone's timestamps available to all threads of one NUMA-zone. That way, transactions can benefit from accessing fresher cached timestamps, even if the thread they are running on never accessed that NUMA-zone.

Nemo provide Serializability [13] as the correctness level. This is because, both its versions make sure that the versions of the objects read during the transaction execution are still the same as those currently committed, before applying the modifications to the shared memory (i.e., committing the writes).

We evaluated Nemo using three benchmarks; Bank, Linked-list and TPC-C [27]. We configured these benchmarks such that they have NUMA-locality and also a percentage of inter-NUMA zones transactions. Bank is configured such that the contention level is low. The results show a near perfect scalability. Only TLC [12] managed to scale similarly since its false aborts are marginal in this benchmark configuration. Linked-list has a high contention level which affected the performance of all other competitors. Nemo has the best scalability

among others although the workload is not very scalable. TPC-C was configured to have moderate contention. In this benchmark, Nemo beats all other competitors and achieves a very good scalability.

1.2 Dissertation Outline

The rest of this dissertation 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. Chapter 6 shows how HTM fallback path can be fine-grained in Precise-tm. Chapter 7 details NUMA architectures issues with TM and how we designed a scalable STM algorithm (Nemo) for NUMA architectures. Finally, the dissertation conclusions are listed in Chapter 8.

Chapter 2

Related Work

2.1 Performance Improvement Using HTM

Research in Hybrid TM (HyTM) [30, 58, 87] started before the recent release of commodity processors with HTM capability.

PhTM [61] 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) [24], which attracted researchers to design the initial Hybrid TM systems [81, 28]. 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: IBM's HTM processors are available as Blue Gene/Q and Power8 [17]; Intel's HTM processor is released as Haswell. In the rest of this dissertation we mostly focus on Intel Haswell because it is much cheaper that its IBM competitor and thus its diffusion is already large.

The release of Haswell processors attracted more research on how to boost HTM capabilities via software [19, 2, 20, 69, 68, 37, 1]. Haswell is a best-effort HTM and requires a software fallback path. Intel suggested global locking (GL-software path) as a default fallback 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 by proposing different approaches.

- Improving the global locking fallback path in order to increase concurrency [20, 37].
- Using STM as a fallback path in order to reduce the amount of conflicts between concurrent HTM and STM transactions [81, 28, 19, 86, 34].
- Using reduced hardware transactions where only the STM commit procedure is executed as HTM transaction [68, 69].

More in detail, in [20] authors propose to defer the check of the global lock at the very 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 access inconsistent states during the execution. The latter problem is (partially [32]) solved relying on the Haswell HTM sandboxing protection mechanism. In [37], a self-tuning mechanism is proposed to decide the best suited fallback path for an aborted transaction.

In [81, 28], a hybrid version of the NOrec [29] 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 [44] 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 [86], a hybrid version of the Cohorts STM algorithm is presented (HyCo). HyCo uses a state machine to represent the algorithm. Each state has a set of properties that allow (disallow accordingly) some operations. For example, the serial state allows only one transaction to be in this state (resembling a global lock or an irrevocable transaction). The system moves from one state to another based on a set of events such as the begin or commit of a transaction. HyCo suffers from the problem of resource limitations as the results shows.

In [34], a new refined technique for lock elision is proposed. Lock elision is similar to HTM as it runs the critical sections protected by locks as transactions (thus they do not acquire the locks). In this work, more concurrency can be achieved. Instead of forcing all HTM transactions to wait until the global lock is released, both the HTM fast-path and the global lock fallback path proceed concurrently. To do that correctly, the critical section must be instrumented when the lock is taken. This work targets the same problem tackled by Precise-tm, namely it tries to be more fine-grained in the fallback path. Instead of having one global lock, they have an array of *orecs* (or locks). Only one software transaction is allowed at a time, but it communicates its reads and writes with concurrent HTM transactions leveraging the orecs instead of a single global lock. Precise-tm eliminates the need of using orecs by exploiting the *address-embedded-locks* technique, thus it is more fine-grained. In addition, Precise-tm-v2 uses the strict 2-phase locking to allow concurrency among transactions executing in the software path. It is worth to note that the refined lock-elision gives the best performance when the orecs table size is one, which resemble a single reader-writer lock and thus confirms the motivations of Precise-tm.

In [68], 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 RH-NOrec [69], 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.

Precise-tm tackles the problem of having a fine-grained fallback path for HTM transactions. This has twofolds benefits: first, HTM transactions can run concurrently with the transactions in the fallback path; second, HTM-HTM and HTM-STM transactions communicate only when they access the same objects (i.e., as in the disjoint-access parallelism property). This property is not available in all HyTM algorithms. For example, RH-NOrec [69] fast-path HTM transactions have to increment the timestamp at the end of each transaction, which introduces a contention point among all transactions (not necessarily those non-conflicting), and unnecessary aborts.

Other HyTM approaches that uses orecs show low performance as shown in [68, 81, 69]. This is because of the added overhead of false conflicts on shared orecs, and the added transaction's footprint because of reading/writing orecs which causes more capacity aborts.

The problem of partitioning memory operations to fit as a single HTM transaction is described also in [3]. 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 [3] because they do not provide a software framework for ensuring consistency and isolation of sub-HTM transactions as we do.

In [2], 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.

Authors of [60] presented SpHT, a general and effective technique for splitting best-effort hardware transactions. This approach also cannot solve the problem of aborts due to resource limitations because the last sub-HTM transaction will still have a write-set that is as big as the original transaction. In details, transactional writes are deferred and buffered in the write-set. Transactional reads are also logged in the read-set. That way, each partition can

validate the consistency of the all reads by validating the read-set. The last partition writes back all the write-set buffer, thus it has a write-set that is as big as the original transaction.

2.2 Transactional Memory Scheduling

Transactional memory scheduling has been studied extensively in Software Transactional Memory systems [39, 11, 95, 64, 83, 41, 7]. However, TM scheduling for HTM-base systems is not still well exlored.

Dragojević et. al. in [41] presented Shrink, a technique to schedule transactions dynamically based on expected working-sets. It uses transactions read- and write-set of committed transactions to predict the working-set of a new transaction from the same thread. We used a similar idea to predict the working set of HTM transaction (of the same profile) via a lightweight instrumentation. Recently, ProPS [82] proposed a similar idea to expect the probability of conflict between two transaction. ProPS collects the information from aborted transactions instead. It also focuses on long transactions. Unfortunately, we cannot extract information from aborted transactions in current Intel's HTM.

In [7], 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. In [6], Steal-On-Abort scheduler is extended to HTM architecture. A new hardware extensions are proposed that implements the algorithm. Changing the hardware architecture usually takes a long time and do not solve current hardware problems.

Adaptive Transaction Scheduler (ATS) [95] monitors the contention level of the system. When it exceeds a threshold, transactions scheduler takes control. Otherwise, transactions proceeds normally without scheduling. In ATS, one scheduling queue is used which serialize all conflicting transactions in the system (acts like a global lock).

CAR-STM is presented in [39]. 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].

Some approaches targeted the operating system scheduler itself (e.g., TxLinux [83] and SER [64]). Having a transactional aware OS scheduler has the benefit of avoiding scheduling a transaction that is doomed to conflict and abort. Other non-OS schedulers had to yield their time-slot after being scheduled by the OS.

In [37], 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 evaluated again. 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.

Concurrently with our work, Seer [38] has been proposed. Seer works on imprecise information collected by observing which transactions are active when a transaction is aborted. It probabilistically identifies transactions that most likely will conflict with each other. Then, it applies a fine-grained dynamic locking to serialize those conflicting transactions.

Another recent work by Xiang and Scott [93] uses *advisory locks* to serialize only the partition where conflicting access exists. The compiler statically analyze the code and define potential locations to place the advisory locks (i.e., contention hot spot), then at run time, and based on previous history, one of the advisory lock is taken. This work requires hardware extensions and is not compatible with the current Intel HTM release.

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 Solution for NUMA architectures

The release of NUMA (Non-Uniform Memory Access) architectures put a pressure on software developers to be NUMA-aware. Hardware vendors tried to make NUMA architectures more appealing by providing a cache coherent NUMA (ccNUMA). ccNUMA provides the same hardware interface and can run all software designed for UMA architectures. This gives the illusion that ccNUMA will provide the same performance for such unmodified UMA (Uniform Memory Access) software. Researcher and software developers accepted the challenge and started to adapt current software and algorithms to take full advantages of NUMA architectures.

Some of the most important software to adapt include operating systems, system libraries, middleware, and database management systems. In this section, we will focus on proposals in the database and transactional memory fields as they share many properties and they are most related to the topics discussed by this dissertation.

2.3.1 Database

Current database management systems perform badly on multicore machines especially NUMA-based machines [88, 47, 56, 76]. In Multimed [88], authors showed that treating multicore machines as a distributed system performs much better. In their evaluation study, deploying multiple replicated instances of the database engine on the same machine performs better that deploying only a single instance that uses all available cores. [76] reached a similar conclusion: shared-nothing deployments perform better than cooperative ones.

In [56], they showed that allocating memory based on data partitioning, and grouping worker threads improve the performance. This new configuration exploits the locality features of NUMA architectures.

2.3.2 Transactional Memory

There are proposals that targeted eliminating centralized timestamp bottleneck. In [80], the timestamp is replaced by a physical (hardware) clock or a set of synchronized physical clocks. In [85], the same idea of exploiting a hardware clock instead of a global timestamp is explored. They proposed an algorithm that leverages the x86 cycle counter. Algorithms based on physical clocks are not expected to scale well as the hardware itself cannot keep a large number of physical clocks synchronized without paying a significant overhead.

In [9], they proposed Adaptive Versioning (AV) which uses a software predictor to expect the probability of conflict among transactions. Based on that, they select between TL2-GV4 [35], when conflict chance is high, and TLC [12], in low conflict scenarios. Although the idea looks appealing, the performance results is limited to 16 threads and show that AV matches the performance of TL2 at high contention levels, and TLC at low contention levels.

In SkySTM [59], authors presented a scalable STM algorithm that is privatization safe. SkySTM is based on semi-visible reads that is implemented using Scalable NonZero Indicator (SNZI) [42]. Semi-visible reads indicates the existence of concurrent readers without knowing which are those readers. In addition, with visible/semi-visible reads, there is no need for maintaining a global timestamp. SkySTM results shows a good scalability, but compared to TL2-GV6 [35], it is slower. This is mainly due to the fact that SkySTM is privatization safe while TL2 is not. Nemo is not privatization safe as it focuses on achieving the maximum performance possible under scalable workloads.

In TrC-MC [23], TLC algorithm is extended. First, they proposed a NUMA-zone level cache (zone partitioning) similar to Nemo-Vector's vector-clocks. Second, they used the timestamp extension mechanism which revalidates the read-set again to confirm that the conflict is indeed a real conflict before aborting the transaction. While designing Nemo, we tried timestamp extension mechanism and it showed a performance degradation although it reduced the number of false aborts. As a result, it represents a technique orthogonal and

applicable to Nemo.

In [63], a NUMA-aware TM is introduced. The basic idea is to change the conflict detector such that it becomes latency-based. It uses an eager conflict detection policy for intra-NUMA zone transactions, and uses a lazy policy for inter-NUMA zones ones. In addition, they deployed a conflict prevention mechanism to reduce conflicts probability. Results show an improved performance but limited scalability.

In [67], they targeted the same problem of Nemo focusing on locality awareness. From the available details, they also used a timestamp per cluster.

In Lock Cohorting [36], a mechanism to convert different types of locks into NUMA-aware locks is proposed. Results showed a significant performance improvement. We plugged this NUMA-aware lock into NOrec [29] algorithm but results were not improved significantly because NOrec has a commit serialization bottleneck. In [18], the lock cohorting idea is extended to reader-writer locks.

Disjoint-Access Parallelism

An important property that can enable the achievement of scalability in Transactional Memory (TM) is Disjoint-Access Parallelism (DAP) [55]. The DAP property also works very well given the NUMA architecture properties, which encourages to limit inter-NUMA zones communication as much as possible in order to achieve high performance. By definition, DAP only allows conflicting transactions to share data/meta-data. Thus, DAP-TM algorithms provide a good scalability on NUMA architectures [31]. Examples include TLC [12], DSTM [50], PermiSTM [10], and [74, 75].

TLC has the most practical implementation among them. The idea of TLC can be applied to timestamp based STM (e.g., TL2 [35], TinySTM [43]). The idea is to remove the global timestamp and replace it with a thread-local timestamp in each thread. In addition, each thread keeps a thread-local cache of other threads' timestamps. This cache is only updated when a conflict in timestamps is detected. Each object has an associated versioned lock. The lock's version includes both the writing transaction's ID and its timestamp at the time of writing. When a thread reads an object with a timestamp larger than its local cache, it aborts and updates the local cache. TLC suffers from a large number of false aborts due to outdated cached copy of timestamp and can only work well under low levels of contention. The design of Nemo shares some of the TLC principles.

Chapter 3

Background

3.1 Parallel Programming

Amdahl's law [4] 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 [46, 5, 91, 24, 22] 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 [89, 52, 92, 79, 57, 49, 16, 73, 90, 35] 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 [62, 60, 77, 30, 71, 94] 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 [48].

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.

3.2.2 Version Control

Version control is the process of managing a transaction's writes during execution. Two types of version control techniques have been studied: *eager versioning* and *lazy versioning* [71]. In eager versioning, a transaction writes directly to the memory (i.e., in-place update) for each object that it modifies, while keeping the old value in an *undo log*. If the transaction aborts, then the old value is restored from the undo log. Eager versioning requires *eager conflict detection*. Otherwise, isolation cannot be maintained and intermediate changes will be visible to other concurrent transactions.

In lazy versioning, a transaction's writes are buffered in a transaction-local write buffer, sometimes called a *redo log*. During a successful commit, values in the write buffer are written back to the memory. In this approach, a transaction's reads need to check if the write buffer has the object before reading the object's value from the memory. If the data is not found in the write buffer, then the object's value is retrieved from the memory. 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 [71]. 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 Transactional Memory Algorithms

In this section, we briefly go through the details of the state-of-the-art TM algorithms that we used as competitors in our evaluation studies or we relate most in our discussions.

3.3.1 TL2

The TL2 [35] algorithm uses lazy versioning and lazy validation. It relies on a shared global timestamp and a global lock table (a table of versioned write-locks). The global timestamp is used to determine the chronological relation between transactions. The lock table is used to lock objects and to store the timestamp at the time of writing.

Each transaction has the following meta-data: read-set, write-set (buffer), and starting time. When a transaction begins, the starting time is set to the global timestamp value. Transactional writes add the written object and the new value to the local write-set. Transactional reads check first if the object exits in the write-set, and return it if so. Otherwise, they confirm that the read object is not locked and its associated version is less than or equal to the transaction's starting time. Otherwise, the transaction aborts. The last step of the read operation is to add the object to the read-set.

At commit time, if the transaction is read-only, then the commit is done immediately. Otherwise, all write-set's objects are locked. If any lock acquisition fails, then the transaction aborts. After that, the read-set's objects are validated to confirm that they are not locked by another transaction and their associated versions are still less than the transaction's starting time. Finally, the global timestamp is incremented. The incremented value is used to set the new version of the locks before releasing them.

TL2 has been subsequently enhanced by providing other versions (i.e., GV4 [35] and GV5 [35], and GV6 [35]) to reduce the contention on the global timestamp, which is clearly a scalability bottleneck.

- GV4 uses the *pass-on-fail* policy. If the atomic compare-and-swap (CAS) operation used to increment the global timestamp fails, then it is not retried. This is because another transaction already did the increment. In addition, that transaction must be disjoint from the current transaction since both have all their write-set's objects locked, thus there cannot be any overlap.
- GV5 updates the global timestamp only when a conflict is detected. Each thread locally increments the global timestamp in the commit operation without updating the global timestamp. When a transaction accesses an object with a version greater than the global timestamp, it aborts and updates the global timestamp with that version. GV5 can cause false aborts even when only a single transaction is running in the system if that transaction accesses the same objects frequently.
- GV6 is a mix of GV4 and GV5 where GV4 is used with probability 1/32, and GV5 is used otherwise.

3.3.2 RingSTM

The RingSTM [90] algorithm uses lazy versioning and lazy validation. Its core innovation is in using a compact representation for the read-set and write-set. In fact, it takes advantage of *Bloom filters* [15] to summarize all reads (read-signature) and all writes (write-signature). Other meta-data includes a write-buffer and the transaction's starting time. As a global meta-data, there is the *ring* data structure (circular buffer) which includes all committed transactions' write-signatures, and the global timestamp which represents the last used index in the ring.

When a transaction begins, the starting time is set to the global timestamp value. A transactional write adds the object to the write-signature and its value to the write-buffer. A transactional read checks first if the object exits in the write-set, and returns it if so. Otherwise, the object is added to the read-signature. Then, the transaction confirms that the read-signature is still valid if the global timestamp is changed since the last successful validation. If the validation succeeds, the transaction starting time is extended to the current global timestamp; otherwise, the transaction aborts.

The read-signature validation is done by intersecting the read-signature against each new committed transaction's write-signature in the ring that is committed after the last successful validation. If any intersection results in a non-zero Bloom filter, then the validation fails.

At commit time, if the transaction is read-only, then the commit can be done without performing any additional task. That task is rather needed for write-transaction, and it consists

of validating the read-signature again if the global timestamp is changed since the last validation. If it is still valid, the global timestamp is atomically incremented, and this increment has a side effect of reserving a slot in the ring for storing the transaction's write-signature. Then, the write-buffer is written back to the memory and the write-signature is copied into the ring. During the write-back and the ring-update operations, other transactions cannot access the ring for validation, thus they wait until the access to the ring is allowed again.

Bloom filters has the advantage of having an $O(1)$ complexity of all its operations (add, contains, intersection), and constant memory space. But, it has the drawback of false positives. False positives cause unnecessary aborts as they represent false conflicts.

3.3.3 NOrec

The NOrec [29] algorithm uses lazy versioning and lazy validation. It is characterized by removing the need for ownership records (Orecs). Instead, it uses a single lock to serialize the transactions' commit phase and a value-based validation to validate the read-set. This algorithm has a single global meta-data, which is the global timestamp that acts also as a global lock. As a local meta-data, there is the read-set, write-set and the transaction starting time.

When a transaction begins, the starting time is set to the global timestamp value. A transactional write adds the object and its new value to the write-set. A transactional read checks first if the object exits in the write-set, and return it if so. Otherwise, before reading the object's value, the transaction confirms that the global timestamp did not change since last successful validation. If the global timestamp is changed, then the read-set is revalidated by confirming that the objects' values currently committed in memory still match the valued stored in the read-set. If the read-set is still valid, then the object and its value are added to the read-set.

At commit time, if the transaction is read-only, then the commit is done immediately because there is no actual commit phase. Otherwise, the transaction's read-set is revalidated if the global timestamp is changed since last successful validation. If the validation is successful, then the timestamp lock is acquired using a CAS operation. It CAS operation failed, then a revalidation is needed before retrying to acquire the lock. After successfully acquiring the lock, the write-set is written back to the memory and the lock is released by incrementing the global timestamp.

NOrec is a very simple and efficient algorithm for a low number of threads, but it suffers from scalability problems as the number of threads increases given the serial execution of transactions' commit phases.

3.3.4 Reduced Hardware NOrec (RH-NOrec)

The RH-NOrec [69] algorithm is a hybrid TM algorithm that extends NOrec algorithm using the reduced-hardware technique for HTM-STM communication [68]. The basic idea of reduced-hardware technique is to execute the commit phase of an STM transaction as a hardware transaction. In RH-NOrec, another hardware transaction is used also to execute the read-prefix of the transaction. Using the two hardware transactions in the slow path allows for delaying the reading of the global timestamp to the end of the read-prefix HTM. It also allows the fast path to read the global timestamp at the very end. In addition, no instrumentation is needed in the fast path.

In details, fast-path HTM proceeds normally without instrumentation, before it commits, it updates the global timestamp (if it is not locked) which notifies other slow-path transactions that a change occurred. Slow-path transactions start with a prefix-HTM transaction that executes the maximum possible reads. Before the prefix-HTM transaction commits, it reads the global timestamp and if it is locked, the transaction aborts. Before starting the hardware commit phase transaction, the global timestamp lock is acquire. Then, the commit phase is done inside the HTM. Finally the HTM commits before releasing the global timestamp.

3.3.5 TLC

The primary goal of the TLC [12] algorithm is to eliminate the need for a global timestamp while maintaining the same correctness level. The idea of TLC can be applied to timestamp based STM (e.g., TL2 [35], and TinySTM [43]). Roughly, they propose to remove the global timestamp and replace it with a thread-local timestamp in each thread. In addition, each thread keeps a thread-local cache of other threads' timestamps. This cache is only updated when a conflict in timestamps is detected. Each object has an associated versioned lock. The lock's version includes both the writing transaction's ID and its timestamp at the time of writing. When a thread reads an object with a timestamp larger than its local cache, it aborts and updates the local cache.

TLC suffers from a large number of false aborts due to outdated cached copy of timestamp and can only work well under workloads with low contention level.

3.4 Intel's Haswell HTM processor

The current Intel's HTM implementation of Haswell processor, also called Intel Haswell Restricted Transactional Memory (RTM) [78], 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 write operations and conflict detection [72]. 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). Transactional reads can go beyond L1 cache size upto 4 MB [72] using a special hardware buffer for transactional reads. This hardware buffer keeps track of read cache lines that are evicted from L1 cache.

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 written 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 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 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.

Intel HTM provides another programming interface which is Hardware Lock Elision (HLE). HLE is an instruction-prefix-based that is added to locking instructions. Thus, it automatically tries to elude the lock by optimistically executing the critical section without acquiring the lock. When the optimistic speculative execution fails, the critical section is re-executed after acquiring the lock. This interface is transparent to the programmer as it is the same as locks-based programming. Thus, it is compatible with legacy code.

Chapter 4

Part-HTM

4.1 Problem Statement

Transactional Memory (TM) [48, 51] is one of the most attractive recent innovations in the area of concurrent and transactional applications. TM is a support that programmers can exploit while developing parallel applications so that the hard problem of synchronizing different threads, which operate on 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 [29, 28]. 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) [29], which can be executed without any transactional hardware support, *hardware* (HTM) [48, 24], which exploits specific hardware facilities, and *hybrid* [29], which mixes HTM and STM.

Very recently two events confirmed TM as a practical alternative to the manual implementation of thread synchronization: first, *GCC* – the famous GNU compiler, embedded interfaces for executing atomic blocks since its version 4.7; second, Intel released to the customer market the *Haswell* processor equipped with *Transactional Synchronization Extensions* (TSX) [78], 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 because the conflict resolution is inherently provided by the hardware cache-coherence protocol; 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 eventual commit of an HTM transaction is guaranteed through a software execution defined by the programmer (called *fallback path*). The default fallback path consists of executing the transaction protected by a single global lock (called GL-software path). In addition, there are

[1]IBM also released the Power8 [17] processor with best-effort HTM support.

other proposals that take the choice of falling back to a pure STM path [28], as well as to a hybrid-HTM scheme [68, 19].

Leveraging the experience learnt from recent papers on HTM [37, 19, 28], three reasons that force a transaction to abort have been identified: *conflict*, *capacity*, and *other*. Conflict failure occurs when two transactions 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 is higher than the maximum allowed; and any extra hardware intervention, including interrupts, is also a cause of abort (see Section 4.1.1 for more details).

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 fallback path [28, 19, 68]; *ii)* tune the number of retries a transaction running in hardware has to accomplish before falling back to the software path [37]; and *iii)* modify the underlying hardware support for allowing special instructions so that conflicts can be solved more effectively [5, 24].

Despite this body of work, one of the main unsolved problems 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 require non-trivial execution time even accessing few objects and thus they are aborted due to a timer interrupt (which triggers the actions of the OS scheduler); or those transactions accessing several objects, such that the problem of exceeding the cache size arises (*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 required to commit, are not available. We name this superset as *resource* failures.

None of the past works target this class of aborted transactions and we turn this observation into our core 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 those that require the execution of irrevocable operations (e.g., system calls).

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 (or from committing after having accessed) those committed (but still locked) objects.

This framework is designed to be 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 at 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 better than 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 due to 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 the conflicts resolution 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 and by still exploiting the effectiveness of HTM (leveraging sub-HTM transactions); and it represents the best trade-off between STM and HTM transactions.

Opacity [45] is the reference correctness criterion for TM implementations because it avoids any inconsistency during the execution, independently from the final transaction outcome (either commit or abort). However, ensuring opacity in Part-htm is challenging because its overhead could nullify Part-htm's benefits. While acknowledging the importance of an opaque hybrid-TM protocol, in this dissertation we present two versions of Part-htm. One aims at obtaining the best performance by relaxing opacity in favor of serializability [13], the well-known consistency criterion for online transaction processing, and by relying on the HTM protection mechanism (i.e., sandboxing), which protects from faulty computations (e.g., division by zero). In the second version, we enriched Part-htm for ensuring opacity but, at the same time, we present a set of innovations (e.g., *address-embedded* write locks) for reducing the transaction's memory footprint so that the overhead is kept limited (less than the achievable gain).

We implemented Part-htm and assessed its effectiveness through an extensive evaluation study (Section 4.6) including a micro-benchmark, a data structure, the STAMP suite [70], and EigenBench [53]. As competitors, we selected a pure HTM with GL-software path as a fallback, two state-of-the-art STM protocols and a recent HybridTM. Results confirmed the effectiveness of Part-htm. It is the best in almost all the tested cases, except those where HTM outperforms all (and therefore no competitor can perform better than that). In these workloads, Part-htm still represents the best among other STM and HybridTM alternatives. The combination of these two contributions gives to Part-htm the unique characteristic of being an effective trade-off, independently from the application workload.

Part-htm has been designed and evaluated using the current Intel Haswell processor (i7-4770). In Section 4.4, we describe how Part-htm's approach can take advantage of the upcoming best-effort HTM processor (i.e., IBM Power8 [17]) with the new support for suspending/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 HTM implementation of the Haswell processor, also called Intel Haswell Restricted Transactional Memory (RTM) [78], 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 conflicts, HTM transactions can be aborted for other reasons. Any cache-line eviction (e.g., due to cache-associativity) of written memory locations causes the transaction to abort (however there is a specialized buffer for handling the eviction of a memory location previously read, but not written). This means that write operations of hardware transactions are limited in space by the size of the L1 cache. However, read operations can go beyond the L1 cache capacity by exploiting the L2 cache. Also, any hardware interrupt, including the interrupts from timers, forces HTM transactions to abort. As stated before, we name the union of these two causes as *resource* limitation and in this dissertation 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 the 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 often execute 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 HTM.

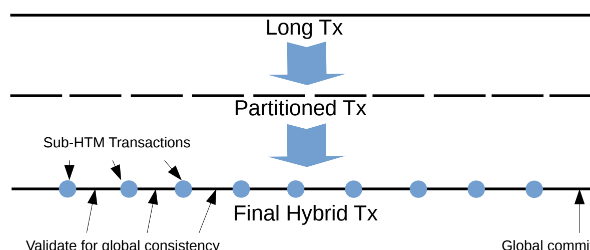


Figure 4.1: PART-HTM's Basic Idea.

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 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 a *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 at 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 until 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/written objects during the rest of T's execution, therefore also T's correctness becomes hard to enforce.

All these problems can be trivially 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 can easily lead to HTM losing its effectiveness and, consequently, lead to poor performance. In the following we point out some of these reasons:

- STM 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; and
- the HTM monitors any memory access within the transaction, including those on the meta-data or local variables, which takes the flexibility for implementing smart contention policies away from the programmer.

Part-htm faces the challenge of how to exploit the efficiency of sub-HTM transactions, which write in-place to the shared memory, by minimizing the overhead of the instrumentation needed for maintaining the isolation and correctness of global transactions. Such a system does not only overcome the limitation of aborting transactions that cannot fit in HTM due to limited resources, but it also performs similar to HTM in HTM's favorable workloads, and better than STM in scenarios where HTM generally behaves badly.

Given that HTM transactions commit directly to the shared memory and Part-htm always executes transactions using HTM (except when the GL-software path is invoked), we opt for using an *eager* approach. We recall that, 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, the state of the shared memory is already updated, and if not, the transaction is aborted and the undo-log is used to restore the old values.

To also cope with transactions that do not fail for resource limitations, 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 resource limitations, 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 and committing or overwriting values created by T^x_y before T^x is committed; and *3)* executing T^x in a way the transaction observes a consistent state of the 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 fallback 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 [29]; rather we rely only on cache-aligned Bloom-filter-based meta-data (just Bloom-filter hereafter) to keep track of read/write accesses. In our solution we refer to a Bloom-filter [15] as an array of bits where the information (addresses in our case) is hashed to a single entry in the array (i.e., single bit). Just before committing, a sub-HTM transaction updates a shared Bloom-filter for notifying its written objects, so that no other transaction can access them. We recall 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*).

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/written objects survive even after the commit of a sub-HTM transaction, allowing the framework to check the validity of the global transaction at any time.

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 non-transactional computation can be executed as a part of the software framework, whereas only the transactional part executes as sub-HTM transaction.

As mentioned before, in this dissertation we provide also a version of Part-htm (called Part-htm-o) that guarantees opacity by introducing some (but limited) overheads. We can summarize them by two additional checks that a Part-htm-o sub-HTM transaction performs to promptly detect inconsistent executions (i.e., before performing any memory access). Unfortunately, any validations involving read and written objects require storing them into per-transaction meta-data, which consume the memory available for supporting HTM transactions. Also, those meta-data will be shared among threads, thus increasing the amount of aborts significantly.

Part-htm-o takes into account these concerns by adopting the following solution. First, once an object is accessed by a sub-HTM transaction, the existence of a write lock is immediately detected. In order to minimize the impact on the memory footprint, we introduce the *address-embedded* write locks, which are locks that do not use additional memory location, whereas they are implemented by "stealing" the last bits from the accessed address. This prevents any false conflicts on the shared write locks set. Second, a sub-HTM transaction is immediately aborted once a global transaction commits (which is detected leveraging the HTM conflict management). This condition, which appears to be very conservative, allows the execution of a software validation, which can assess if the just-committed transaction conflicts with the on-going global transaction. If that is not the case, then the aborted sub-HTM transaction is immediately restarted, thus saving the previous computation of the global transaction.

Before we proceed with Part-htm's algorithmic details in Section 4.3, a comparison between the executions of a pure HTM, a lazy STM (e.g., [29, 90, 40, 35]), and Part-htm is reported in Figure 4.2. In an HTM transaction, a group of reads and writes are wrapped in between a

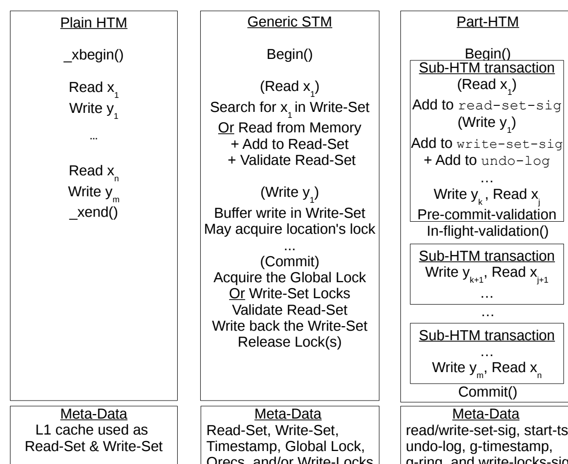


Figure 4.2: Comparison between HTM, STM, and PART-HTM.

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

In the following we refer the objects committed by a sub-HTM transaction whose global transaction is still executing as *non-visible*. 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. In the following subsections we refer to specific pseudo-code line using the notation [Line X].

4.3.1 Protocol Meta-data

PART-HTM uses meta-data; some of them are local, thus visible by only one transaction, others are shared by all transactions. In order to reduce their size, most of them are Bloom-filters (i.e., a compact representation). 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 false conflict, in our implementation Bloom-Filters are bit-arrays of 2048 bits (4 cache-lines) with a single hash function. If two

```
First HTM trial
tx_begin()
1. if (GLock) _xabort();
tx_read(addr)
2. read_sig.add(addr);
3. return *addr;
tx_write(addr, val)
4. write_sig.add(addr);
5. *addr = val;
htm_pre_commit()
6. if (write_locks ∩ write_sig
      || write_locks ∩ read_sig)
7.        _xabort();
8. ts = ++timestamp % RING_SIZE;
9. ring[ts] = write_sig;
10. _xend();
htm_post_commit()*
11. write_sig.clear();
12. read_sig.clear();
Sub-HTM
tx_read(addr)
13. read_sig.add(addr);
14. return *addr;
tx_write(addr, val)
15. undo_log.add(addr, *addr);
16. write_sig.add(addr);
17. *addr = val;
htm_pre_commit()
18. others_locks = (write_locks - agg_write_sig);
19. if (others_locks ∩ write_sig
        || others_locks ∩ read_sig)
20.       _xabort();
21. write_locks ∪= write_sig;
22. _xend();
Part-HTM
tx_begin()*
23. while (Glock) PAUSE();
24. atomic_inc(active_tx);
25. if (Glock) tx_abort();
26. start_time = timestamp;
in_flight_validation()*
27. ts = timestamp;
28. if (ts != start_time)
29.   for (i=ts; i >= start_time + 1; i--)
30.      if (ring[i % RING_SIZE] ∩ read_sig)
31.         tx_abort();
32.   if (timestamp > start_time + RING_SIZE)
33.      tx_abort(); //Abort at ring rollover
34.   start_time = ts;
htm_post_commit()*
35. agg_write_sig ∪= write_sig;
36. write_sig.clear();
tx_commit()*
37. if (is_read_only)
38.     atomic_dec(active_tx);
39.     return;
//The following two lines are atomic
40. ts = atomic_inc(timestamp) % RING_SIZE;
41. ring[ts] = agg_write_sig;
//The following line is atomic also
42. write_locks = write_locks - agg_write_sig;
43. agg_write_sig.clear();
44. read_sig.clear();
45. atomic_dec(active_tx);
tx_abort()*
46. undo_log.undo();
47. write_locks = write_locks - agg_write_sig;
48. agg_write_sig.clear();
49. read_sig.clear();
50. atomic_dec(active_tx);
51. exp_backoff();
52. restart_tx();
Acquire GL*
53. while (!CAS(GLock, 0, 1));
54. while (active_tx);//Wait for active tx
```

Figure 4.3: PART-HTM's pseudo-code. Procedures marked as * are executed in software.

transactions update different bits, they will not necessarily conflict on the same cache-line, thus saving an abort due to a *false* memory conflict.

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. All transactional threads share:

- *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 signatures.
- *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 against committed transactions, in a similar way as proposed in RingSTM [90].

4.3.2 Begin Operation and Partitioning

PART-HTM processes transactions at the beginning as HTM transactions (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 resource limitation, then the software framework splits the transaction into multiple sub-HTM transactions. The splitting process does not constitute the main contribution of the dissertation because there are several efficient policies that can be applied. Examples include those using compiler supports, such as [3, 2], or techniques that estimate the expected usage of cache-lines so that they can propose an initial partitioning.

In our prototype, we partition the application manually. Each transaction is written in three versions: one for the first-trial HTM; one with partitions; and one uninstrumented for the GL-software path. Partitions are static and determined based on profiler analysis. This analysis splits transactions into multiple basic blocks, and measures the size of accessed shared objects and the duration of each basic block. A partition will be then composed of one or more basic blocks according to their capability of fitting HTM resource limitations. We also excluded basic blocks that access no shared objects from being executed in sub-HTM transactions.

When a transaction starts, it reads the *global-timestamp* and stores it as the *starting-timestamp* [Line 26]. 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 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. A transaction that falls back to the GL-software path acquires the global lock and waits until the completion of all active transactions [Line 53-54].

A first-trial HTM transaction checks the global lock at the beginning such that it aborts if it is, or will be, acquired [Line 1]. A sub-HTM transaction is not allowed to start its execution until the global lock is not taken [Line 23].

4.3.3 Transactional Read and Write Operation

Read operations are always performed by HTM transactions, 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 3 or 14]. 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 order to detect if the accessed object is a non-visible version, the read memory location is recorded into the *read-set-signature* [Line 2 or 13]. This information is fundamental for preventing a sub-HTM transaction that accessed non-visible object from committing, thus guaranteeing the isolation from other transactions.

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 5 or 17]. The same considerations made for the read operations apply also for write operations. Thus any written locations is added to the transaction's *write-set-signature* [Line 4 or 16]. This information will be used by the HTM transaction before proceeding with the commit phase.

In addition to the above steps, two other important actions (that do not apply to first-trial HTM) must be taken into account. 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 15]. 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* [Line 21]. 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, we delay the update of the *write-locks-signature* at the end of the sub-HTM transaction so that false conflicts are minimized.

In practice, 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. Besides 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.

4.3.4 Validation, Commit and Abort

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

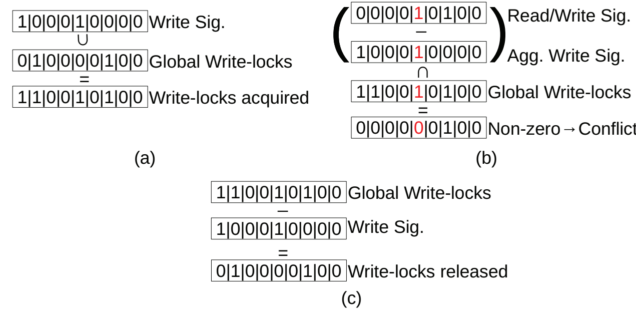


Figure 4.4: Acquiring write-locks (a). Detecting intermediate reads or potential overwrites (b). Releasing write-locks (c).

transaction (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 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 6 or 19]) 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 7 or 20].

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 18]. The aggregated *write-set-signature* is updated in software after each sub-HTM transaction [Line 35].

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 6 or 19] (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 (described in [32]) 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 [32], for example, 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), then the commit is called without executing the HTM-pre-commit-validation. PART-HTM-O (Section 4.3.5) solves such an issue by guaranteeing opacity [45].

In-flight-validation. This validation is performed by the software framework after the commit of every sub-HTM transaction in case some global transaction (including *first-trial* HTM) committed in the meanwhile, 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.

Assuming the scenario with two global transactions T^x and T^y, both having two sub-HTM transactions each. Let us assume that T^x_1 reads the value of object o and commits. Let us also assume that o is not locked at this time. After that, T^y_2 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^x_2's commit, but let us assume that the commit of T^y is scheduled before T^x_2's commit (in fact, T^y_2 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^x_2's *read-set-signature* and the *write-locks-signature* does not report any conflict on o, therefore T^x_2 can 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 27-34]. 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 29]. After a successful *in-flight-validation*, the transaction's *starting-timestamp* is advanced to the current *global-timestamp* [Line 34]. 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 (excect for PART-HTM-O where it is mandatory). 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 because detecting invalidated objects early in the execution avoids unnecessary computation, saves HTM resources, and makes the software framework's execution always consistent.

Commit. The commit of a transaction is straightforward. First-trial HTM transactions are committed in HTM, and added to the *global-ring* if not read-only [Line 8-10]. If the

```
First HTM trial
tx_begin()
1. if (GLock) _xabort();
tx_read(addr)
2. if (addr & 1) //Locked
     _xabort();
3. return *addr;
tx_write(addr, val)
4. if (addr & 1) //Locked
     _xabort();
5. write_sig.add(addr);
6. *addr = val;
htm_pre_commit()
7. ts = ++timestamp % RING_SIZE;
8. ring[ts] = write_sig;
9. _xend();
htm_post_commit()*
10. write_sig.clear();
Sub-HTM
tx_sub_begin()
11. if (start_time != timestamp)
12.    _xabort(TS_CHANGED);
not_self_lock(addr)
13. foreach (entry in undo_log)
14.    if (entry.addr == addr)
15.       return false;
16. return true;

tx_read(addr)
17. if ((addr & 1) && not_self_lock(addr))
18.    _xabort(CONFLICT); //Locked by others
19. read_sig.add(addr);
//Remove lock bit before dereferencing
20. return *(addr & ~1);
tx_write(addr, val)
21. if (addr & 1) //Locked by others or self?
22.   if(not_self_lock(addr)) _xabort(CONFLICT);
23.   else goto 27
24. undo_log.add(addr, *addr);
25. write_sig.add(addr);
26. addr = addr | 1; //Acquire lock
27. *(addr & ~1) = val;
Part-HTM
tx_begin()*
28. while (Glock) PAUSE();
29. atomic_inc(active_tx);
30. if (Glock) tx_abort();
31. start_time = timestamp;
in_flight_validation()*
32. ts = timestamp;
33. if (ts != start_time)
34.   for (i=ts; i >= start_time + 1; i--)
35.      if (ring[i % RING_SIZE] ∩ read_sig)
36.         tx_abort();
37.   if (timestamp > start_time + RING_SIZE)
38.      tx_abort(); //Abort at ring rollover
39.   start_time = ts;

tx_commit()*
40. if (is_read_only)
41.      atomic_dec(active_tx);
42.      return;
//The following two lines are atomic
43. ts = atomic_inc(timestamp) % RING_SIZE;
44. ring[ts] = write_sig;
45. foreach (entry in undo_log) //Unlock all
46.     entry.addr = entry.addr & ~1;
47. write_sig.clear();
48. read_sig.clear();
49. atomic_dec(active_tx);
tx_abort()*
50. undo_log.undo();
51. foreach (entry in undo_log) //Unlock all
52.     entry.addr = entry.addr & ~1;
53. write_sig.clear();
54. read_sig.clear();
55. atomic_dec(active_tx);
56. exp_backoff();
57. restart_tx();
Acquire GL*
58. while (!CAS(GLock, 0, 1));
59. while (active_tx);//Wait for active tx
Sub-HTM Abort Handler*
60. if (abort_code == TS_CHANGED)
61.     in_flight_validation(); //Valid? Abort?
62.     restart_sub_HTM(); //Still valid
63. else tx_abort();
```

Figure 4.5: Part-htm-o's pseudo-code. Procedures marked as * are executed in software.

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 37].

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 41] and increments the *global-timestamp* [Line 40], atomically. Finally, all transaction's write locks should be released [Line 42]. This operation is done by executing an atomic bitwise *XOR* between the transaction's *write-set-signature* and the global *write-locks-signature*, as shown in Figure 4.4(c).

Abort. 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 46]. After that, the transaction's owned write-locks are released from the global *write-locks-signature* [Line 47] and a retry is invoked after an exponential back-off time [Line 51-52]. 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, the transaction finally falls back

to the GL-software path.

4.3.5 Ensuring Opacity

Part-htm cannot guarantee opacity. This is because the consistency of the execution history is not verified encounter time but only before committing a sub-HTM transaction, as well as during the *in-flight-validation*. Roughly, the former validation checks if objects accessed during the current sub-HTM transaction were non-visible; the latter verifies that the memory snapshot observed by the global transaction is still consistent against all committed transactions. These validations do not prevent the transaction to perform a memory read if the object is non-visible or the global transaction's history is not valid anymore, whereas they "only" prevent the transaction to finally commit.

Two extensions are needed for making Part-htm opaque: *1)* once a locked object is accessed, the global transaction should be immediately aborted; *2)* no memory access should be performed if the snapshot observed by the sub-HTM transaction, as well as the global transaction, is not valid. Figure 4.5 shows the pseudo-code of Part-htm-o's core operations. In this sub-section, line numbers refer to Figure 4.5.

Encounter time lock detection. In principle, checking if an object is locked is straightforward because we could analyze the *write-locks-signature* just before performing the actual read. Unfortunately, the *write-locks-signature* is a global meta-data, which is updated anytime a sub-HTM transaction commits any object. As a result, reading it during an HTM transaction means being aborted anytime another sub-HTM transaction updates it, even if the accessed object is not the same and their entries in the *write-locks-signature* are different. Another solution is creating an external lock-table for storing locks. However, this solution has the same pitfalls as the *write-locks-signature* because they both rely on a hash-function.

Part-htm-o solves this problem by introducing the *address-embedded* locks, which is a novel technique never used in the HTM context, that embeds the information about the lock acquisition into the memory address of the shared object itself. With it, we assign the address of shared objects in a way such that they are always memory-aligned. If so, we set the least significant bit (meaningless because we know the object is always memory-aligned) to 1 (locked) or 0 (unlocked). With *address-embedded* locks we eliminate any false conflicts due to shared meta-data. In practice, when an object is accessed inside a sub-HTM transaction, the least significant bit of its address is checked and if a lock is found, the transaction is explicitly aborted [Line 2, 4, 17, 21].

The deployment of *address-embedded* locks requires a memory location for storing the actual memory-aligned address that points to the shared object. Although it does not generate any perceivable performance overhead, the implementation of this indirect addressing layer needs a modification of the memory allocation of the application (which is a downside). As an example, if the shared object is a primitive type (e.g., integer), we need to manage the

value of its pointer. This indirect addressing layer must be added.

In more details, we exploit the memory alignment of addresses, which allows the last bit to be manipulated arbitrarily without corrupting the address itself. If the application accesses a scalar X with address $addr(X)$, in order to change the last bit of $addr(X)$ we need an indirect reference to X ($wrap(X)$). This way, the value of $wrap(X)$ is $addr(X)$ but with the last bit ready to be used for locking. Therefore, the deployment of address-embedded-locks requires modifying the application (although simple) to use $wrap(X)$ rather than X. If X is a pointer, then no wrapper is needed and the lock is embedded in X itself. For instance, in a linked-list, nodes store pointers to other nodes (`Node* next`), thus we already have a container for modifying the addresses directly to embed the lock. Modifications are rather needed if there is a scalar (e.g., `int size`). If so, we wrap it with a pointer (`int* sizep = &size`) so it is only accessed indirectly via the wrapper (`*sizep`).

Consistent reads. Opacity requires that any memory access is performed only if it does not violate the consistency of the snapshot observed so far by the transaction. Part-htm does no provide this because there is no way to detect if an object read in a previous sub-HTM transaction becomes not valid while executing a subsequent sub-HTM transaction. As a consequence, a read operation can access to an object committed by a transaction whose history is not consistent with the global transaction. Part-htm allows this anomaly and aborts the global transaction once the sub-HTM transaction is already committed exploiting the *in-flight-validation*. A trivial solution for ensuring consistent reads is to validate all objects accessed before reading a new shared object, but this solution is unfeasible because it would generate several false conflicts and require maintaining all read objects, thus consuming resources.

Part-htm-o adopts a strategy that overcomes the above limitations. At the beginning of each sub-HTM transaction, the *global-timestamp* is compared against the transaction's *starting-timestamp* [Line 11-12]. The goal is to abort the sub-HTM transaction anytime a new global transaction commits. Once this happen, the *in-flight-validation* is called and, in case it succeeds, just the sub-HTM transaction is restarted [Line 60-62], otherwise the whole global transaction is aborted [Line 36]. Reading the *global-timestamp* allows the sub-HTM transaction to avoid any validation while executing because, once a new global transaction commits and it is added to the *global-ring*, the *global-timestamp* is changed and this forces the sub-HTM transaction to abort due to the hardware conflict detection. The combination of both the above extensions make the sub-HTM's *HTM-pre-commit-validation* useless in Part-htm-o because its goal is already provided earlier in the execution.

By guaranteeing opacity we prevent any step made by HTM transactions if the observed history is not consistent anymore. This proscribes the pathologies describe in [32].

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. 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 [13] as correctness criterion and PART-HTM-O ensures opacity [45]. We now show the first and then we extend the discussion to opacity.

Three types of conflicts can invalidate the transaction's execution: *write-after-read*, *read-after-write*, *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 detects a taken lock by intersecting the transaction's *write-set-signature* with the global *write-locks-signature*.

Following the above rules, a transaction starts the commit phase by having observed a state that is still valid. 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, as defined in [45]. As a partial fallback plan, the HTM provides a sandboxing feature, which eventually aborts misbehaving HTM transactions that generate infinite loops or erroneous computations. However, without guaranteeing Opacity, the protocol cannot prevent corner case situations where a sub-HTM transaction is committed skipping the pre-HTM validation.

PART-HTM-O addresses this problem by avoiding any memory operation in case *A)* the snapshot observed by the transaction is not consistent anymore, and *B)* if the memory access itself would break the consistency of the transaction.

(A) We ensure the point A by monitoring the *global-timestamp* as the first operation of a sub-HTM transaction. This way, if the *in-flight-validation* performed before the activation of a sub-HTM transaction missed some object committed just after the *in-flight-validation* or if some global transaction commits while a sub-HTM transaction is executing, then the *global-timestamp* is changed and any HTM transaction is aborted and forced to perform a validation of all accessed objects.

(B) If a sub-HTM transaction accesses an object already locked (if the object becomes locked after the access, then the HTM will detect the conflict), then before to finalized the access the HTM transaction is explicitly aborted by leveraging the *address-embedded* write locks.

4.6 Evaluation

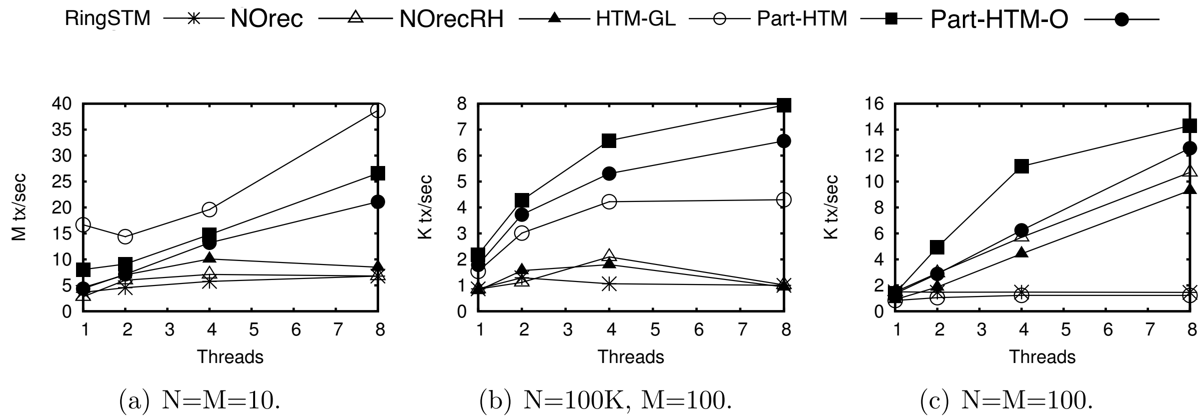
Part-htm has been implemented in C++. To conduct a comprehensive evaluation, we used four benchmarks: *N-reads M-write*, a configurable application provided by RSTM [66]; the linked-list data structure; STAMP [70] (v0.9.10), the popular suite of applications used for evaluating STM- and HTM-related concurrency controls; and EigenBench [53], a customizable TM benchmark. Very recently, a new version of STAMP [84] has been made available. It is implemented using the new C++ transactional constructs so that any transactional instrumentation is handled by the compiler itself. Part-htm requires at least one additional construct exposed by the compiler to define the boundaries of a sub-HTM transaction and its context. Extending Part-htm to comply with that is not the focus of the dissertation. However, to address this issue we re-implemented Labyrinth (the one changed most) according to the new specification as in [84] and the results are reported in Figure 4.9.

As competitors, we included two state-of-the-art STM protocols, RingSTM [90] and NOrec [29]; one Hybrid TM, Reduced Hardware NOrec (NOrecRH) [68]; and one HTM with the GL-software path as fallback (HTM-GL). Also, the ring used by RingSTM and Part-htm have the same size and signature. NOrecRH and HTM-GL retry a transaction 5 times as HTM before falling back to the software path. All are implemented such that they do not suffer from the lemming effect [33]. As suggested in [33], a transaction does not retry until the global lock is not acquired. In this evaluation study we used the Intel Haswell Core i7-4770 processor and GCC 4.8.2. All data points are the average of 5 repeated execution. To show the viability of using the *address-embedded* write locks, we also included the performance of Part-htm-o in most of the used applications.

As a general comment of our evaluation, Part-htm represents the best solution in almost all the tested workloads, except for those where pure HTM transactions always commit. In these cases, outperforming HTM is impossible without additional hardware support, but our approach, thanks to the first-trial HTM transactions, does not pay a significant performance penalty.

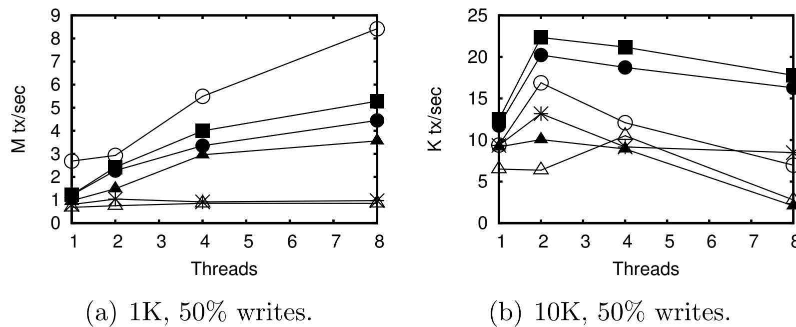
N-Reads M-Writes. In this benchmark each transaction reads N elements from one array and writes M to another. The benchmark can also be configured to access disjoint elements (i.e., no contention). We take advantage of this latter feature so that we can evaluate our approach in scenarios where the aborts due to non-false conflicts of HTM transactions are minimized.

Figure 4.6(a) shows the results of reading and writing 10 disjoint elements. In this experiment, few transactions are aborted for resource failure, thus almost all commit as HTM. As expected, HTM-GL has the best throughput, followed by Part-htm. This scenario is not the best case for Part-htm but still, thanks to the lightweight instrumentation of first-trial HTM transactions, it shows a slow-down limited to 45% over HTM-GL, whereas the best competitor (NOrecRH) is 91% slower than Part-htm. Interestingly, Part-htm-o is slightly slower than its non-opaque version due to the need of aborting a sub-HTM transac-



(a) N=M=10. (b) N=100K, M=100. (c) N=M=100.

Figure 4.6: Throughput using N-Reads M-Writes benchmark.



(a) 1K, 50% writes. (b) 10K, 50% writes.

Figure 4.7: Throughput using Linked-List.

tion once a global transaction commits. However this overhead is limited because, in case of no real conflict (as in this experiment), only the aborted sub-HTM transaction is restarted. On the other hand, the *address-embedded* write locks eliminate any false conflict due to the compact representation of the write locks set, thus regaining performance.

Figure 4.6(b) shows an experiment where 100k 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 [26], however most of HTM transactions fall back to the GL-software path. For this reason, the benefit of partitioning and committing through sub-HTM transactions, which are much faster than falling back to the GL-software path, is evident. 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. Part-htm gains around 20% over Part-htm-o.

In Figure 4.6(c), transactions perform one read on an object and then it does some floating

point operations before writing its new value back to the destination array. This sequence is repeated 100 times on different objects. 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. Part-htm-o follows the same trend line as Part-htm but with a small performance gap as showed before.

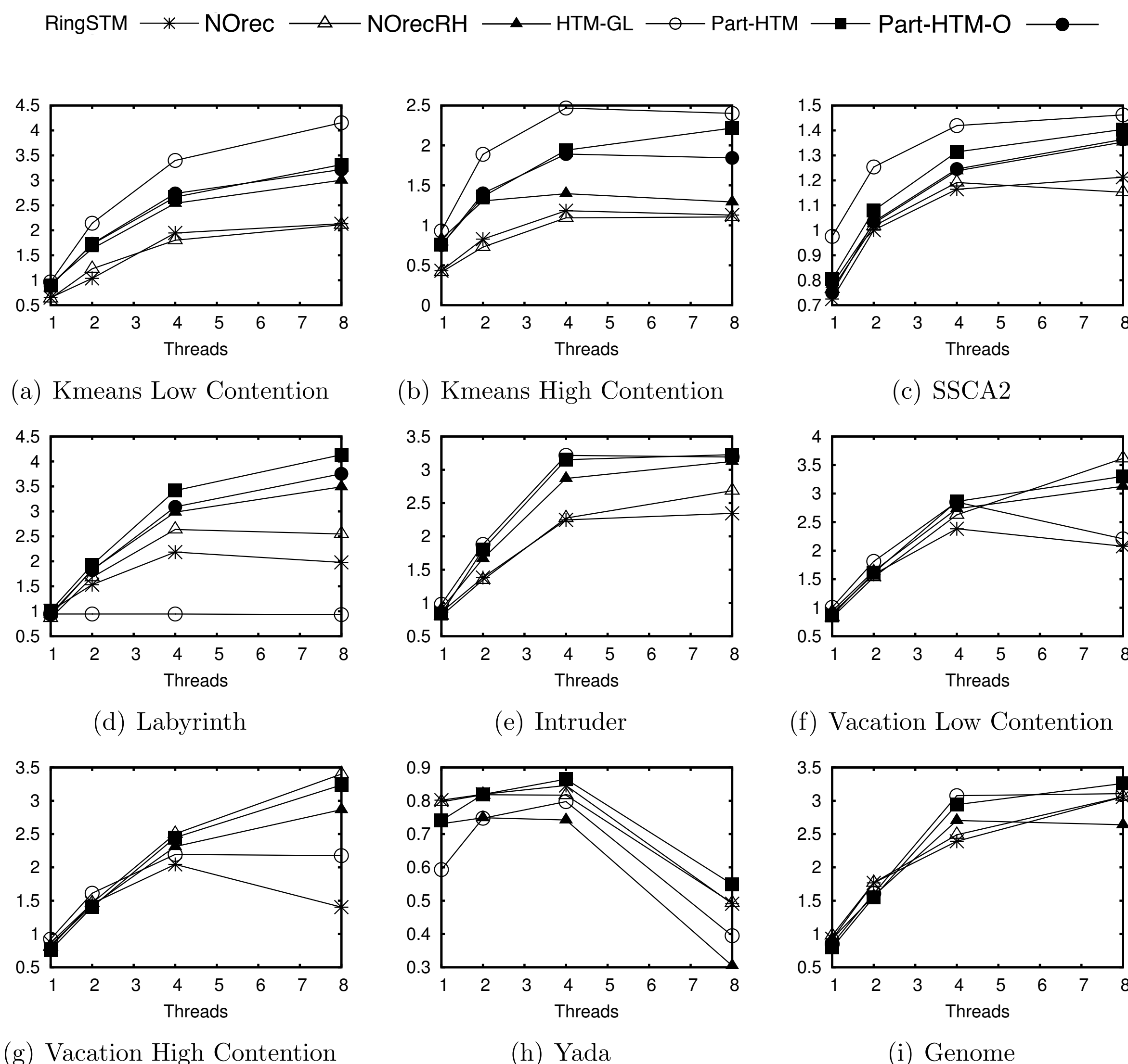
Linked-List. In this benchmark, we do operations on a linked list. We change its size, and the percentage of write operations (insert and remove) against read operations (contains). Linked list transactions traverse the list from the beginning until the requested element. This increases the contention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

Figure 4.7(a) shows the results of a 1K elements linked list using 50% of write operations. Linked list operations do several memory reads to traverse the data structure, and some writes. Thus, almost all transactions commit in hardware and HTM-GL has the best throughput. However, following the same trend as Figure 4.6(a), Part-htm places its performance closer to HTM-GL.

Figure 4.7(b) shows a larger linked list with 10K elements. Here, most of the transactions fail in hardware for resource failures. As for the case in Figure 4.6(c), Part-htm's throughput is the best as sub-HTM transactions pay a limited instrumentation cost and fast execution in hardware. Part-htm gains up to 74% over HTM-GL.

STAMP. Figure 4.8 shows the results of 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 effectiveness of Part-htm's design because it is the best in almost all cases, and the closest to the best competitors when HTM is the best. All data points report the achieved speed-up with respect to the sequential execution of the application.

Kmeans (Figure 4.8(b) and 4.8(a)), Vacation low-contention (Figure 4.8(f)), SCAA2 (Figure 4.8(c)), Intruder (Figure 4.8(e)), and Genome (Figure 4.8(i)) 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. Interestingly, SCAA2 and Kmeans show the instrumentation overhead of Part-htm while executing with only one thread.



(a) Kmeans Low Contention (b) Kmeans High Contention (c) SSCA2

(d) Labyrinth (e) Intruder (f) Vacation Low Contention

(g) Vacation High Contention (h) Yada (i) Genome

Figure 4.8: Speed-up over sequential (non-transactional) execution using applications of STAMP Benchmark.

On the other hand, applications like Labyrinth (Figure 4.8(d) and Table 4.1) and Yada (Figure 4.8(h)) are suited more for STM protocols than HTM. That is because more than half of the generated transactions in Labyrinth are large and long (thus HTM cannot be efficiently exploited), but they also rarely conflict with each other. As a result, NOrecRH and NOrec perform worse than, but closer to, Part-htm. HTM-GL is the worst. We also observe a 10% of gap between Part-htm and Part-htm-o. This gap is basically the cost of performing the *in-flight-validation* once a global transaction commits and a sub-HTM transaction is executing. Labyrinth is not characterized by short transactions, thus updates of the *global-timestamp* are not very frequent, and this helps to reduce the gap between Part-htm-o and Part-htm.

	% 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.

In Figure 4.8(f) and 4.8(g) we observe the impact of hyper-threading (thus reduce number of cache-lines available per executing thread). Moving from 4 to 8 threads, the performance of HTM-GL drops due to the increased capacity aborts. Figure 4.8(h) shows the results of Yada. This application has transactions that are long and large, generating a reasonable high contention level. Thus it represents a favorable workload for PART-HTM and the plot confirms it. We do not report the results using the Bayes application given its non-deterministic execution.

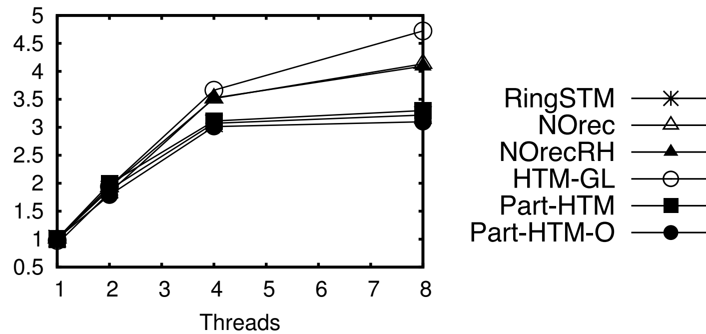
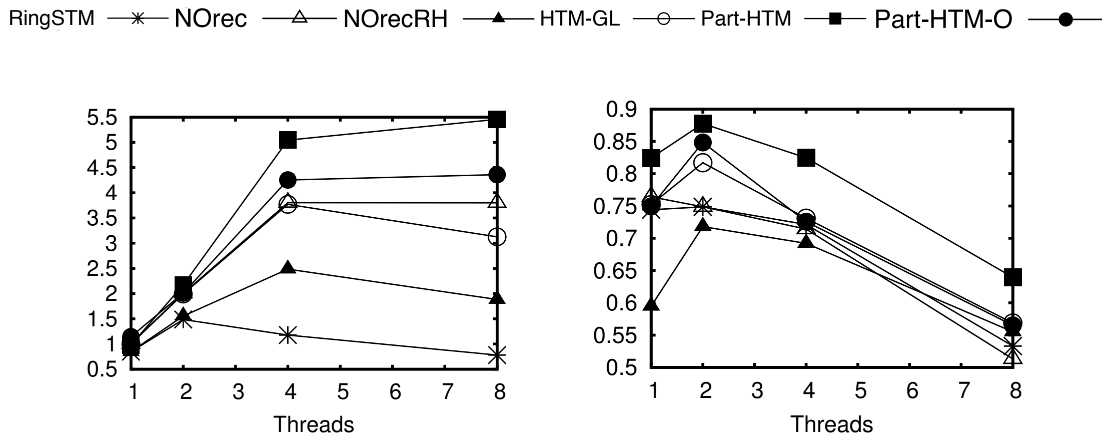


Figure 4.9: Speed-up over sequential (non-transactional) execution using Labyrinth as in [84].

Figure 4.9 shows the performance using a newer version of Labyrinth as introduced in [84]. We re-implemented this version according to the new specification as in [84]. This version of the benchmark produces transactions short in time (because the non-transactional computation has been moved from the transaction body) but that access several shared objects, thus they likely fail in HTM thus falling back to the GL-software path. However, HTM-GL still provides the best performance, even if very close to the others, because it falls back to the software path sooner than the original version of Labyrinth (Figure 4.8(d)). PART-HTM suffer from high percentage of false conflicts on the *write-locks-signature* due to the presence of several short HTM transactions that access several (likely different) objects.

EigenBench. EigenBench is a comprehensive benchmark, which can generate transactions with different properties. We used it to build a workload with 50% long and 50% small transactions, thus the latter will likely fit in HTM. A small transaction does 50 read and 5 write operations to an array of 1024 words while long transactions add non-transactional computation. Accesses are disjoint. Figure 4.10(a) plots the results. PART-HTM has the best performance as it executes the long transactions efficiently. PART-HTM-O follows with average overhead of 15%. Other competitors suffered with the long transactions.



(a) 50% Long transactions and 50% short ones

(b) High Contention

Figure 4.10: Speed-up over sequential (non-transactional) execution using EigenBench.

Figure 4.10(b) shows the results of EigenBench under high contention scenario. EigenBench is configured to access the shared `hot-array` of size 32K. Each transaction performs 10K reads and 100 writes with 50% repeated access Part-htm has the lowest overhead and the best performance due to detecting conflicts earlier than other techniques and encounter-time write locks. In addition, partitioning allows the execution of transaction in hardware, thus maximizing the exploitation of HTM.

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) [68]. 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

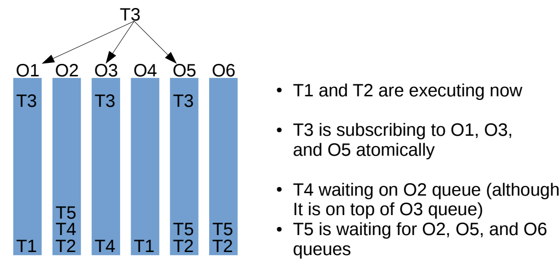


Figure 5.1: Scheduling transactions.

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.

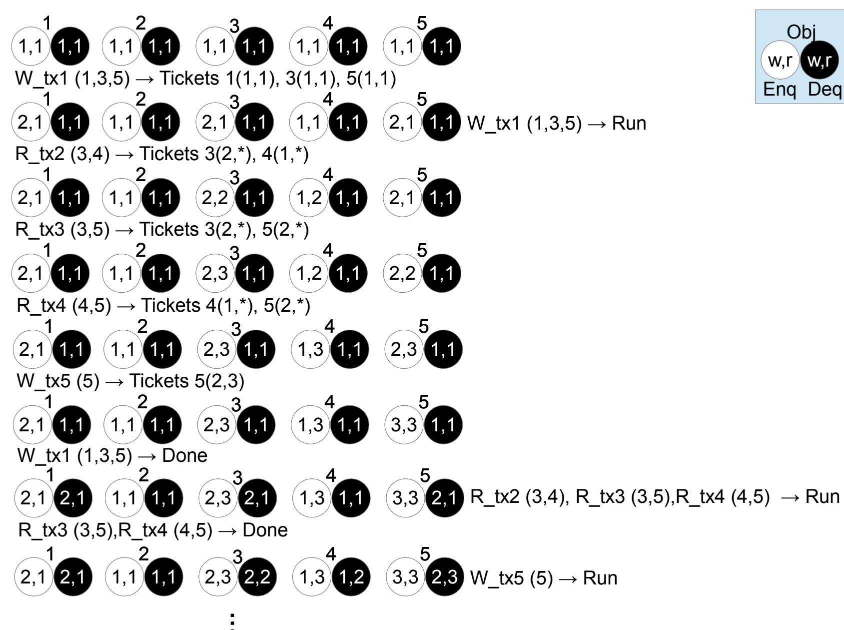


Figure 5.2: Readers-Writers ticketing technique.

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.

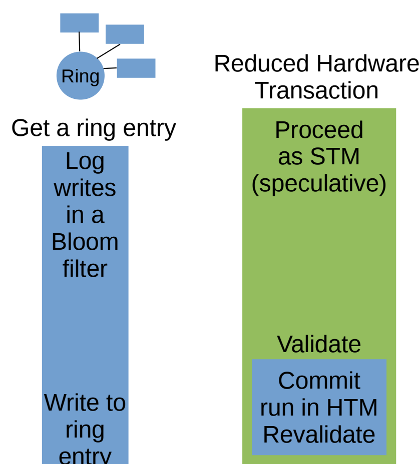


Figure 5.3: HTM-STM communication.

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) [68]). 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.

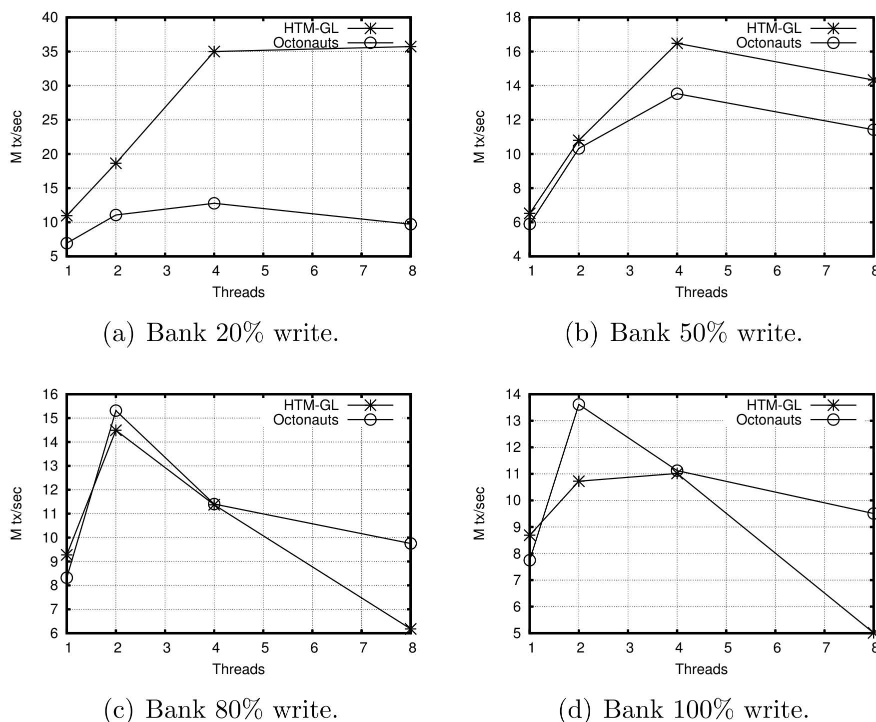
This technique seems to favor HTM transactions, but since both HTM and STM transactions subscribe to the same scheduler queues, HTM and STM transactions can only conflict due to inaccurate determination of the working-set or due to Bloom filters false conflicts. Thus, STM transaction cannot suffer from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

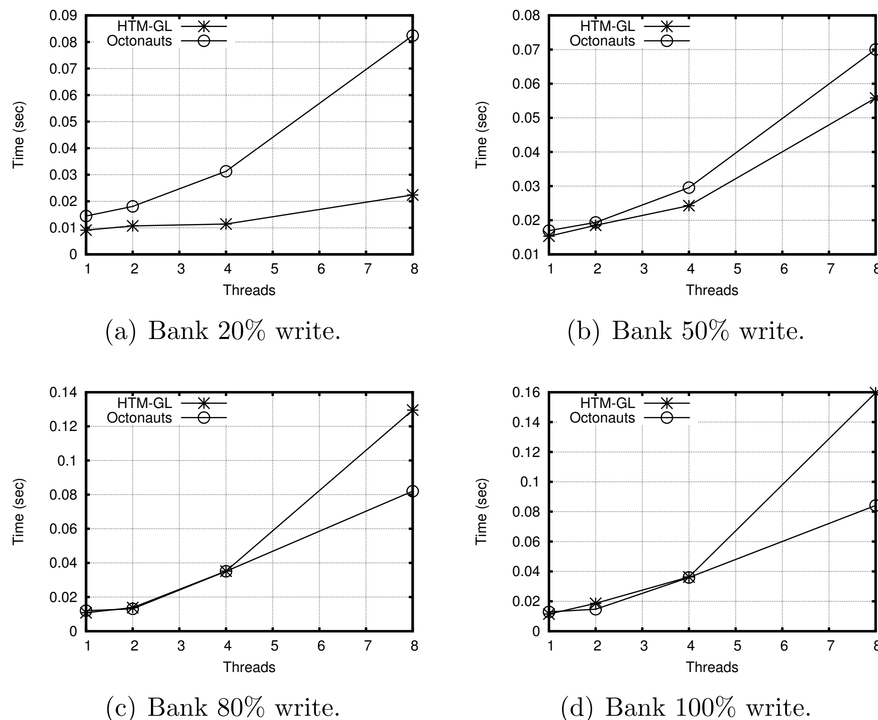
(d) Bank 100% write.

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).

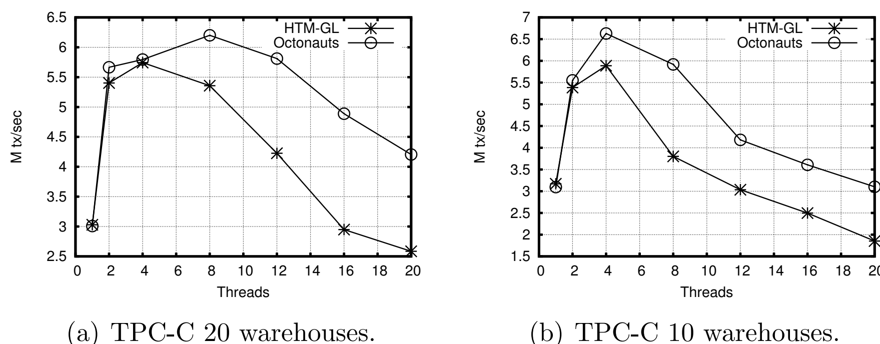
5.4 Evaluation

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 [27], 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.

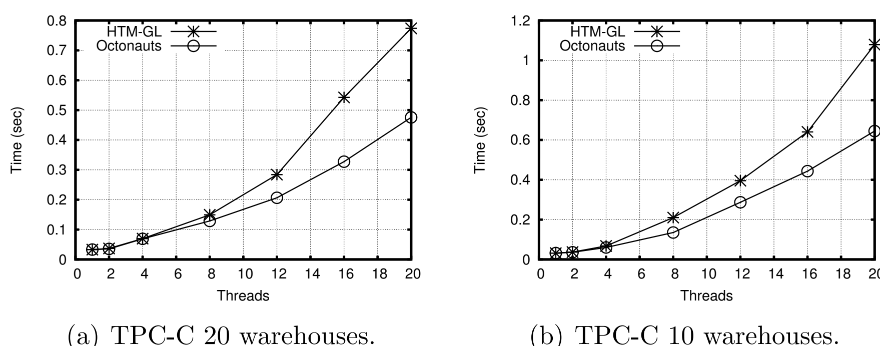
5.4.1 Bank

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

Precise-TM

6.1 Problem Statement

Transactional Memory community reached a consensus that HTM provides performance and scalability higher than STM. However, due to architecture design limitations, all the released processors that support HTM have no guarantees on the progress of HTM transactions [72], and hence any HTM algorithm is required to provide an alternative software *fallback* path. The default HTM algorithm for intel TSX APIs [78] protects the *slow-path* with a single global lock and monitors this lock at the beginning of the *fast-path* itself. We call this algorithm HTM-GL hereafter in this chapter.

Based on the experience of a decade of research in STM, when research moved back to HTM it was not surprising to propose the best STM algorithms as candidate fallback paths to HTM transactions. That is why one of the first proposals was falling back to TL2-like software path [68, 81] (we call it TL2-HTM). In the context of pure STM algorithms, it has been shown that TL2 [35] performs and scales better than most of the other STM algorithms because it uses fine-grained ownership records to lock and monitor memory locations, which reduces false conflicts and increases the level of concurrency. However, in the HTM context, proposals subsequent of TL2-HTM [69, 19, 20], which fall back to algorithms that do not scale as good as TL2 in their software versions, have been shown to perform and scale better than TL2-HTM. The main reason for this apparently inconsistent behavior is related to the nature of HTM itself: execution optimistically starts in an HTM *fast-path*, and in case of failure it falls back to a software *slow-path*. Based on this pattern, the software *slow-path* should have a minimal interference with the HTM *fast-path* even if the *slow-path* becomes less optimized. TL2-HTM fails to achieve that goal because it uses fine-grained meta-data in the *slow-path* (i.e. the ownership records) that are required to be monitored in the *fast-path*. This need of monitoring them adds at least one more read or write operation on a meta-data per memory access. Considering the problem of having limited resources in the

current HTM architectures, which has been discussed earlier in Chapter 4, the HTM *fast-path* in TL2-HTM is negatively affected, and subsequently the overall performance degrades. On the other hand, approaches that appear later in literature avoids this problem by using lightweight software *slow-paths* that use minimal meta-data, usually one global lock that is acquired either at the beginning or during the commit phase of the *slow-path*. Those approaches limit the effect of the *slow-path* on the *fast-path*, similarly to HTM-GL.

6.1.1 Drawbacks of Using a Single Global Lock as *Slow-path*

Despite the importance of having a lightweight *slow-path*, relying on a global lock (which is the common way to make the slow-path lightweight as proposed so far) has clear limitations due to two major issues:

- The software *slow-paths*, or at least parts of them (usually their commit phases), are executed sequentially, which results in poor scalability in the cases where transactions repeatedly fall back to the *slow-path*.
- The global lock has to be monitored in the *fast-path* in order to guarantee the synchronization with transactions running in the *slow-path*. The direct impact of monitoring the global lock in the *fast-path* is that it gives higher priority to the *slow-path* than the *fast-path* (i.e., HTM transactions running in the *fast-path* will abort when a transaction running in the *slow-path* acquires the global lock).

Algorithm 1 shows how those two issues appear in HTM-GL. The first issue is clear because the *slow-paths* are completely serialized using the global lock (Line 14). Even though we acknowledge that a fallback path relying on a single global lock is needed to guarantee the execution of those transactions that perform irrevocable (or more in general non-rollbackable) operations, it is still true that it introduces a coarse-grained serialization point when those (rare) transactions are not invoked. The second issue is raised because of Line 8, which starts the *fast-path* by checking the global lock. This line is important for the safety of the *fast-path* because it is not guaranteed whether any concurrent *slow-path* is conflicting with it or not. However, Line 8 is too pessimistic since it prevents the *fast-path* from running concurrently with any *slow-path* even if the two paths do not conflict with each other.

Summarizing, Lines 8 and 14 enforces Algorithm 1 to alternately execute transactions in two mutually exclusive phases: one phase that executes multiple HTM *fast-paths*, and another phase that executes a single software *slow-path*, while giving higher priority to the latter than the former.

6.1.2 On Reducing the Effect of Global Locking

Some optimization has been recently proposed to enhance the performance of HTM-GL.

Algorithm 1 HTM-GL Algorithm.

```
 1: procedure Tx-Begin
 2:     tries ← 2
 3:     while true do
 4:         while isLocked(global-lock) do
 5:             PAUSE
 6:         status ← _xbegin()
 7:         if status = OK then
 8:             if isLocked(global-lock) then
 9:                 _xabort();
10:             break
11:         else
12:             tries ← tries - 1
13:             if tries ≤ 0 then
14:                 acquire(global-lock)
15:                 break
16: end procedure
17:
18: procedure Read(x)
19:     return x
20: end procedure
21:
22: procedure Write(x, val)
23:     x ← val
24: end procedure
25:
26: procedure Tx-End
27:     if tries > 0 then
28:         _xend()
29:     else
30:         release(global-lock)
31: end procedure
```

First, since it is useless to start an HTM *fast-path* while the lock is acquired by a concurrent *slow-path*, any transaction waits until the global lock is released before starting the *fast-path* (Line 4). Adding this line is important as it avoids the lemming effect of Line 8 (i.e. cascading the failures in HTM, ending up with all transactions running in the *slow-path*) [33]. This optimization is enabled in the HTM-GL version we used in our implementation and evaluation study.

Another common optimization is the *lazy subscription* to the global lock, which means deferring Line 8 to the end of the *fast-path* [20]. This optimizations reduces the time the global lock is monitored in the *fast-path*, and hence it reduces the probability of aborting HTM transactions due to conflicts on the global lock. However, *lazy subscription* does not solve the original problems of serializing the *slow-paths* and treating conflicting and non-conflicting *slow-paths* similarly in the *fast-path*. Additionally, and more importantly, it has been proven in [32] that this solution is not safe because it breaks opacity [45]. Although any unexpected behaviour due to breaking opacity will be sandboxed by HTM in most workloads, the authors of [32] show some scenarios where zombie transactions are not sandboxed and may provide an unexpected behaviour. For that reason, we did not include this optimization in the HTM-GL version we used.

Although the approaches we proposed in Chapters 4 and 5 address different problems related to the best-effort nature of current HTM processors, they implicitly aim at solving the same high-level problem: minimizing the effect of the global locking in the *slow-path* by reducing the probability of falling back to it, either by partitioning long transactions to fit in

HTM (in Part-htm) or by providing an efficient scheduling of HTM/STM transactions (in Octonauts). A similar approach proposed in literature is to dynamically tune the number of retrials in the *fast-path* before falling back to the *slow-path* [37].

However, the effectiveness of all those approaches decreases if the workload cannot avoid the generation of some transaction that need to fall back to the *slow-path* for being successfully executed. For example, if a transaction calls unsafe instructions[1], it will always fall back to the *slow-path* even with the existence of the aforementioned optimizations. Also, in some dynamic workloads, the scheduling/tuning may not converge on accurate settings that ensure high performance. The problem becomes more difficult to address if the workload contains some transaction that always fails to execute in HTM, and some other transaction that natively fits HTM time/space constraints. In those cases, using an inefficient global locking approach may significantly affect the performance the latter by enforcing them to unnecessarily abort and fall back to the *slow-path* (i.e., being serialized along with software transactions).

In this chapter, we aim at solving the problem of global locking in such workloads by moving back to the original fine-grained locking direction, but overcoming the existing limitations. Since the main issue in the former fine-grained designs was in the overhead of handling meta-data, our main target is to minimize this overhead. Specifically, we introduce Precise-tm, an HTM algorithm that uses a fine-grained locking approach in the *slow-path* with a minimal interference with the HTM *fast-path*.

Precise-tm is orthogonal to Part-htm and Octonauts: in the cases where those approaches succeed to avoid falling back to the *slow-path*, Precise-tm only adds a marginal overhead to the *fast-path*, otherwise, Precise-tm reduces the overhead of acquiring a global lock in the *slow-path*.

6.2 Precise-TM Design Principle: Fine-Grained Embedded Locks

The core idea of Precise-tm is to replace the global lock that is acquired in the *slow-path* (Line 14) and monitored in the fast path (Line 8) with fine-grained locks. Doing that naively means that every read/write in the *fast-path* will check also the lock attached to the memory location, which adds significant overheads on the fast path.

To avoid such an unnecessary overhead of the fine-grained locking mechanism, in Precise-tm we exploit the following intuitions:

- **References can be locked/monitored using *address-embedded-locks*:** If a transaction only reads and/or writes variables by reference, we can reuse the idea

[1] In TSX, Intel identified some instruction that will always result in aborting HTM transactions.

of *address-embedded-locks* (or embedded locks) that we used in the opaque version of PART-HTM to synchronize transactions running in both *fast-path* and *slow-path*. The idea can be reused in PRECISE-TM in a much simpler way than PART-HTM.

Specifically, since the read/write operation is already on references, there is no need for wrapping them. The only requirement is that the read/written references are properly memory-aligned so that the stolen bits for embedding the locks are not used to identify the referenced memory location. In PRECISE-TM, as we will show later, we need to steal the least significant two bits of the references for embedding the locks, which means that the referenced variables should be aligned at four bytes. This assumption is acceptable when referencing most of the scalars (e.g. integers), as well as the `structs` that are composed of those scalars.

The main advantage of embedding locks into the references themselves is that every read/write in the *fast-path* does not need an extra meta-data to be read/written. This way, we allow the *fast-path* to speculate references without any overhead on the limited resources of HTM transactions (i.e., no additional cache-lines are needed). Additionally, the conflict detection granularity is at the level of the memory references themselves, which minimizes false conflicts, unlike the former techniques that use ownership records (e.g. TL2-HTM [68] or Refined Lock Elision [34]), where the granularity of the lock tables is clearly more coarse-grained. Summarizing, the name "PRECISE-TM" reflects the fact that it provides the most precise conflict detection between the *slow-path* and the *fast-path* without any additional meta-data to be monitored in the *fast-path*.

- **Scalars can use the original global lock:** For any transaction that reaches a read or write operation where the locks cannot be embedded (e.g., scalar variables or non-aligned references), a safe fallback strategy is to start locking or monitoring a global lock. This means that for any arbitrary transaction, locking (in the *slow-path*) and monitoring (in the *fast-path*) can be kept fine-grained until the first read or write that is not compatible with the *address-embedded-locking* mechanism occurs.

  The main advantage of this approach is that it guarantees an execution that is, in the worst case, similar to the default HTM algorithm that falls back to global locking.

- **Embedded locks are used only to notify HTM transactions:** In the original HTM-GL algorithm, the global lock is used for two reasons. First, if two transactions fall back to the *slow-path*, the global lock guarantees executing them sequentially. Second, if a transaction X is executing in the *fast-path* and transaction Y is executing concurrently in the *slow-path*, the global lock is used to abort transaction X. In other words, it allows both X and Y to run safely without the need for making the reads and writes of transaction Y visible to transaction X.

  Since it is clear that the last case (fast-path/fast-path synchronization) is internally handled by HTM, any concurrent execution is guaranteed to be consistent irrespective of the path of each transaction. Our *address-embedded-locking* mechanism focuses only

on optimizing the *fast-path/slow-path* synchronization. Specifically, in the aforementioned example, it replaces the global locking approach with a mechanism that makes reads/writes of transaction Y visible to transaction X. The importance of this observation is that *slow-path/slow-path* synchronization can be designed independently from the *address-embedded-locking* mechanism. For example, the *slow-paths* can be still sequentially executed using a global lock. Also they can be synchronized using traditional *mutex locks* or *readers-writer locks*. In Section 6.3, we show how this observation allows for more optimizations in designing the *slow-path*.

Based on the above observations, we design two versions of PRECISE-TM. The first version (we call it PRECISE-TM-V1) uses a global lock in the *slow-path* but does not naively monitor it in the *fast-path*. Instead, the global lock is only monitored when the *fast-path* reaches a read/write operation that cannot be monitored using *address-embedded* locks. The second version (we call it PRECISE-TM-V2) uses the stolen bits as fine-grained mutex locks in the *slow-path* instead of the global lock. In Section 6.3, we show the details of those two versions. Then, in Section 6.4, we compare them with HTM-GL, showing the advantages of each.

6.3 Algorithm Details

6.3.1 Precise-TM-V1: Precise Monitoring in the *Fast-Path*

In PRECISE-TM-V1, like HTM-GL, we use a global lock to protect the *slow-path*. The main difference between HTM-GL and PRECISE-TM-V1 is that the latter does not monitor the global lock at the beginning of HTM transactions. Serializing the *slow-paths* simplifies the design because it allows only one transaction (executing a *slow-path*) to lock/unlock the *address-embedded-locks* of the references at a time.

In PRECISE-TM-V1, we steal two bits from any reference for embedding the locks, one for reading and one for writing. Distinguishing the two cases is important because it optimizes the read-read conflict cases. Generally, two transactions are conflicting if they both access the same variable and at least one of them writes that. That is why, optimally, a *fast-path* should never abort if it has a read-read conflict with a *slow-path* on a certain reference. As we mentioned before, those stolen bits are only used to synchronize *fast-paths* with *slow-paths* because *slow-slow* synchronization is guaranteed by the global lock, and *fast-fast* synchronization is guaranteed by HTM. We only need two bits because the *fast-path* does not need to know the owner of the lock and only cares whether the reference is locked or not.

Algorithm 2 shows the implementation details of PRECISE-TM-V1. In the remaining of this section, we briefly discuss each component in Algorithm 2. It is worth to note that since the *slow-path* is executed in a mutual exclusion mode, any transaction that succeeds to acquire

the global lock is guaranteed to complete. Thus, there is no need to define an abort handler for those transactions.

Algorithm 2 PRECISE-TM-V1 Algorithm.

```
1: procedure INITIALIZE
2:     initialize reference-log array
3: end procedure
4:
5: procedure TX-BEGIN
6:     tries ← 2
7:     while true do
8:         status ← _xbegin()
9:         if status = OK then
10:            break
11:        else
12:            tries ← tries - 1
13:            if tries ≤ 0 then
14:                acquire(global-lock)
15:                break
16: end procedure
17:
18: procedure READ-REFERENCE(x)
19:     if fast-path then
20:         if x & 0x0002 then
21:             _xabort();
22:     else
23:         x ← x | 0x0001
24:         add(reference-log, x)
25:     return x & !(0x0003)
26: end procedure
27:
28: procedure WRITE-REFERENCE(x, val)
29:     if fast-path then
30:         if x & 0x0003 then
31:             _xabort();
32:         x ← val
33:     else
34:         x ← val | 0x0002
35:         add(reference-log, x)
36: end procedure
37:
38: procedure READ-SCALAR(x)
39:     if fast-path and isLocked(global-lock) then
40:         _xabort();
41:     return x
42: end procedure
43:
44: procedure WRITE-SCALAR(x, val)
45:     if fast-path and isLocked(global-lock) then
46:         _xabort();
47:     x ← val
48: end procedure
49:
50: procedure TX-END
51:     if tries > 0 then
52:         _xend()
53:     else
54:         for each ref in reference-log do
55:             ref ← ref & !(0x0003)
56:         clear(reference-log)
57:         release(global-lock)
58: end procedure
```

Initialization

Each thread defines a local array of references (called *reference-log* array) that is used to save the references accessed by its transactions during their *slow-paths*. This array is used to reset the embedded locks at the end of the *slow-path*. We store this array, along with a unique transaction ID, in a local context attached with each thread.

Transaction Begin

Each transaction tries n times in the *fast-path* before falling back to the *slow-path* as usual ($n = 2$ in our evaluation study). Also, like HTM-GL, the *slow-path* starts with the global lock acquisition. The transaction begin in PRECISE-TM-V1 differs from HTM-GL in two aspects: *i)* the *fast-path* does not monitor the global lock; *ii)* there is no need to spin on the global lock before starting the *fast-path* if it only accesses references because the *fast-path*

will not be affected by the global lock in such cases. This simply means that Lines 4 and 8 in Algorithm 1 are removed.

Removing those lines is the main source of performance gain in PRECISE-TM-V1 because it allows the *fast-path* to start immediately without the need of waiting for the completion of concurrent *slow-paths*, and with a precise conflict management strategy that aborts in the *fast-path* only if there is a real conflict with any of the concurrent transactions in the *slow-path*.

Reading References

When a transaction reads a reference, it has the obligation to handle the *address-embedded-locks* of that reference. Specifically, the *address-embedded-locks* should be acquired in the *slow-path* and monitored in the *fast-path*. It is worth to recall that the opposite (i.e., acquiring the *address-embedded-locks* by HTM transactions) is meaningless given that any modification made by a HTM transaction is invisible to any other transactions (either software of hardware) before the HTM transaction itself commits.

If a transaction is in the *slow-path*, it sets the *read-lock* bit to notify any concurrent *fast-path* that attempts to write the value stored by the same reference. After that, the transaction adds the reference to its local *reference-log* array. Then, it returns the reference with its stolen bits cleared.

If a transaction is in the *fast-path*, it checks the *write-lock* bit of the reference and aborts itself if it is locked (self abort). Checking the *read-lock* bit is not needed because read-read conflicts are allowed. Avoiding this check saves unnecessary aborts when the transaction reaches this read while a concurrent transaction is already acquiring the *read-lock* of the reference. However, and unfortunately, once the reference is read, any access to that reference in a concurrent *slow-path* will abort the transaction because the concurrent *slow-path* will write (modify) the embedded *read-lock* of the reference and hence will invalidate the reference itself.

Writing References

Similar to the case of reading references illustrated above, a transaction in the *slow-path* sets the *write-lock* bit and then it adds the reference to its local *reference-log* array. There is no need to distinguish between reads and writes in the *reference-log* array because in both cases the two bits are cleared at the end of the transaction (recall that we restrict our design to aligned references, where originally those two bits are always zeros, namely they do not count in identifying the actual value addressed).

If a transaction is in the *fast-path*, it has to check both *read-lock* and *write-lock* bits to avoid conflicting with both reading and writing *slow-paths*. Although this approach gives

slow-paths higher priorities than *fast-paths*, it does so in a fine-grained manner, which has much lower negative impact than the global lock fallback path in HTM-GL.

Reading/Writing Scalars

PRECISE-TM-V1 is favorable in workloads that mainly access references. However, we provide a safe fallback strategy for transactions that access scalars at any point of their execution. This is done by monitoring the global lock when the first read/write to a scalar appears. Given that the global lock is already acquired at the beginning of the *slow-path*, starting from this point PRECISE-TM-V1 behaves as HTM-GL, with a constant (and marginal) overhead of checking the embedded locks before every read or write operation to a reference.

Transaction End

The transaction completion has the same responsibilities as HTM-GL. A transaction calls *_xend* if it is in the *fast-path*, and unlocks the global lock if it is in the *slow-path*. The only difference is in the *slow-path*, where all the references accessed during the path (saved in the local *reference-log* array) are unlocked by resetting their stolen bits. Then, the local array itself is cleared. Resetting the embedded locks is safe because at most one transaction is executing a *slow-path* at a time.

6.3.2 Precise-TM-V2: Precise Locking in the *Slow-Path*

PRECISE-TM-V1 reduces the contention management granularity between *fast-path* and *slow-path* to the most precise level (i.e., the memory locations). However, it still serializes the *slow-paths* using a global lock. Although this approach simplifies the design of the *slow-path*, it may affect the performance if the transactions that fall back to the slow-path are non-conflicting. Given that HTM transactions may fail due to many reasons other than conflicts, this scenario can practically happen, resulting in executing non-conflicting transactions serially.

PRECISE-TM-V2 addresses this problem by reducing the contention management granularity between two *slow-paths* as well. Since transactions in PRECISE-TM-V1 already steal two bits from each references to manage conflicts with the *fast-path*, PRECISE-TM-V2 exploits those bits to lock references in the *slow-path* instead of using a global lock. In that sense, the precision in detecting conflicts between *slow-path/slow-path* becomes similar to that of *fast-path/slow-path*. Given that the precision of the latter (i.e., *fast-path/fast-path*) is non-configurable as it depends on the HTM contention management itself, PRECISE-TM-V2 achieves the best precision in all the cases.

Using fine-grained two-phase locking in the *slow-path* adds two obligations on the transaction

execution: *i)* transactions have to eventually abort if they fail to acquire locks (otherwise they may deadlock); *ii)* transactions have to save their writes to be undone in case of aborts. Providing that is straightforward but it needs a particular care because, unlike Precise-tm-v1, transactions in the *slow-path* may repeatedly abort and retry, and thus an efficient contention manager would be needed. Generally, if transactions are conflicting, Precise-tm-v1 is expected to be better because Precise-tm-v2 will suffer from frequents aborts (Intuitively, the best way to execute conflicting transactions is to execute them sequentially). We detail this point in the next section.

Algorithm 3 shows the implementation details of Precise-tm-v2. In the remaining of this section, we briefly discuss each procedure of Algorithm 3. Specifically, we show how each component is different from the corresponding one in Precise-tm-v1.

Initialization

Initialization in Precise-tm-v2 is similar to Precise-tm-v1 except that the *reference-log* array should store both the references accessed and their old values. In case of *slow-path* aborts, those values will be used to rollback the changes made by the transaction.

Transaction Begin

Unlike Precise-tm-v1, the global lock is not acquired when the *slow-path* starts. This is the main performance advantage of Precise-tm-v2 over Precise-tm-v1 as it allows more concurrency between *slow-paths*. Other than that, transaction begin handler is similar to Precise-tm-v1.

Reading/Writing References

Reading and writing references in the *fast-path* is similar to Precise-tm-v1. In the *slow-path*, however, the stolen bits need to be atomically modified in order to achieve an exclusive access on the reference, and prevent any concurrent transaction running in the *slow-path* from accessing the same reference. To do so, any transaction starts reading/writing references by checking that this reference is not locked by any other transaction (either for reading or for writing), specifically by checking the least significant two bits. Then it tries to atomically change the corresponding bit (according to whether the operation is read or write). If it fails, it aborts the transaction. If it succeeds, it saves the reference, along with its old value, in the *reference-log*. For simplicity, we save the reference and its old value in both reads and writes.

Although we differentiate between reading and writing in the *fast-path*, we do not do so in the *slow-path* to keep the algorithm design simple. That is why we abort the transaction

if the reference is locked by another transaction for either reading or writing. Transactions clearly should not abort if the reference is self-locked. We detect this case by scanning the *reference-log* array to know whether the lock is acquired by the same transaction or by another one.

Reading/Writing Scalars

Before reading or writing scalars, transactions get an exclusive access to them. This is achieved by acquiring the global lock before the first read/write to a scalar occurs in the slow-path, and monitoring the global lock before the occurrence of every read/write to a scalar in the *fast-path*. As we mentioned before, like most HTM algorithms, this approach gives higher priority to the *slow-path*.

Unlike PRECISE-TM-V1, since the global lock is no longer the only lock acquired in the *slow-path*, the *tryLock* primitive is used instead of *Lock* to avoid deadlocks. If the *tryLock* fails, the transaction aborts similar to the way it does in reading/writing references.

Transaction End

This routine resets the stolen bits in all the accessed references and then clears the *reference-log* array. Additionally, the global lock is released only if it was acquired due to a scalar read/write operation.

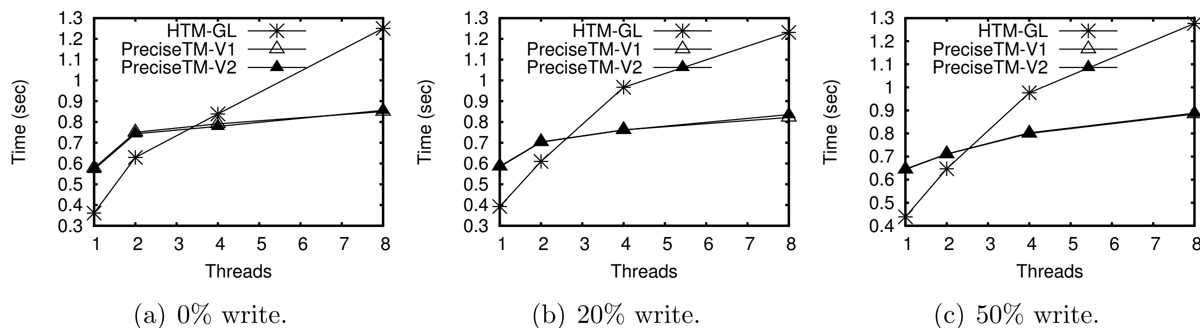
Abort in the *slow-path*

The *slow-path* aborts when either the global lock or the lock of one of the accessed references is found to be locked by another transaction. Aborting a *slow-path* is similar to the previous routine (which represents committing a *slow-path*) except that for each entry in the *reference-log* array the old value is restored (instead of resetting to the stolen bits for each entry).

At the end of the abort handler, we call a statistical *assess-abort-rate* method that measure the frequency of aborts in the *slow-path* so far and decide accordingly whether it is better to switch to PRECISE-TM-V1 or not. This method forms a simple contention management approach that catches the cases when transactions start to repeatedly abort each other due to competing on the same set of locks.

6.4 Evaluation

To evaluate PRECISE-TM, we implemented it in C++ and tested it on an Intel Haswell Core i7-4770 processor (4 cores, 8 hardware threads) and GCC 4.8.2. All data points are the



(a) 0% write. (b) 20% write. (c) 50% write.

Figure 6.1: Execution time using Bank benchmark.

average of 5 repeated execution.

We compared PRECISE-TM to HTM-GL since we consider PRECISE-TM the precise fine-grained version of HTM-GL. Showing such a comparison helps reasoning about the effect of making any *global-lock-based* HTM algorithm more precise using our idea. We believe that the same idea can be extended for algorithms other than HTM-GL.

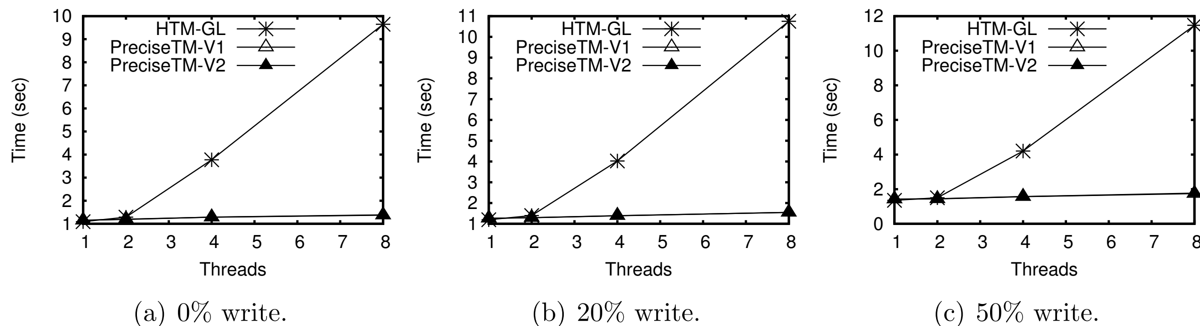
To conduct a comprehensive evaluation, we tested PRECISE-TM in three different workloads: Bank; EigenBench [53]; and linked-list data structure. Those workloads have different characteristics that show the advantages of both versions of PRECISE-TM as well as their limitations.

6.4.1 Bank

The first set of experiments is a customized *Bank* benchmark, where a set of 64K accounts (typically an array of accounts references) are accessed using two API methods: *transfer*, which selects two random accounts and transfers money from one account to the other; and *check-balance* which returns the current balance of an account. This way, both methods are executed within a transaction that only reads from and writes into references, which is the best test case of PRECISE-TM.

Figure 6.1 shows the results of an experiment that creates different number of client threads and runs a fixed number of operations (10M operations) on each thread. The execution time for three different configurations, where 0%, 20%, and 50% of the operations are *transfer* operations (i.e., writing transactions), is measured in Figures 6.1(a), 6.1(b), and 6.1(c) respectively.

A general observation in all the plots is that both PRECISE-TM-V1 and PRECISE-TM-V2 perform better than HTM-GL for high number of threads (4-threads and 8-threads). This is expected because when contention increases, all algorithms start to observe more transac-



(a) 0% write. (b) 20% write. (c) 50% write.

Figure 6.2: Execution time using Bank benchmark with disjoint accesses to accounts.

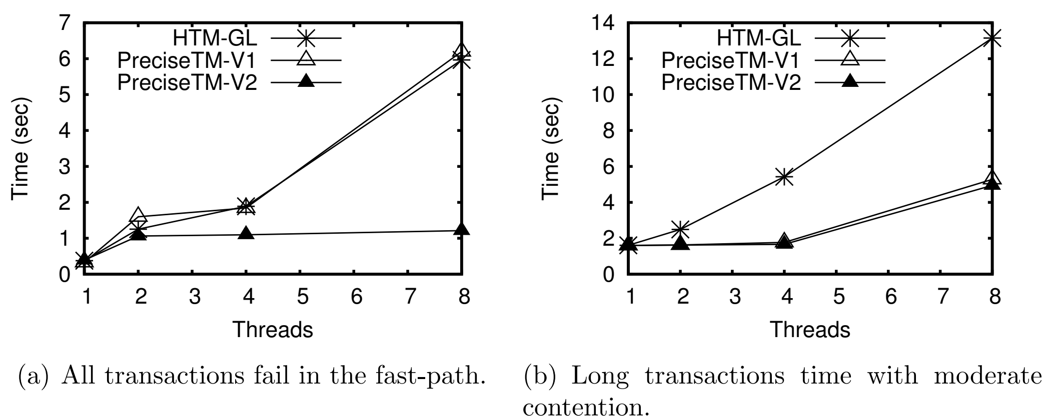
tions falling back to the *slow-path*. Since HTM-GL acquires a global lock to execute those transactions, more transactions fail in the *fast-path* because they monitor the global lock. PRECISE-TM, on the other hand, does not monitor the global lock (PRECISE-TM-V2 does not even acquire it in the *slow-path*), and thus it does not suffer from cascading aborts in the *fast-path*.

The second observation is that when all the operations are *check-balance* operations (Figure 6.1(a)), the execution time in PRECISE-TM remains the same starting from two threads. This is also expected because conflicts are rare, thus concurrency between transactions is maximized. HTM-GL does not behave similarly because it suffers from failures due to reasons other than conflicts (recall that HTM may fail due to other reasons like capacity failures and external interferences). The level of concurrency of PRECISE-TM decreases in the writing cases, but it remains better than HTM-GL.

For small number of threads (1-thread and 2-threads), HTM-GL performs better than PRECISE-TM. The main reason for that is that PRECISE-TM adds a constant overhead of locking and monitoring the stolen bits for each read and write without a real benefit because most of the transactions succeed in the *fast-path*.

We also measure the percentage of transactions that fall back to the *slow-path* to better understand the relation between performance and frequency of failures in the *fast-path*. We found that in all the cases (even the *read-only* one) the average ratio of falling back to the *slow-path* is 20% for 4-threads and 30% for 8-threads in HTM-GL, and almost 0% for the two versions of PRECISE-TM in all thread counts, which confirms the results in Figure 6.1. It is important to note that this measurement is not the only factor that affects the execution time. For example, spinning on the global lock before starting the *fast-path* of HTM-GL is another factor. However, measuring the *fast-path* aborts gives a good intuition about the behaviour of each algorithm.

In Figure 6.2, we repeated the same experiment while making the accesses to accounts disjoint (i.e. every thread accesses different set of accounts). This modification is biasing the



(a) All transactions fail in the fast-path. (b) Long transactions time with moderate contention.

Figure 6.3: Execution time using EigenBench benchmark for two special cases.

benchmark more to algorithms like PRECISE-TM because in this case there is no contention at all. That is why both versions of PRECISE-TM linearly scale (which is inferred from having a constant execution independent from the number of threads). On the other hand, HTM-GL still suffers starting from four threads for the same problem of cascading aborts (that are initially raised because of HTM limitations) due to monitoring/locking the global lock. In low thread count, all the algorithms perform similarly.

6.4.2 EigenBench

EigenBench is a configurable benchmark that can be used to generate different workloads, including the special cases of execution. We exploited that to generate two of those special cases, shown in Figure 6.3 in order to complete the picture about the behaviour of PRECISE-TM.

The first case, shown in Figure 6.3(a), is when all transactions fail in the *fast-path* (due to capacity failures) and fall back to the *slow-path*. In this specific case, all algorithms will behave similarly in the *fast-path*. That is why both HTM-GL and PRECISE-TM-V1 perform similarly, because they also behave similarly in the *slow-path* (i.e. both acquire a global lock). However, PRECISE-TM-V2 performs better in this case, which shows the benefits of having fine-grained locking approach in the *slow-path*

The second case, shown in Figure 6.3(b), represents the case of having long transactions with moderate contention, where not all transaction fail in the *fast-path*, but the percentage of them is more than Bank benchmark. That is why HTM-GL starts to perform worse than PRECISE-TM even for low number of threads.

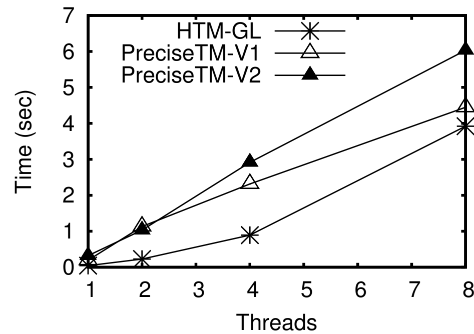


Figure 6.4: Execution time using linked-list benchmark of 5K elements and 20% write operations.

6.4.3 Linked-list

In this experiment, shown in Figure 6.4.3, we show the worst case for PRECISE-TM, where a linked list of 5K elements is accessed with 20% write operations (inserts and removes). In this case, as all the transactions start traversing the list from its head, it becomes useless for PRECISE-TM to reduce the granularity of locking/monitoring because all transactions acquire at least a *read-lock* on the head. This means that all the overheads added by PRECISE-TM are not exploited. It is also clear why PRECISE-TM-V2 performs worse than PRECISE-TM-V1, because it adds more overheads. In fact, this experiment shows one of the major drawbacks of two-phase locking approaches in general.

Algorithm 3 PRECISE-TM-V2 Algorithm.

```
1: procedure INITIALIZE
2:     initialize reference-log array
3: end procedure
4:
5: procedure TX-BEGIN
6:     tries ← 2
7:     while true do
8:         status ← _xbegin()
9:         if status = OK then
10:            break
11:        else
12:            tries ← tries - 1
13:            if tries ≤ 0 then
14:                break
15: end procedure
16:
17: procedure READ-REFERENCE(x)
18:     if fast-path then
19:         if x & 0x0002 then
20:             _xabort();
21:     else
22:         if !isLocked(x) then
23:             if CAS(x, x & !(0x0003), x | 0x0001) then
24:                 add(reference-log, x)
25:             elseAbort-Slow-Path()
26:         else if isLockedByMe(x) then
27:             x ← x | 0x0001
28:         elseAbort-Slow-Path()
29:     return x & !(0x0003)
30: end procedure
31:
32: procedure WRITE-REFERENCE(x, val)
33:     if fast-path then
34:         if x & 0x0003 then
35:             _xabort();
36:         x ← val
37:     else
38:         if !isLocked(x) then
39:             if CAS(x, x & !(0x0003), val | 0x0002) then
40:                 add(reference-log, x)
41:             elseAbort-Slow-Path()
42:         else if isLockedByMe(x) then
43:             x ← val | 0x0002
44:         elseAbort-Slow-Path()
45: end procedure
46:
47: procedure READ-SCALAR(x)
48:     if fast-path then
49:         if isLocked(global-lock) then
50:             _xabort();
51:     else
52:         if !isLockedByMe(global-lock) and !tryLock(global-lock) then
53:             Abort-Slow-Path()
54:     return x
55: end procedure
56:
57: procedure WRITE-SCALAR(x, val)
58:     if fast-path then
59:         if isLocked(global-lock) then
60:             _xabort();
61:     else
62:         if !isLockedByMe(global-lock) and !tryLock(global-lock) then
63:             Abort-Slow-Path()
64:     x ← val
65: end procedure
66:
67: procedure TX-END
68:     if tries > 0 then
69:         _xend()
70:     else
71:         for each ref in reference-log do
72:             ref ← ref & !(0x0003)
73:         clear(reference-log)
74:         if isLockedByMe(global-lock) then
75:             release(global-lock)
76: end procedure
77: procedure ABORT-SLOW-PATH
78:     for each ref in reference-log do
79:         ref ← ref-old-value
80:     clear(reference-log)
81:     if isLockedByMe(global-lock) then
82:         release(global-lock)
83:     if assess-abort-rate = HIGH then
84:         Switch-to-PRECISE-TM-V1
85:     Restart
86: end procedure
```

Chapter 7

Nemo

7.1 Problem Statement

Transactional Memory (TM) is a powerful programming abstraction for implementing parallel and concurrent applications. TM frees programmers from the complexity of managing multiple threads that access the same set of shared objects. The advent of multi-core architectures, which provide (sometimes massive) parallel computing capabilities for thread execution, clearly favors the diffusion of TM. Today, this hardware is widely available on the open market; even inexpensive processors are equipped with more than one physical core, improving parallel computing capabilities.

The growing number of cores per processor has led designers to produce architectures where the whole address space is divided into multiple slices (or zones). The latency for performing a memory access varies depending on a number of factors, such as the processor on which the thread executes and the actual placement of the accessed memory location. Such a design, also called Non-Uniform Memory Access (NUMA) [65]), is becoming the de-facto standard for upcoming multi-/many-core platforms which possess extremely high parallel computing capability (e.g., Intel QuickPath Interconnect, Opteron/HyperTransport, UltraSPARC/FirePlane [96, 8, 25]).

Many algorithms to manage contention have been proposed since TM became a real and simple alternative to locking as a synchronization abstraction; however none of them have been specifically designed to achieve scalability in NUMA multi-core architecture. This stems from the fact that usually the logical content of the application itself prevents the full exploitation of the underlying hardware parallelism. In fact, when two or more application threads request the same memory space and at least one wants to perform a write operation on it, they cannot proceed in parallel. Instead, one of them should be executed after the other (i.e., serialized). Serializing their executions results in underutilizing one of the two threads. As a result, the overall application performance cannot be increased further and

scalability is no longer provided.

There is a class of workload where application data can be partitioned to provide scalability. Examples include TPC-C [27], Bank (i.e., a micro-benchmark resembling monetary operations), and (in general) in-memory databases. In this class of applications, data can be organized such that an object is placed in a zone close to the the thread the accesses the object. This organization fits the NUMA design - a processor is physically bound to one memory zone which offers very fast execution and access to other zones costs much more than the local one.

The TM literature lacks solutions that scale for workloads with characteristics similar to the ones described above. We name this class of workloads as *scalable*. Programmers expect these applications' performance to scale up when they increase the number of threads physically executing in parallel; however, this does not happen in existing TM solutions.

To quantitatively support our claim, we conduct an evaluation study consisting of two major tests. A detailed description and plots are reported in Section 7.2.

- With the first test, we deploy several state-of-the-art TM algorithms over an AMD 64-core machine equipped with 4 physical sockets, each of which hosts a 16-core processor. The memory is physically partitioned into 8 NUMA zones; each of these directly interfaces with 8 cores. We evaluate five algorithms, spanning from those relying on a single lock or a global timestamp to protect the transaction commit phase to those that lock individual written objects, thus enabling more concurrency at the cost of managing more meta-data. For our application, we use a version of the well-known Bank benchmark, which performs monetary transfers from multiple accounts. The entire shared dataset is partitioned across NUMA zones, and threads running on the cores of one NUMA zone are forced to access only objects stored on that NUMA zone.
- The second test aims to identify the inherent cost of a NUMA architecture when a single shared variable is used to manage the synchronization among parallel threads. In this test, each application thread increments a shared timestamp. The purpose of the test is to measure the difference in the latency needed to update (using a CAS operation) a shared timestamp that is physically located in the same NUMA zone the thread is working on versus one that is located in another NUMA zone.

The lesson learned from the above tests is twofold. On the one hand, letting non-conflicting threads access some shared meta-data causes traffic on the physical bus interconnecting different processor sockets and thus the NUMA zones. This is a common practice in designing TM solutions, as having shared meta-data allows transactions to efficiently identify an inconsistent operation. Unfortunately, given the constraints of NUMA architectures, updating that meta-data becomes the bottleneck when the workload is mostly non-conflicting (or scalable as defined above). On the other hand, when threads deployed within a single processor socket cooperate using shared variables stored in the NUMA zone connected to that socket,

the aforementioned bottleneck no longer arises, as the NUMA architecture handles such traffic inside the socket itself without making use of the slower inter-sockets bus.

We use the above observations as the design principles of a new TM algorithm, which we name NEMO. NEMO is scalable - in the presence of a scalable workload, its performance increases when the number of threads deployed on different cores increases. The core idea of NEMO is to treat conflicts involving transactions that execute within the same socket differently from those that involve transactions on another socket (or NUMA zone[1]). This allows NEMO to resolve some conflicts (e.g., those based on a single shared timestamp) simply and efficiently within a single socket, as updating shared variables is fast. To identify conflicts with a thread on another socket, it uses a low-overhead optimistic policy.

Such a design lets two threads access some common object or meta-data only when the threads belong to the same physical socket or when they run conflicting transactions. We name this property as *NUMA Disjoint Access Parallelism* (or NUMA-DAP) because it is built on top of the original (and more theoretical) DAP proposed in [55], where the NUMA performance constraints were not taken into account.

More practically, NEMO uses a single shared timestamp to synchronize the operations of threads executing within a socket. This timestamp is updated every time a writing transaction commits, and it is also used as a means for detecting a possibly dangerous transactional access to a shared object created after the transaction begins. When a transaction conflicts with a transaction executing on another socket, additional synchronization is needed to preserve the protocol's correctness. To do this, each thread keeps a cached version of the timestamps of other NUMA zones. Obviously, those cached copies are not necessarily up-to-date with the actual value of the timestamp stored in each NUMA zone. This results from the fact that given the NUMA-DAP property, non-conflicting transactions running on different NUMA zones should not access any common object - which also means that they cannot update the cached copy of the timestamp of another NUMA zone. The cached copies update when a transaction requests an object located in a different NUMA zone, and the object is associated with a newer timestamp than that of the cached copy. Thus, the object cannot perform a consistent operation. When such a case occurs, the transaction undergoes an additional check which reveals whether an abort/restart is needed or not. After that, the transaction updates its cached value of the other NUMA zone's timestamp and can proceed.

Clearly, aborting the transaction after discovering that the cached copy of a timestamp is outdated may seem costly; however this will likely happen infrequently when the application provides a scalable workload. In order to further reduce this overhead, we designed a version of NEMO that makes all of the cached copies of the other NUMA zones' timestamps available to all threads of one NUMA zone. That way, per-thread meta-data is reduced and transactions can benefit from accessing more recently-cached timestamps, even if the thread they are running on never accessed that NUMA zone. We name this version of NEMO as NEMO-VECTOR.

[1]In the rest of the chapter we use the terms "socket" and "NUMA zone" interchangeably.

NEMO provides Serializability [13] as a correctness level, as both flavors of NEMO ensure that the versions of the objects read during the transaction execution are identical to those currently committed before applying the modifications to the shared memory (i.e., committing the writes).

NEMO has been implemented in C++ and evaluated using well-known benchmarks (e.g., TPC-C [27]) which are properly modified to provide scalable workloads. However, in order to also assess the new protocol's performance in adverse scenarios, we tuned the percentage of transactions that perform accesses to objects allocated in a non-local NUMA zone. Our findings show that NEMO's scalability is strong. It is the only solution that continues to offer increased application performance beyond the threshold corresponding to the number of cores enclosed in a single socket. Specifically, NEMO outperforms TLC [12] and TL2-GV5 [35], which are NUMA-compatible, and all other STM approaches that we tested.

7.2 Non-Uniform Memory Access: architecture, characteristics, and performance using atomic operations

Recent multi-/many-core hardware architectures are composed of multiple sockets (usually 4 or 8), each of which can deploy a multi-core chip. In commercial platforms, a shared bus interconnection enables communication among hardware threads executing on different sockets. Emerging architectures also include a physical communication grid in which interactions exploit the message-passing paradigm.

The Non-Uniform Memory Access (NUMA) design is the de-facto standard for interfacing hardware threads with the main memory. In a NUMA design, one memory socket (i.e., a memory chip that constitutes a part of the overall system memory) is physically attached with one processor socket (or, if the socket is capable of maintaining multi-dies, one die inside the socket), thus creasing the so called NUMA zone. We say that a thread executing on a particular socket accesses a *local* NUMA zone when it accesses a memory location that is maintained within the NUMA zone connected to that socket. Otherwise, we say that the thread accesses a *remote* NUMA zone.

When a hardware thread accesses a memory location whose address is located in the local NUMA zone, the latency is very small (e.g., 9 nsec using DDR3-2000 memory) and the access is performed without contention on the shared bus resource. On the other hand, if the memory location is stored in a remote NUMA zone, the hardware thread is forced to use the shared bus that interconnects all of the sockets to fetch the desired value. The latter access is clearly slower than the former, and it decreases the overall parallel computing capability because the access to the shared bus is exclusive - only one thread at a time can use it. Conversely, if two threads operating in two different NUMA zones work on data

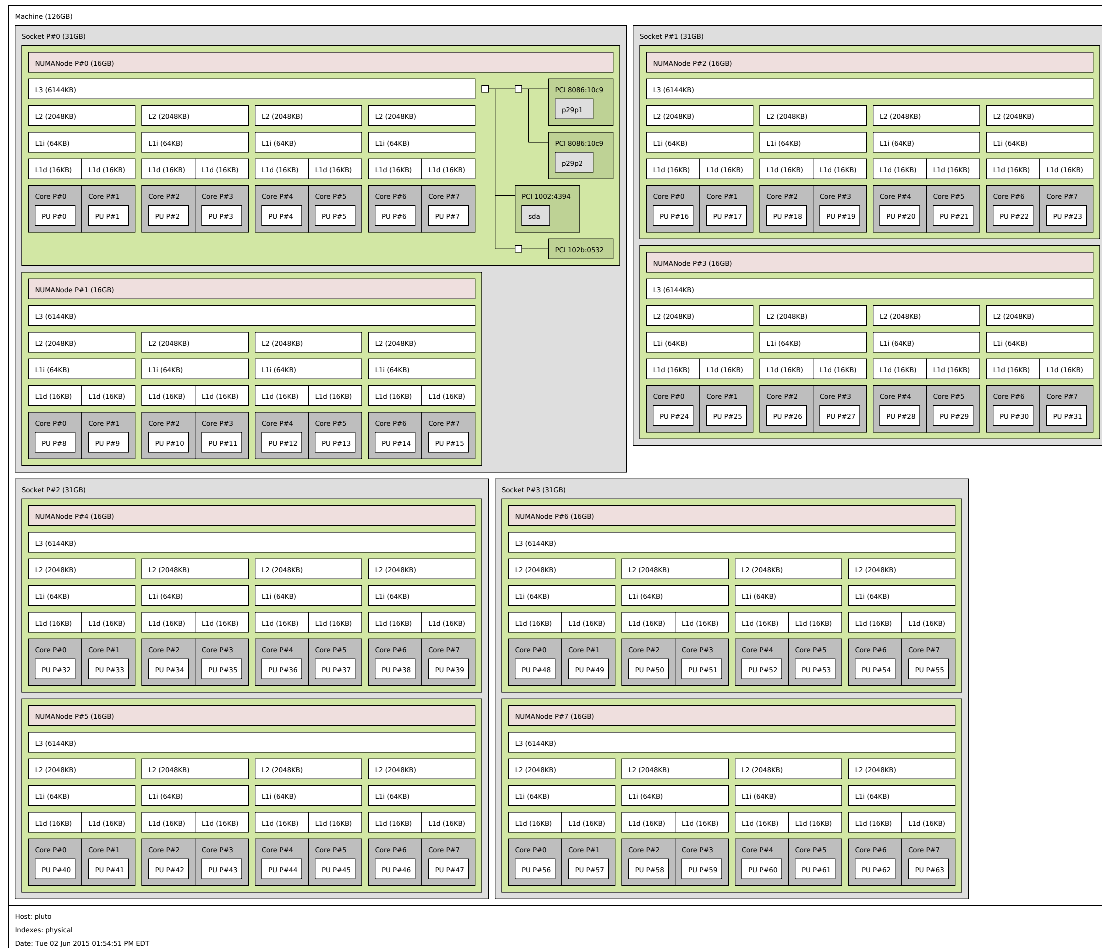


Figure 7.1: Hardware architecture of an AMD Opteron 64-cores (4 sockets and a 16 cores processor per socket.

stored in their own local NUMA zones, they can proceed in parallel without any hardware synchronization point (as is represented by the bus itself).

Figure 7.1 shows the hardware architecture of a widely used AMD 64-core commercially-available server. The figure shows the presence of four sockets, each containing a processor with two dies; each die contains 8 cores. There are a total of 8 NUMA zones (one per die). Overall, a set of 8 threads has fast access to its local NUMA zone.

On this machine, we perform the tests described in Section 7.1. Shortly, in the first test we deploy a version of the Bank benchmark where all of the *accounts* (the most contested object in Bank) are partitioned across NUMA zones and application threads operate only on accounts stored in their local NUMA zone. This workload matches our definition of scalable. As well-known TM algorithms, we implement TL2 [35], SwissTM [40], TinySTM [43], RingSTM [90], and NOrec [29]. TL2, SwissTM, and TinySTM use different conflict detec-

tion policies, but all lock written objects individually by relying on a shared lock table which in our case is partitioned across NUMA zones. Conversely, NOrec protects the transaction commit phase using a single shared lock, and RingSTM uses a (complex) ring data structure to catch invalid executions and abort them.

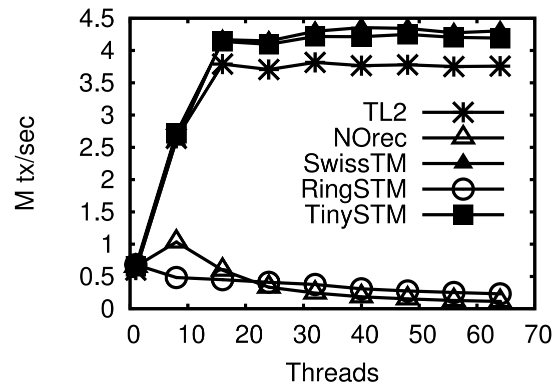


Figure 7.2: Bank benchmark configured for producing a scalable workload (disjoint transactional accesses).

The result of this experiment are shown in Figure 7.2. As expected, the algorithms using per object locks provide better performance than the others because they allow for more concurrency in the system, which pays off under a scalable workload. However, beside the specific performance provided, all of the algorithms stop scaling after 16 threads. This configuration represents the maximum number of parallel threads allowed within a single socket. After that point, the cost of updating global meta-data becomes very high as this operation likely involves traversing the shared bus that connects different sockets, therefore hampering the overall scalability. As an example SwissTM, which provides the best performance, relies on a single timestamp to validate transaction's read-set. This timestamp is incremented by every writing transaction, which represents a high contention bottleneck.

The second experiment shows specifically the latency needed for incrementing a shared timestamp. Two configurations are deployed. One uses a single timestamp located in one NUMA zone that all application threads increment. The other configuration includes 8 timestamps, where each is located in one NUMA zone and threads increment only the timestamp located in their local NUMA zone. The plot in Figure 7.3 shows the average time (in millisec) to perform 100k increments. On the x-axis we vary the number of threads per NUMA zone. For example, the datapoint at 3 threads represents the configuration with 3 threads per NUMA zone executing update operations, producing a total of 24 threads given the 8 NUMA zones available in the testbed.

Results show that updating a single timestamp does not provide any scalable performance given the high traffic generated on the shared bus among sockets. On the other hand, the cost for updating a local timestamp using a CAS operation is very small. It is worth noting that

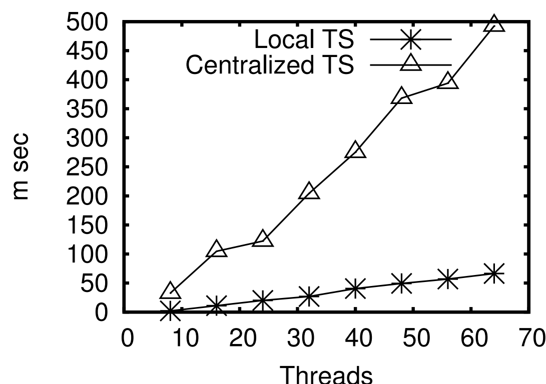


Figure 7.3: Average cost of 100k increment operations of a centralized vs local timestamp.

at any point in time there are always 8 timestamps getting updated in parallel using atomic operations. This consideration is important because it shows that even though there is local contention, the hardware is able to handle it locally at each socket without significantly affecting the work performed on other sockets. The centralized configuration ensures strong scalability.

The results detailed in this section form the basis of NEMO, our solution for providing scalable performance in the presence of scalable workloads. To accomplish such a goal, two principles should be taken into account: *1)* threads executing on data stored in different NUMA zones should not interfere with each other if the transactions they are executing do not manifest any conflict; *2)* threads executing on the same socket are allowed to share information to make their execution faster, as this cooperation will not significantly affect the performance of threads executing on other sockets. NEMO develops a solution that takes advantage of these characteristics.

7.3 Algorithm Design

Based on our observations of current NUMA machines as described in Section 7.2, a NUMA-aware TM algorithm should avoid any centralized shared meta-data and should limit data transfers between NUMA zones. In addition, we perform specific tests to show the interferences between atomic operations (e.g., CAS) executed on different NUMA zones in parallel. We find that each NUMA zone can handle NUMA-local those privileged operations without any scalability bottlenecks. Thus, our basic idea is to use a traditional centralized-like STM algorithm inside a NUMA zone and limit inter-NUMA zones communication to when they are actually needed - namely, when the transaction requests an object stored in another NUMA zone.

This approach of using two different schemes is similar to the Globally asynchronous locally

synchronous (GALS) hardware design principle. GALS relaxes the synchrony assumption by having synchronous islands communicating between each other asynchronously. Similarly, we relaxed the condition of having a single synchronized global timestamp and used synchronous islands where each has its own synchronized timestamp. Communication between these islands is asynchronous in a way that maintain correctness and isolation between islands.

In NEMO, we present two algorithms that satisfy these two main requirements: NEMO-TS and NEMO-VECTOR. In general, these algorithms work intra-NUMA zone, like TL2 [35], and inter-NUMA zone, like TLC [12], but with fine-grained caching techniques and other algorithmic optimizations. Another important factor is to keep meta-data in the same NUMA zone of its corresponding data (e.g., a separate lock-table per NUMA zone). In fact, it does not make sense paying an expensive cost for accessing an object located in a certain NUMA zone and then paying an additional cost to access another NUMA zone so as to analyze the object's meta-data.

The basic idea of NEMO-TS is to have a separate timestamp for each NUMA zone, called a *local timestamp*. Local transactions, which are those that work only on a single NUMA zone, use the local timestamp to perform a validation. No inter-NUMA-zone communication is needed in such a case. Similar to TL2, local transactions use *invisible reads*; thus concurrent writing transactions do not know about concurrent readers of their written object[2]. Each object has a version, which is the value of the local timestamp at the time the writing transaction commits. In addition, each transaction keeps a list of all read object in its *read-set*. A consistent view of read objects is maintained by validating the read-set using the transaction starting time (taken from the local timestamp when a transaction begins) and objects' versions. If a transaction reads an object with a version greater that the transaction's starting time, then the transaction aborts.

Transaction writes are buffered in the *write-set* until commit time. At commit time, a transaction acquires all locks associated with written objects in the write-set; it revalidates the read-set; it writes back the whole write-set to the main memory; it atomically increments the local timestamp; and finally it updates the acquired locks with the new timestamp value and unlocks them. If the locks acquisition fails or the read-set results are invalid after invoking the validation procedure, then the transaction aborts and restarts.

To support inter-NUMA-zone transactions, each thread keeps a local copy (or cached) of other NUMA zones' timestamps. This local cache acts like the starting time of a local transaction. When a transaction reads an object from a NUMA zone with a version greater than the local cached timestamp of that NUMA zone, the transaction aborts and updates the local cache of that zone. In order to reduce the number of unnecessary aborts due to outdated caching, we periodically refresh the cached timestamps. The latter operation is done without saturating the shared bus connecting the system's NUMA zones. In fact, when there is the chance of having outdated cached timestamps, it means that the shared bus is likely not busy given that few transactions are accessing objects from non-local NUMA

[2]We use the term *object* to refer to memory locations

zones. In such a case, it is worth using the shared bus to update the cached timestamp because very few transactions will be affected; many future transactions will benefit from this task by preventing unnecessary aborts.

When an inter-NUMA-zone transaction commits, it follows the same procedure of a local transaction; however, it also atomically increments all accessed NUMA zones' timestamps and updates the cached timestamps with the incremented values.

NEMO-VECTOR optimizes over NEMO-TS by introducing a *vector clock* per NUMA zone. This decision has a twofold benefit: *1)* The local cache of other NUMA zones' timestamps is at the NUMA zone level instead of at the thread level; and *2)* we can avoid some of the false conflicts due to outdated caching (as detailed later).

With NEMO-VECTOR, at the beginning of a transaction, the NUMA zone's vector clock is copied into the transaction context as its starting time. A local transaction proceeds the same as NEMO-TS. The local NUMA zone's entry in the vector clock represents the NUMA zone's local timestamp. Inter-NUMA transactions also proceed in a similar fashion to NEMO-TS, but when a conflict is detected, the NUMA zone's vector clock is updated. In addition, given that now there is no single value (i.e., the timestamp) to update, NEMO-VECTOR proposes an optimized solution to increment those multiple values efficiently. Accessed NUMA zone's vector clocks are locked, and the entries inside each vector are updated. For example, suppose the system is equipped with three NUMA zones Z_1, Z_2, and Z_3. Each zone will have a vector clock x of three entries $VCZ_x[i]$ where i is 1, 2, or 3. If a transaction touched Z_1, and Z_3, then at commit time, VCZ_1 and VCZ_3 are locked and entries $VCZ_1[1]$, $VCZ_1[3]$, $VCZ_3[1]$, and $VCZ_3[3]$ are updated to $max(VCZ_1[1], VCZ_1[3], VCZ_3[1], VCZ_3[3]) + 1$. We also designed another technique to update these vector clocks without locking, which will be illustrated in Section 7.4.

NEMO-VECTOR can also avoid the following false conflict scenario, which is critical to achieve high performance as it happens often. Suppose a transaction T_1 on a NUMA zone Z_1 wants to read from Z_2 for the first time (i.e., read X_{Z_2}). If the version of X_{Z_2} is greater than the local cache at the starting time, then in NEMO-TS the transaction will be aborted. In NEMO-VECTOR, we can save this abort if $VCZ_2[1]$ is less than or equal to, the transaction's starting time. In other words, this condition means that the transaction that wrote X_{Z_2} did not invalidate T_1's read-set as it did not write to any object from Z_1. Our experiments showed that this condition saves 10% of the false conflict cases.

It is important to mention again that the working set of each transaction should be mostly local to reduce the communication overhead on the shared bus between NUMA zones. For this reason, we also localize meta-data alongside of the data itself. For example, all objects on NUMA zone Z_1 are protected by a lock table maintained by Z_1 itself (e.g., the lock table is partitioned across NUMA zones). That way, in our approach transactions on different NUMA zones can proceed in a completely-isolated manner if their operations are all local.

Another important issue to address with NEMO is the way memory is allocated. In order to

maintain data locality, newly allocated memory must be placed on a specific NUMA zone (e.g., adding a new element to a linked list in zone Z_1 must allocate the new list node on Z_1 memory). NEMO provides a custom memory allocator to properly organize an application's shared memory.

7.4 Algorithm Details

In this section NEMO-TS and NEMO-VECTOR are detailed.

7.4.1 Nemo-TS

Figure 7.4 shows the pseudo-code of NEMO-TS's core operations. In the following subsections we refer to a specific pseudo-code line using the notation [Line X].

```
tx_begin()
1.  if (cache_is_too_old)
2.    foreach (z in numa_zones)
3.       start_time[z] = timestamps[z]
4.  else
5.    start_time[numa_zone] = timestamps[numa_zone]
tx_read(addr)
6.  if (write_set.exist(addr))
7.      return write_set.find(addr);
8.  zone = zone_of(addr);
9.  hash = hash(addr);
10. entry = &lock_table[zone][hash];
11. v1 = entry->version;
12. val = *addr;
13. v2 = entry->version;
14. if (v1 > start_time[zone] || v1 != v2 || entry->lock)
15.    start_time[zone] = timestamps[zone]
16.    tx_abort();
17. read_set.add(addr);
18. return val;
tx_write(addr, val)
19. write_set.add(addr, val);
20. touched_zones[zone_of(addr)] = true
tx_abort()
21. foreach (w in write_set)
22.    if (w.acquired)
23.        zone = zone_of(w.addr);
24.        hash = hash(w.addr);
25.        lock_table[zone][hash].lock = 0; //unlock
26. tx_restart()
```

```
tx_commit()
27. if (write_set.is_empty()) return; //read-only tx
28. foreach (w in write_set)
29.     zone = zone_of(w.addr);
30.     hash = hash(w.addr);
31.     entry = &lock_table[zone][hash];
32.     if (entry->lock == id) continue;
33.     if (!CAS(entry->lock, 0, id))
34.         tx_abort();
35.     else
36.         w.acquired = true;
    //validate read-set
37. foreach (r in read_set)
38.     zone = zone_of(r.addr);
39.     hash = hash(r.addr);
40.     entry = &lock_table[zone][hash];
41.     if (entry->version > start_time[zone]
            || (entry->lock && entry->lock != id))
42.         tx_abort();
43. write_set.writeback();
44. foreach (z in numa_zones)
45.     if (touched_zones[z])
46.         end_timestamp[z] = atomic_inc(timestamps[z]);
47.         start_time[z] = end_timestamp[z];
48. foreach (w in write_set)
49.     zone = zone_of(w.addr);
50.     hash = hash(w.addr);
51.     entry = &lock_table[zone][hash];
52.     entry->version = end_timestamp[zone];
53.     entry->lock = 0; //unlock
```

Figure 7.4: NEMO-TS's pseudo-code.

Protocol Meta-data

NEMO-TS uses the following thread-local and NUMA-local meta-data (i.e., meta-data shared by all threads belonging to certain NUMA zone).

Thread-local Meta-data. Each thread (and transaction given that a thread can execute only one transaction at a time) records a local:

- *read-set*, a list of all objects read by the transaction. It is used to validate that the transaction has seen a consistent view.

- *write-set*, a write-buffer of all written objects. It is implemented as a hash table to enhance the get and update operations.

- *start-time*, an array storing all the NUMA zones timestamps seen by the thread. It contains an entry for each NUMA zone in the system.

- *touched-zones*, an array that shows which NUMA zone the transaction wrote to.

NUMA-local Meta-data. Each NUMA zone has its local:

- *timestamp*, a counter that is incremented with every writing transaction that touches its associated NUMA zone. Its value is used to mark the versions of objects' locks in the lock-table and know the chronological order of transactions.

- *lock-table*, a large array of versioned locks where an object is mapped to its associated lock using a hash function.

Begin

NEMO-TS reads the transaction's NUMA-zone timestamp at the beginning of each transaction and update the *start-time* entry of that zone [Line 5]. Other entries in *start-time* remain unchanged from the previous transaction executed by the current thread.

Periodically, we update all *start-time* entries when the cache is outdated [Line 1-3]. Different policies can be used to guess that the cache is outdated. One policy is to use the actual time; thus the cache is updated periodically. Another policy is to use a probabilistic function. In our implementation we use a probabilistic function that updates the cache 1/50 of the time.

Transactional Read and Write Operation

A read operation checks first if the requested object was previously written by the transaction. In that case, the buffered value is returned from the write-set [Line 6-7].

The read operation reads the object's version before and after reading the object itself [Line 11-13] to ensure that nothing changed between reading the object and its associated version. If these two versions do not match, the object is locked. If the object version is greater than the transaction *start-time* of that object's NUMA zone [Line 14], then the transaction updates the local cache and aborts [Line 15-16]. Otherwise, the object is added to the read-set and the read value is returned [Line 17-18].

The write operation is much simpler: the written object is added to the write-set and the object's zone is marked as touched [Line 19-20]

The function `zone_of` is used to get the NUMA zone of an object. Since we have our own NUMA memory allocator, we know the address range of each NUMA zone memory in out process address space.

Commit and Abort

If a transaction is read-only (i.e., no write operation is performed), then nothing more is needed and the commit operation immediately returns [Line 27]. For write transactions, a commit operation starts by acquiring locks of all write-set entries [Line 28-36]. In some cases, two objects from the write-set map to the same lock-table entry due to a hash collision. This case is handled by marking the lock owner with the thread id [Line 33]. If an object is already locked by the same thread, it is simply skipped [Line 32]. If acquiring any lock fails, then the transaction aborts [Line 34].

After successfully acquiring all write-set objects' locks, the read-set has to be validated. The validation confirms that the read-set that was used to produce the transaction write-set is still consistent. The validation is done by confirming that read-set entries' versions are still less than or equal to the transaction start-time of each object's zone. It also confirms that read-set objects are not locked by other transactions [Line 37-42].

At this stage, the transaction can safely write-back the write-set buffer to the main memory [Line 43]. Then, the transaction atomically increment all touched zones timestamps and updates the local cache [Line 44-47]. The new timestamps' values are used as the new versions of the written objects [Line 52]. The last step in the commit operation is to update all write-set objects' versions and then unlock them [Line 48-53].

7.4.2 Nemo-Vector

Figure 7.5 shows the pseudo-code of NEMO-VECTOR's core operations.

Protocol Meta-data

NEMO-VECTOR uses the same meta-data of NEMO-TS except for timestamps. Timestamps are replaced with *vector-clocks*. A *vector-clock* is an array that represents a NUMA zone view of all NUMA zones timestamps. It contains an entry for each NUMA zone in the system. Entry z inside *vector-clock* of NUMA zone z represents the zone z timestamp and is always up-to-date. Other entries represent NUMA-level caches of other zones' timestamps.

```
tx_begin()
1.  foreach (z in numa_zones)
2.     start_time[z] = vectors[numa_zone][z]

tx_read(addr)
3.  if (write_set.exist(addr))
4.      return write_set.find(addr);
5.  zone = zone_of(addr);
6.  hash = hash(addr);
7.  entry = &lock_table[zone][hash];
8.  v1 = entry->version;
9.  val = *addr;
10. v2 = entry->version;
11. if (v1 != v2 || entry->lock)
12.    tx_abort();
13. if (v1 > start_time[zone])
14.    if (zone != numa_zone)
15.       if (vectors[numa_zone][zone] < v1)
16.          vectors[numa_zone].lock();
17.          vectors[numa_zone][zone] = vectors[zone][zone];
18.          vectors[numa_zone].unlock();
19.       if (!touched_zones[zone])
20.          foreach (z in touched_zones)
21.             if (vectors[zone][z] > start_time[z])
22.                tx_abort();
23.          start_time[zone] = vectors[zone][zone];
24.    else
25.       tx_abort();
26. touched_zones[zone] |= READ;
27. read_set.add(addr);
28. return val;

tx_write(addr, val)
29. write_set.add(addr, val);
30. touched_zones[zone_of(addr)] |= WRITE;

tx_abort()
31. foreach (w in write_set)
32.    if (w.acquired)
33.        zone = zone_of(w.addr);
34.        hash = hash(w.addr);
35.        lock_table[zone][hash].lock = 0; //unlock
36. tx_restart()

tx_commit()
37. if (write_set.is_empty()) return; //read-only tx
38. foreach (w in write_set)
39.     zone = zone_of(w.addr);
40.     hash = hash(w.addr);
41.     entry = &lock_table[zone][hash];
42.     if (entry->lock == id) continue;
43.     if (!CAS(entry->lock, 0, id))
44.         tx_abort();
45.     else
46.         w.acquired = true;
    //validate read-set
47. foreach (r in read_set)
48.     zone = zone_of(r.addr);
49.     hash = hash(r.addr);
50.     entry = &lock_table[zone][hash];
51.     if (entry->version > start_time[zone]
              || (entry->lock && entry->lock != id))
52.         tx_abort();
53. write_set.writeback();
54. foreach (z in numa_zones)
55.     if (touched_zones[z] & WRITE)
56.        tx_zones.add(z)
57. max_ts = 0;
58. foreach (z in tx_zones)
59.    vectors[z].lock();
60.    if (vectors[z][z] > max_ts)
61.          max_ts = vectors[z][z];
62. max_ts++;
63. foreach (z in tx_zones)
64.   foreach (z2 in tx_zones)
65.      vectors[z][z2] = max_ts;
66.   vectors[z].unlock();
67. foreach (w in write_set)
68.     zone = zone_of(w.addr);
69.     hash = hash(w.addr);
70.     entry = &lock_table[zone][hash];
71.     entry->version = max_ts;
72.     entry->lock = 0; //unlock
```

Figure 7.5: NEMO-VECTOR's pseudo-code.

Begin

A transaction begins by copying its NUMA zone *vector-clock* to its *start-time* [Line 1-2]. In pseudo-code, we represent *vector-clock* as a 2-D array named `vectors`, where `vectors[z]` represents the *vector-clock* of zone z.

Transactional Read and Write Operation

The read operation in NEMO-VECTOR is changed such that we can avoid some of the false conflicts of NEMO-TS. When an object X_z with a newer version is read from another NUMA zone z [Line 13-14], we first update the *vector-clock* entry of that thread's zone [Line 15-18]. Before updating the *vector-clock*, we must acquire its associated lock first [Line 16].

If this is the first time reading from NUMA zone z, then an abort can be avoided if NUMA zone z did not invalidate any of the transaction's touched zones. The condition $VCZ_z[w] \leq VCZ_w[w]$ guarantees that zone z did not change any object in zone w since $VCZ_z[w]$. We want to know, however, if any transaction in zone z invalidated any object in zone w since transaction T started. The condition is changed to $VCZ_z[w] \leq ST_T[w]$ where ST_T is the

start-time of transaction T. If this condition is true for all zones touched by T [Line 20-22] then the read is valid and we can safely advance the *start-time* of zone z ($ST_T[z]$) to the current timestamp of zone z ($VCZ_z[z]$) [Line 23].

Another change in NEMO-VECTOR is that we need to keep track of all zones that a transaction read from them. This is important to know if reading for the first time from a given zone[Line 19, 26, 30]. The *touched-zones* array now is used to mark both read and written zones using different flags.

Commit and Abort

The commit operation is the same as NEMO-TS until the write-back stage [Line 53]. The main difference is how we update the *vector-clocks*. First, we acquire the locks of all touched NUMA zones (touched in a write operation) [Line 58-59]. Then we find the maximum timestamp `max_ts` in all touched zones' timestamps [Line 58-61]. This maximum value is incremented and used to update touched zones' *vector-clocks* entries (the entries of touched zones only) [Line 62-65]. Then the *vector-clocks* locks are released [Line 66].

Finally, the maximum timestamp value `max_ts` is used as the new version of the transaction's written objects [Line 67-72].

Lock-free version of Nemo-Vector

Locking the entire *vector-clocks* to update them is costly; thus, we designed a lock-free version of NEMO-VECTOR. Figure 7.6 shows the pseudo-code of the lock-free version of NEMO-VECTOR's core operations. This version is identical to the original NEMO-VECTOR except for how *vector-clocks* are updated.

In read operation [Line 16-19], instead of locking the entire *vector-clock*, we use an atomic *compare-and-swap (CAS)* operation to update the outdated entry. We use the version of CAS that returns the old value [Line 17]. Using this old value, we can know if another thread updated the same entry and finished the job (if the entry value is now greater than or equal to the desired value) [Line 16, 18]. Otherwise, we retry the CAS operation until the entry is updated (or some other thread updates it)

In commit operations, we care about incrementing the timestamps of touched zones correctly (using atomic increment) [Line 58-59]. Then we try to update the other cache entries in each *vector-clock*. The update is done in a similar fashion of the read operation. Using the version of CAS that returns the old value, we keep trying to update an entry until it reaches the desired value (via a successful CAS or by another thread) [Line 60-67].

Our argument that this lock-free way of updating *vector-clocks* is safe is as follows. By atomically incrementing the main timestamps entry in each *vector-clock*, we guarantee the

```
tx_begin()
1.  foreach (z in numa_zones)
2.     start_time[z] = vectors[numa_zone][z]

tx_read(addr)
3.  if (write_set.exist(addr))
4.       return write_set.find(addr);
5.  zone = zone_of(addr);
6.  hash = hash(addr);
7.  entry = &lock_table[zone][hash];
8.  v1 = entry->version;
9.  val = *addr;
10. v2 = entry->version;
11. if (v1 != v2 || entry->lock)
12.    tx_abort();
13. if (v1 > start_time[zone])
14.    if (zone != numa_zone)
15.       ts = vectors[numa_zone][zone];
16.       while (ts < v1)
17.          old_val = CAS_val(vectors[numa_zone][zone], ts,
                               vectors[zone][zone]);
18.          if (old_val > ts)
19.             ts = old_val
20.       if (!touched_zones[zone])
21.          foreach (z in touched_zones)
22.             if (vectors[zone][z] > start_time[z])
23.                tx_abort();
24.          start_time[zone] = vectors[zone][zone];
25.    else
26.       tx_abort();
27. touched_zones[zone] |= READ;
28. read_set.add(addr);
29. return val;

tx_write(addr, val)
30. write_set.add(addr, val);
31. touched_zones[zone_of(addr)] |= WRITE;

tx_abort()
32. foreach (w in write_set)
33.    if (w.acquired)
34.         zone = zone_of(w.addr);
35.         hash = hash(w.addr);
36.         lock_table[zone][hash].lock = 0; //unlock
37. tx_restart()

tx_commit()
38. if (write_set.is_empty()) return; //read-only tx
39. foreach (w in write_set)
40.     zone = zone_of(w.addr);
41.     hash = hash(w.addr);
42.     entry = &lock_table[zone][hash];
43.     if (entry->lock == id) continue;
44.     if (!CAS(entry->lock, 0, id))
45.         tx_abort();
46.     else
47.         w.acquired = true;
    //validate read-set
48. foreach (r in read_set)
49.     zone = zone_of(r.addr);
50.     hash = hash(r.addr);
51.     entry = &lock_table[zone][hash];
52.     if (entry->version > start_time[zone]
               || (entry->lock && entry->lock != id))
53.         tx_abort();
54. write_set.writeback();
55. foreach (z in numa_zones)
56.     if (touched_zones[z] & WRITE)
57.        tx_zones.add(z)
58. foreach (z in tx_zones)
59.    end_ts[z] = atomic_inc(vectors[z][z]);
60. foreach (z in tx_zones)
61.    foreach (z2 in tx_zones)
62.       if (z != z2)
63.          ts = vectors[z][z2];
64.          while (ts < end_ts[z2])
65.             old_val = CAS_val(vectors[z][z2], ts, end_ts[z2]);
66.             if (old_val > ts)
67.                ts = old_val
68. foreach (w in write_set)
69.     zone = zone_of(w.addr);
70.     hash = hash(w.addr);
71.     entry = &lock_table[zone][hash];
72.     entry->version = end_ts[zone];
73.     entry->lock = 0; //unlock
```

Figure 7.6: Lock-free version of NEMO-VECTOR's pseudo-code.

correctness of the objects' new versions. Then, while updating the cache entries, the write-set entries are locked and other transactions cannot read from them. Thus, until the operation is finished, no invalid object can be read. In addition, using atomic CAS operation to update the cache ensures that threads do not overwrite the values of each other and guarantees that the final value matches the latest increment of each timestamp. This is because each thread tries to update the cache entry with its own value, but it stops in case some other thread updated the cache entry with a greater value.

7.4.3 NUMA Memory Allocator

A NUMA library (i.e., *libnuma* in Linux) provides an API to allocate memory in a specific NUMA zone memory (`numa_alloc_onnode`). `numa_alloc_onnode` allocates the requested memory size rounded up to a multiple of the system page size. In addition, `numa_alloc_onnode` is relatively slow compared to `malloc`. Thus, we decided to build our own NUMA memory allocator, which uses `numa_alloc_onnode` internally. For example, to allocate an `int` using `numa_alloc_onnode`, you will end up with a whole system page. In our NUMA allocator, we consume the page returned from `numa_alloc_onnode` completely before requesting another

page.

7.4.4 Nemo and multi-sockets HTM architectures

Next generation HTM processors from Intel (Haswell-EX, e.g., Xeon E7-48xx v3 and Xeon E7-88xx v3) were introduced in May 2015. Each processor has a high core count (up to 18 cores) and supports multi-sockets operation (up to 8 sockets). These processors adopt a NUMA architecture in their multi-socket operations. In addition, HTM support still comes from cache coherence protocol; thus, HTM will suffer from the same scalability problems of non-HTM NUMA architectures.

Our approach that decouples intra-NUMA transactions from inter-NUMA transactions can be extended to support HTM transactions. Clearly, we cannot change the hardware - thus the way contention between hardware transactions is handled cannot be modified, but our approach can improve the software fallback path. Using a single global lock will definitely impact the performance similar to NOrec results (see Section 7.5). A NUMA-aware global lock can enhance performance, but it still reduces concurrency as only one transaction will be allowed to run at a time. NEMO can be extended to allow concurrency between HTM and the software fallback path by relying on the solution presented in Chapter 6.

7.5 Evaluation

NEMO has been implemented in C++. Since NEMO is a solution to provide scalable performance in presence of scalable workload, we modify our tested benchmarks to be NUMA-local - namely, to exploit data locality of the NUMA zone which the transaction is executing on. We conduct a comprehensive evaluation using the following benchmarks: Bank, Linked-List, and TPC-C. *Bank* mimics a monetary application that transfers capital between accounts. We modify Bank to be NUMA-local by partitioning accounts among NUMA zones. Inter-NUMA-zone operations represent operation that work on accounts stored on different NUMA zones (e.g., a transfer from an account in the NUMA zone z_1 to an account in the NUMA zone z_2). *NUMA-Linked-list* is a new benchmark where we place a separate and independent linked-list in each NUMA zone. Inter-NUMA-zone operations represent moving an object from one NUMA zone to another (e.g., a remove operation from the linked-list of the NUMA zone z_1 followed by an add operation in the linked-list of the NUMA zone z_2). *TPC-C* [27] is the famous on-line transaction processing (OLTP) benchmark which simulates an ordering system on different warehouses. TPC-C includes five transaction profiles, three of which are write transactions and two of which are read-only. TPC-C is modified to be NUMA-local by partitioning the in-memory database tables that represent it. Each group of warehouses is located in a single NUMA zone along with all its related data. Since TPC-C default transaction profiles work on a single warehouse, we choose to represent inter-NUMA operations

by running a transaction on a remote warehouse.

As competitors, we include two state-of-the-art centralized STM protocols, TL2 [35] and NOrec [29]; two NUMA-optimized STM protocols, TLC [12] and TL2 GV5 [35]; and we also develop a version of NOrec enhanced by our implementation of the NUMA-aware lock of [36] (specifically C-BO-BO Lock [36]). In addition, we also compare against strict 2-Phase Locking (2PL), which is a complete DAP approach that strictly uses no shared meta-data. TLC has no shared global timestamp at all, but instead each thread has its own timestamp. In addition, a thread-local cache of other threads' timestamps is maintained in each thread. When a thread reads an object with a version newer than its local cache, it aborts and updates its local cache. The object version includes both the writing thread id and that thread's timestamp at the time of writing. TL2 GV5 is a version of TL2 that limits updates of the shared global timestamp to aborted transactions (when a timestamp conflict is detected). When a transaction reads an object with a version that is greater than the global timestamp, the transaction aborts and updates the global timestamp with the object's version.

The NUMA-aware lock implementation is not available, so we implemented it by using the provided description in [36]. The tests of our implementation of the NUMA-lock show that it is working correctly. Figure 7.7 shows a comparison of a centralized lock and our implementation of the NUMA-lock. In this experiment, each thread tries to acquire the lock 100,000 times, performs some dummy work, and then releases the lock. The reported data is the average time spent by each thread to finish the task; thus lower is better. It is worth noting that threads are distributed among NUMA zones in a round-robin fashion (e.g., the configuration with 8 threads means 1 thread per NUMA zone, while 16 threads means 2 threads per NUMA zone). Thus, lock cohorting of the NUMA-lock has no benefits at 1 and 8 threads.

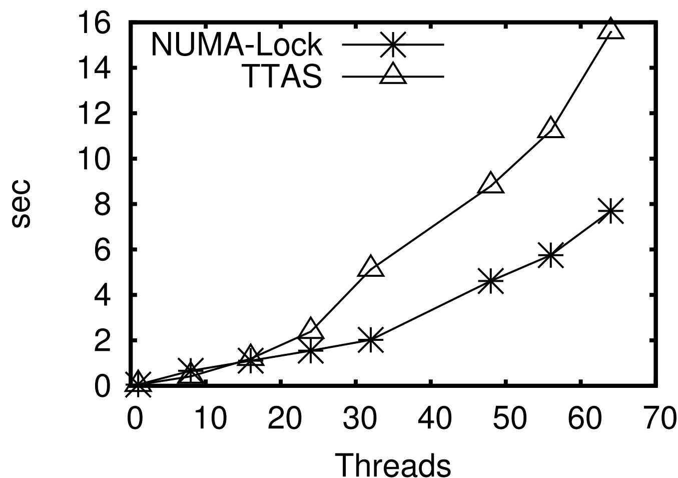


Figure 7.7: Comparison of a centralized global lock and a global NUMA-lock.

In this evaluation study we use the 64-core machine described in Section 7.2 and Figure 7.1. It has four AMD Opteron 6376 Processors (2.3 GHz) and 128 GB of memory. This machine has 8 NUMA zones (2 per chip) and each NUMA zone has 16 GB of the memory. The code is compiled with GCC 4.8.2 using the O3 optimization level. We ran the experiments using Ubuntu 14.04 LTS and libnuma. All data points are the average of 5 repeated execution.

It is worth to note that we refer to scalability as the ability of the system to produce more output under an increased workload when more cores are added. Thus, in our experiments, when we add more cores (threads) and more workload, and get more throughput, we say that the system is scalable. A perfect scalability is achieved when the output is doubled with doubling the number of cores.

Bank

In this benchmark, each transaction produces 10 transfer operations accessing 20 random bank accounts. Each NUMA zone has 1 million accounts (a total of 8 million accounts). Thus, the contention level in this configuration is very low. In addition, 10% of the transactions are inter-NUMA-zone, which means that they invoke a transfer operation between two different NUMA zones.

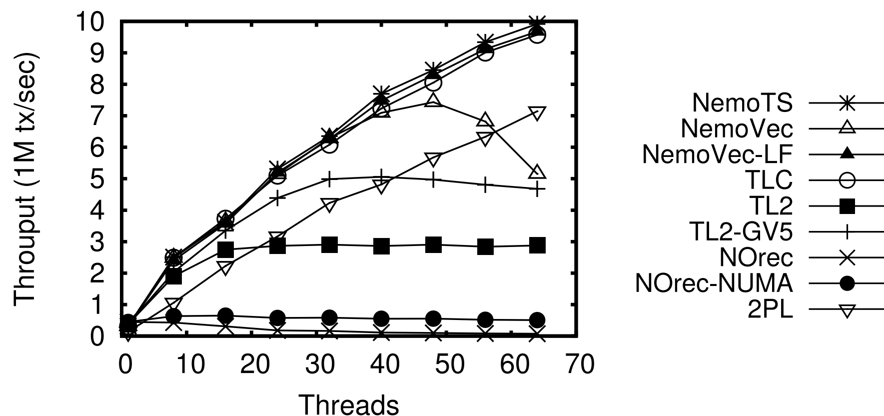


Figure 7.8: Throughput using Bank benchmark.

Figure 7.8 shows the results. NEMO-TS (NemoTS) and the lock-free version of NEMO-VECTOR (NemoVec-LF) have the best performance and scalability. NEMO-VECTOR (NemoVec) starts to suffer at high threads count due to the contention on vector-clocks's locks. TLC has very close performance to NEMO since the level of contention in this benchmark is very low, and thus it does not suffer from a high number of aborts. 2-Phase locking scales well, but the overhead of acquiring locks on both read and written objects at encounter-time is evident, even with such low contention level. TL2-GV5 stopped scaling after 32 threads as it is still using a centralized single timestamp. TL2 does not scale beyond 16 threads (a single

socket). NOrec and NOrec with NUMA-lock (NOrec-NUMA) have very limited scalability up to 16 threads. It is clear that using a NUMA-aware lock enhances performance, but serializing the commit phase in a write-dominated benchmark kills performance.

NUMA-Linked-list

In this benchmark, each NUMA zone has a sorted linked-list of size 10000 elements. Initially the linked-list is half-empty. Each transaction does an insert (30%), a remove (30%), or a contains (40%) operation on a single linked-list. Inter-NUMA-zone transactions removes an item from one linked-list and add it to another linked-list. Given the large size of the linked-lists, transaction execution time is long. In addition, each transaction traverses the linked-list from the beginning to the desired node. During this traversal, all visited nodes are kept in the transaction read-set. Thus, the contention level is high since any write to a node that is read by another transaction will abort the reading transaction.

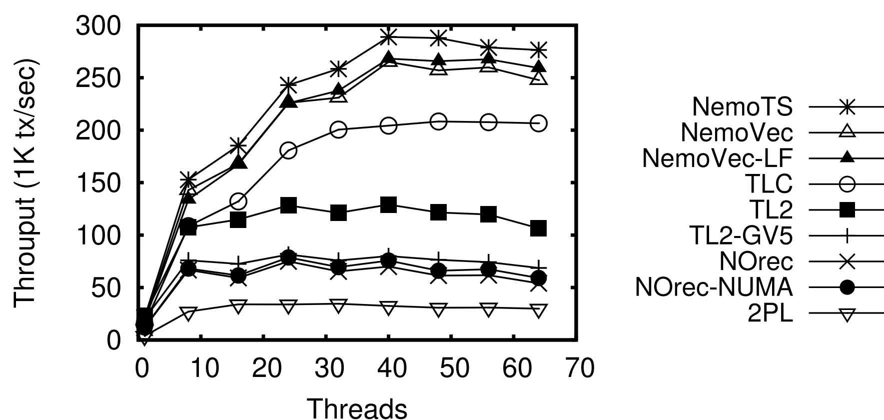


Figure 7.9: Throughput using NUMA-Linked-list benchmark.

Figure 7.9 shows the results. In this benchmark, we notice that all approaches cannot scale well; the level of contention is high and the cost of aborting a transaction is high as well given the transaction's long duration. All NEMO approaches are the best in terms of scalability and performance. TLC suffers from high abort rates in this benchmark because of the outdated cache in each thread. TL2 shows no scalability after 8 threads. TL2-GV5 also suffers from high number of aborts since transactions access common objects frequently and the global timestamp is not updated with every write. Thus, even a single thread aborts every other transaction when it reads an object written in the previous transaction (by the same transaction). NOrec and NOrec-NUMA show better performance since there is 40% read-only transactions which can proceed concurrently. 2PL suffers significantly in this benchmark because transactions are blocked by read-locked objects and are aborted.

TPC-C

In this benchmark, we partition TPC-C in a memory database such that each NUMA zone has 20 warehouses with its associated data. Inter-NUMA transactions are actually out-of-NUMA transactions in this benchmark, where a thread queries or updates a remote partition. TPC-C transactions are complex and long, and it also has a medium level of contention given 20 warehouses per zone. This workload represents the sweet spot for NEMO.

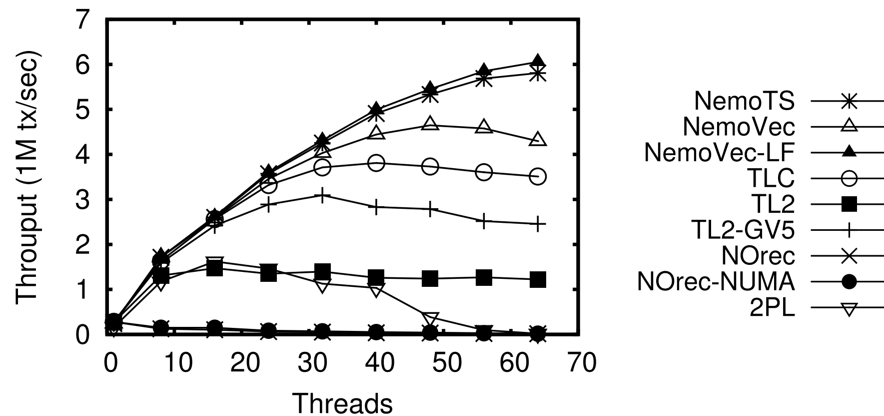


Figure 7.10: Throughput using TPC-C benchmark.

Figure 7.10 shows the results. NemoTS and NemoVec-LF have the best scalability. In addition, NemoVec-LF is better than NemoTS since it is better optimized and the moderate contention workload allows these optimizations to pay-off. As the number of threads increases, NemoVec suffers from a bottleneck due to the vector-clocks locking. TLC also suffers from a large number of aborts in this benchmark. TL2 did not scale after 16 threads. TL-GV5 shows the effect of relaxing the contention of centralized global meta-data. As the contention level is moderate, TL-GV5 is able to scale up to 32 threads. 2PL suffers in moderate contention workloads too. Acquiring objects' read-locks increases the level of contention and the abort rate. NOrec and NOrec-NUMA do not scale at all as 92% of TPC-C's default workload is write transactions.

7.5.1 Effect of Inter-NUMA zones Transactions

In this experiment we show the effect of increasing the percentage of transactions accessing objects stored in different NUMA zones. Figure 7.11 show the results of the Bank benchmark with the same configuration as shown above. We fix the number of thread to 48 threads so that we have enough contention in each NUMA zone without saturating it.

Clearly, NEMO is not originally designed to support a high number of non-NUMA-local transactions. The results shows that NEMO-VECTOR cannot scale beyond 10% because of

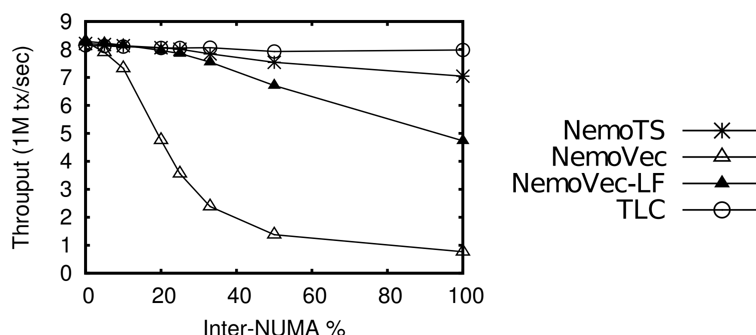


Figure 7.11: Throughput using Bank benchmark under different inter-NUMA-zone transactions percentage. The number of threads is 48.

the bottleneck introduced by the locks on the vector-clocks. The lock-free version of Nemo-Vector scales much better up to 25%. Nemo-TS has the best scalability. Practically, Nemo is designed to handle up to 10% inter-NUMA-zone transactions.

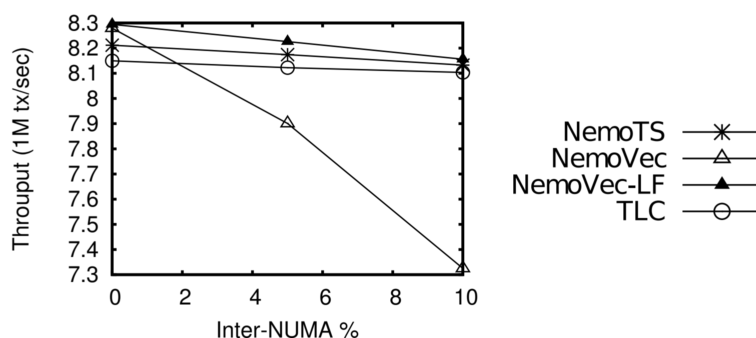


Figure 7.12: Zooming Figure 7.11 in between the 0% datapoint and the 10% datapoint.

Figure 7.12 reports the detailed performance in the range of 0% to 10% of Figure 7.11. Here we see that the lock-free version of Nemo-TS has the best results. TLC is slightly affected with the percentage of inter-NUMA-zone transactions as it has no overhead related to updating the different NUMA zones meta-data.

7.5.2 Summary

Our experiments results show that Nemo has the best performance and scalability when the majority of the workload is scalable (i.e., it minimizes the operations involving more than one NUMA zone). In addition, our best results are achieved when the contention level is medium/high. Nemo-Vector's performance does not match our other approaches because it suffers from the bottleneck due to vector-clocks locking. This bottleneck has been

eliminated in the lock-free version of Nemo-Vector that achieved the best performance in the low and medium contention scenarios.

Chapter 8

Conclusions

In this dissertation, we proposed contributions aimed at optimizing the performance of concurrent applications by leveraging Transactional Memory (TM). We exploited HTM as one of the best candidates for improving performance of applications deployed on multi-core architectures, and easing the parallel programming. Our target is to overcome most of best-effort HTM limitations. We identified three major limitations: resource limitations; lack of an advanced contention manager; and poor communication techniques between HTM and the software fallback path. 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 addressed the communication problem by a fine-grained fallback mechanism in Precise-tm. Another important problem that affects TM systems in general is scalability on Non-Uniform Memory Access (NUMA) architectures, a problem that affects current STM systems and will affect HTM systems in the near future. Nemo addresses this problem by proposing a NUMA-aware design, and by supporting NUMA-locality.

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 into multiple sub-transactions and run them in hardware with a minimal instrumentation. Part-htm's performance is appealing. In our evaluation it is the best in almost all the tested workloads, and it is close to HTM's performance where HTM performs the best. Part-htm represents the first solution that solves the resource limitation of best-effort HTM.

Octonauts represents one of the first HTM-aware schedulers. 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.

Precise-tm tackles the problem of the coarse-grained fallback path of best-effort HTM where HTM transactions are interrupted by transactions executing in the software fallback path. It presents a unique and precise technique for a fine-grained fallback path without introducing any share meta-data, which would be a source of high overhead. Precise-tm used the address-embedded lock technique to implement fine-grained locks because it has no space overhead (as it steals bits from pointers) and minimal instrumentation overhead. In addition, HTM transactions and software fallback path communicate with each other only when they access a common object as imposed by the application logic itself. Results show that Precise-tm allows more concurrency between transactions, reduces false conflicts, and minimizes added meta-data.

Finally, we presented Nemo, a NUMA-aware scalable STM algorithm that exploits NUMA-local workloads. Nemo allows intra-NUMA transactions to run efficiently and share NUMA-local meta-data to achieve the best possible performance. Inter-NUMA zones transactions, which represent the uncommon case, can still be executed efficiently but without interference with NUMA-local transactions. Nemo results showed a near perfect scalability for scalable workloads, while other STM algorithms stop scaling after 16 parallel threads in a 64-core AMD multi-core machine.

8.1 Summary of Contributions

Our contributions are summarize as follows:

- Part-htm is the first hybrid TM that solves the problem of resource limitations (space/time) of current best-effort HTM. Part-htm has the best performance in all tested cases that are not suitable for HTM (where HTM cannot be outperformed).
- Octonauts is one of the first HTM-aware schedulers that orchestrates conflicting transactions. One of the main contributions of Octonauts is identifying the issues of HTM-aware schedulers and why solving the well-studied problem of scheduling is not trivial for current best-effort HTM implementations. This is because current HTM design does not support non-transactional instructions and thus it does not provide enough information about aborted transactions so that effective scheduling policies can be defined. Results show performance improvement when Octonauts is deployed in comparison with pure HTM with falling back to global locking.
- Precise-tm is a unique approach to solve the granularity of the software fallback path of best-efforts HTM. It presented a new and precise technique towards fine-grained software fallback path. Results show that our precise fine-grained fallback path allows more concurrency between transactions and reduces false aborts with minimal space/time overhead.

- NEMO is a NUMA-aware STM algorithm that provides scalable performance in the presence of locality-aware workloads. NEMO aims at optimizing the common case of transactions running within one NUMA-zone, and handling inter-NUMA-zone transactions efficiently by minimizing the thread interferences if transactions are not actually conflicting (an idea inspired by the Disjoint-Access-Parallelism property).

8.2 Future Work

As a future work, we suggest extending NEMO to support the newly released NUMA architectures with HTM support integrated with the cache-coherence protocol that manages the consistency of caches on the whole machine, including other sockets and NUMA-zones. We see PRECISE-TM as a good candidate to be adapted and integrated into NEMO, as it already supports the Disjoint-Access-Parallelism property. The focus should be on the software fallback path as the conflict detection of HTM transactions is handled by the hardware itself. A fine-grained fallback path in crucial for HTM in NUMA settings where coarse-grained locking or unnecessary communications degrade performance and scalability.

Another extension for PRECISE-TM is to allow a more efficient STM-STM synchronization technique, which would allow for more concurrency between transactions running in the software fallback path. In addition, a good contention manager would be needed to manage conflicts in the software fallback path itself. Thus, merging OCTONAUTS and PRECISE-TM is another good possible research direction to develop.

Another important direction is to build a mathematical model where we define a set of parameters representing the system variables. Using this model, we can predict the performance. In addition, another model can be developed to approximate the optimal performance we can get for a given workload. For example, a model for PART-HTM can be developed where the number, size, working set and duration of each partition are the parameters of the software component. The hardware parameters include the read/write buffer size, cache line size, the conflict detection technique, the conflict resolution policy, and the number of cores. Using that model we can predict PART-HTM performance without running the new workload. Similar techniques can be used to model OCTONAUTS, PRECISE-TM, and NEMO.

Bibliography

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] Mohammad Ansari, Behram Khan, Mikel Lujn, Christos Kotselidis, Chris Kirkham, and Ian Watson. Improving performance by reducing aborts in hardware transactional memory. In YaleN. Patt, Pierfrancesco Foglia, Evelyn Duesterwald, Paolo Faraboschi, and Xavier Martorell, editors, *High Performance Embedded Architectures and Compilers*, volume 5952 of *Lecture Notes in Computer Science*, pages 35–49. Springer Berlin Heidelberg, 2010.

[7] 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.

[8] Joseph Antony, PeteP. Janes, and AlistairP. Rendell. Exploring thread and memory placement on numa architectures: Solaris and linux, ultrasparc/fireplane and opteron/hypertransport. In Yves Robert, Manish Parashar, Ramamurthy Badrinath, and ViktorK. Prasanna, editors, *High Performance Computing - HiPC 2006*, volume 4297 of *Lecture Notes in Computer Science*, pages 338–352. Springer Berlin Heidelberg, 2006.

[9] Ehsan Atoofian and AmirGhanbari Bavarsad. Maintaining consistency in software transactional memory through dynamic versioning tuning. In Yang Xiang, Ivan Stojmenovic, BernadyO. Apduhan, Guojun Wang, Koji Nakano, and Albert Zomaya, editors, *Algorithms and Architectures for Parallel Processing*, volume 7440 of *Lecture Notes in Computer Science*, pages 40–49. Springer Berlin Heidelberg, 2012.

[10] Hagit Attiya and Eshcar Hillel. Single-version stms can be multi-version permissive (extended abstract). In MarcosK. Aguilera, Haifeng Yu, NitinH. Vaidya, Vikram Srinivasan, and RomitRoy Choudhury, editors, *Distributed Computing and Networking*, volume 6522 of *Lecture Notes in Computer Science*, pages 83–94. Springer Berlin Heidelberg, 2011.

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

[12] Hillel Avni and Nir Shavit. Maintaining consistent transactional states without a global clock. In AlexanderA. Shvartsman and Pascal Felber, editors, *Structural Information and Communication Complexity*, volume 5058 of *Lecture Notes in Computer Science*, pages 131–140. Springer Berlin Heidelberg, 2008.

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

[17] 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.

[18] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. Numa-aware reader-writer locks. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 157–166, 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] 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.

[23] Kinson Chan and Cho-Li Wang. Trc-mc: Decentralized software transactional memory for multi-multicore computers. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 292–299, Dec 2011.

[24] 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.

[25] Hypertransport Technology Consortium et al. Hypertransport i/o link specification revision 3.00. *Document# HTC20051222-0046-0008*, 2006.

[26] 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`.

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

[28] 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.

[29] 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.

[30] 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.

[31] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *SOSP '13*, 2013.

[32] Dave Dice, Timothy L. Harris, Alex Kogan, Yossi Lev, and Mark Moir. Pitfalls of lazy subscription. In *WTTM*, 2014.

[33] Dave Dice, Maurice Herlihy, Doug Lea, Yossi Lev, Victor Luchangco, Wayne Mesard, Mark Moir, Kevin Moore, and Dan Nussbaum. Applications of the adaptive transactional memory test platform. In *TRANSACT '08*, 2008.

[34] Dave Dice, Alex Kogan, and Yossi Lev. Refined transactional lock elision. 2015.

[35] 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.

[36] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: A general technique for designing numa locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 247–256, New York, NY, USA, 2012. ACM.

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

[38] Nuno Diegues, Paolo Romano, and Stoyan Garbatov. Seer: Probabilistic scheduling for hardware transactional memory. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, New York, NY, USA, 2015. ACM.

[39] 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.

[40] 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.

[41] 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.

[42] Faith Ellen, Yossi Lev, Victor Luchangco, and Mark Moir. Snzi: Scalable nonzero indicators. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 13–22, New York, NY, USA, 2007. ACM.

[43] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 237–246, New York, NY, USA, 2008. ACM.

[44] 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.

[45] 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.

[46] 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.

[47] Nikos Hardavellas, Ippokratis Pandis, Ryan Johnson, Naju Mancheril, Anastassia Ailamaki, and Babak Falsafi. Database Servers on Chip Multiprocessors: Limitations and Opportunities. In *Proceedings of the Biennial Conference on Innovative Data Systems Research*, 2007. SYSTEMS.

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

[49] 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.

[50] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.

[51] 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.

[52] 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.

[53] Sungpack Hong, Tayo Oguntebi, Jared Casper, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. Eigenbench: A simple exploration tool for orthogonal tm characteristics. In *IISWC*, pages 1–11, 2010.

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

[55] Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, pages 151–160, New York, NY, USA, 1994. ACM.

[56] Tim Kiefer, Benjamin Schlegel, and Wolfgang Lehner. Experimental evaluation of numa effects on database management systems. In *BTW*, pages 185–204, 2013.

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

[58] 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.

[59] Yossi Lev, Victor Luchangco, Virendra Marathe, Mark Moir, Dan Nussbaum, and Marek Olszewski. Anatomy of a scalable software transactional memory. In *4th ACM SIGPLAN Workshop on Transactional Computing*, TRANSACT '09, 2009.

[60] 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.

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

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

[63] Kai Lu, Ruibo Wang, and Xicheng Lu. Brief announcement: Numa-aware transactional memory. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, pages 69–70, New York, NY, USA, 2010. ACM.

[64] 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.

[65] Nakul Manchanda and Karan Anand. Non-Uniform Memory Access (NUMA). *New York University*, 2010.

[66] 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.

[67] Patrick Marlier, Anita Sobe, and Pierre Sutra. A locality-aware software transactional memory. In *Euro-TM Workshop on Transactional Memory (WTM 2014)*, WTM '14, 2014.

[68] 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.

[69] Alexander Matveev and Nir Shavit. Reduced Hardware NOrec: A Safe and Scalable Hybrid Transactional Memory. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 59–71. ACM, 2015.

[70] 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.

[71] 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.

[72] Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M. Michael, and Hisanobu Tomari. Quantitative comparison of hardware transactional memory for blue gene/q, zenterprise ec12, intel core, and power8. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 144–157, New York, NY, USA, 2015. ACM.

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

[74] Sebastiano Peluso, Roberto Palmieri, Paolo Romano, Binoy Ravindran, and Francesco Quaglia. Disjoint-access parallelism: Impossibility, possibility, and cost of transactional memory implementations. Technical report, Technical report, Virginia Tech, 2015.

[75] Sebastiano Peluso, Roberto Palmieri, Paolo Romano, Binoy Ravindran, and Francesco Quaglia. Disjoint-access parallelism: Impossibility, possibility, and cost of transactional memory implementations. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, PODC '15, 2015.

[76] Danica Porobic, Ippokratis Pandis, Miguel Branco, Pinar Tozun, and Anastasia Ailamaki. Oltp on hardware islands. *Proc. VLDB Endow.*, 5(11):1447–1458, July 2012.

[77] 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.

[78] 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.

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

[80] Torvald Riegel, Christof Fetzer, and Pascal Felber. Time-based transactional memory with scalable time bases. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '07, pages 221–228, New York, NY, USA, 2007. ACM.

[81] 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.

[82] Hugo Rito and Joo Cachopo. Props: A progressively pessimistic scheduler for software transactional memory. In Fernando Silva, Ins Dutra, and Vtor Santos Costa, editors, *Euro-Par 2014 Parallel Processing*, volume 8632 of *Lecture Notes in Computer Science*, pages 150–161. Springer International Publishing, 2014.

[83] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, and Emmett Witchel. Txlinux: Using and managing hardware transactional memory in an operating system. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 87–102, New York, NY, USA, 2007. ACM.

[84] Wenjia Ruan, Yujie Liu, and Michael Spear. STAMP need not be considered harmful. In *TRANSACT '14*.

[85] Wenjia Ruan, Yujie Liu, and Michael Spear. Boosting timestamp-based transactional memory by exploiting hardware cycle counters. *ACM Trans. Archit. Code Optim.*, 10(4):40:1–40:21, December 2013.

[86] Wenjia Ruan and Michael Spear. An opaque hybrid transactional memory. 2015.

[87] 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.

[88] Tudor-Ioan Salomie, Ionut Emanuel Subasu, Jana Giceva, and Gustavo Alonso. Database engines on multicores, why parallelize when you can distribute? In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 17–30, New York, NY, USA, 2011. ACM.

[89] 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.

[90] 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.

[91] 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.

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

[93] Lingxiang Xiang and Michael L. Scott. Conflict reduction in hardware transactions using advisory locks. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, New York, NY, USA, 2015. ACM.

[94] 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.

[95] 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.

[96] D. Ziakas, A. Baum, R.A. Maddox, and R.J. Safranek. Intel quickpath interconnect architectural features supporting scalable system architectures. In *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*, pages 1–6, Aug 2010.